SharpImage: An Image Processing Prototyping Environment

Release 1.0

Dan Mueller[1]

February 16, 2008

[1]Queensland University of Technology, Brisbane, Australia

Abstract

This paper describes an image processing prototyping environment called SharpImage. SharpImage is a stand-alone Windows application geared towards the pre-processing and creation of volume illustrations, although other generic tasks are inherently supported. The user interacts with the environment using a command console, and can view images as a series slices or through volume rendering. We describe the design and introduce functionality using a number of examples. We have found the environment useful for prototyping a range of image processing tasks and provide the source-code in the hope it will be useful for others.

Keywords: *SharpImage, ITK, prototyping, scripting*

Contents

1 Introduction

The Insight Toolkit (ITK) [13] is an open-source software system designed primarily for medical image segmentation and registration. The architecture and design of ITK — together with the complied C++ implementation — provides for efficiency, speed, and flexibility.

However, prototyping image processing pipelines in native ITK can be a daunting task. As such a number of systems generate wrappers around common functionality: *WrapITK* [8] uses CableSwig to automatically generate wrappers for Python (Tcl and Java are also supported to a lesser degree by this system); *MATITK* [5] allows some ITK algorithms to be called from MATLAB; and *Managed-ITK* [9] uses a set of templates and CMake scripts to generate wrappers for .NET languages (such as C# and IronPython).

A number of data-flow applications have been built using ITK, allowing users to build pipelines by constructing block diagrams, for example *SCIRun* [2] and *MeVisLab* [1]. Various stand-alone "turn-key" applications also exist, most of which support some form of extensibility, such as *VolView* [4] and *Slicer* [3].

SharpImage can be considered a stand-alone scripting application. It was originally designed to facilitate pre-processing and prototyping of volume illustration techniques, however it also supports generic image processing tasks. The environment can be customised by editing existing scripts, creating and invoking new scripts at run-time, or by adding new window elements (forms). The full source-code is provided in the hope it will be of use to others in the medical image processing community.

2 Design

SharpImage consists of two major components: scripts to perform processing, and renderers to preview images. An application framework is provided to glue these components together.

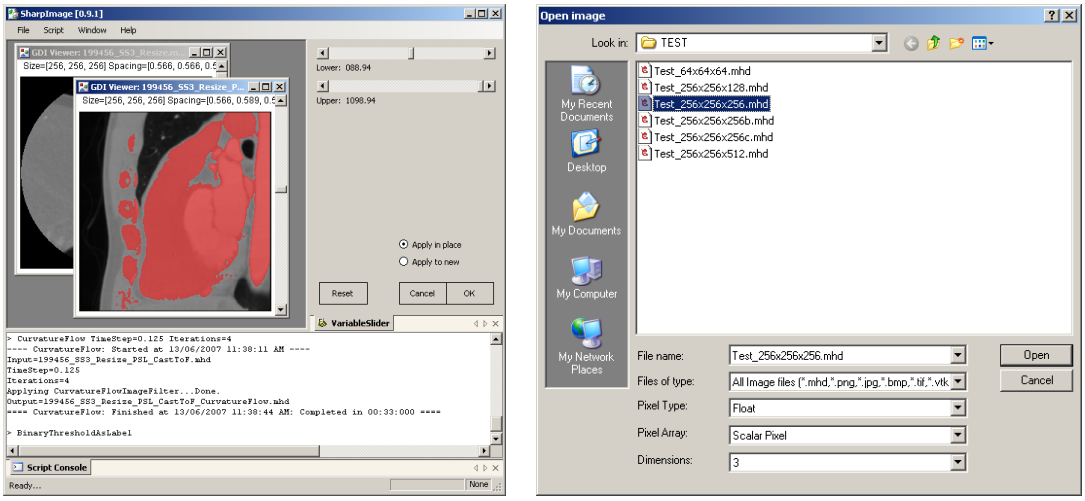
2.1 Framework

The application framework acts a skeleton on which the scripts and renderers are hung. The framework consists of three major components:

Main Window: The main window parents the renderers, script console, and other windows (see Figure 1). It also has a status label and progress bar which scripts use to report on their status.

Script Manager: The script manager is responsible for creating, initialising, running, and monitoring scripts. It parses a command string (eg. `CurvatureFlow TimeStep=0.15 Iterations=8`), extracting the name and parameters. Given the script name, it searches the scripting folder for possible matches, sets the parameters, and starts it running.

Dialogs: A number of dialog windows exist for various tasks including opening and saving images. The most important dialog is the open image dialog which allows the user to specify an image path, type, and dimensionality (see Figure 1). The dialog attempts to automatically detect the correct pixel type and dimensions, but these may be overridden by the user to force the ITK IO module to cast the image to the desired representation.



(a) Main Window (b) Open Image Dialog

Figure 1: The SharpImage application framework components, including the main window and open image dialog.

2.2 Scripts

At the heart of SharpImage is a set of IronPython scripts, most of which operate on images and show their results in new renderers. Many of the existing scripts apply single ITK filters, however more complex pipelines can also be constructed. As depicted in Figure 2, all scripts must derive from the base `Script` class and consist of a number of properties and methods. A majority of scripts inherit from `ImageToImageScript` and consist of the following:

`Name`: A string representing the name of the script.

`Help`: A string describing how to use the script.

`Input`: The input image.

`Output`: The resultant output image.

`Run()`: The entry-point used by the manager to invoke the script.

`Initialise()`: Performs the script initialisation tasks.

`Finalise()`: Performs the script finalisation tasks.

`DoWork()`: Creates a new thread and calls `ThreadedDoWork()`.

`ThreadedDoWork()`: Performs the main function of the script on a background thread.

The use of this script-based architecture is advantageous for a number of reasons: (1) existing scripts can be edited at run-time, (2) new scripts can be easily added and executed immediately, (3) functionality can be extended by inheriting from existing scripts, and (4) scripts can encapsulate functionality allowing for reproducible experiments given only the input image and script parameters.

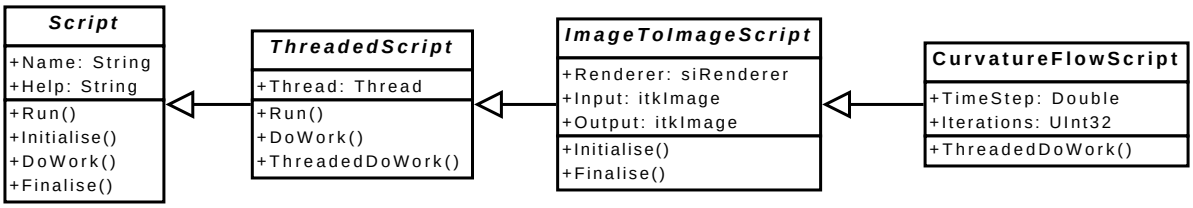


Figure 2: A UML diagram depicting the inheritance hierarchy of SharpImage scripts. In this figure the `CurvatureFlowScript` is used as an example.

2.3 Renderers

There are currently two renderers available in SharpImage: a slice renderer for previewing image slices, and a volume renderer for direct volume rendering. Other renderers (such as those exposed by VTK) could be added if desired.

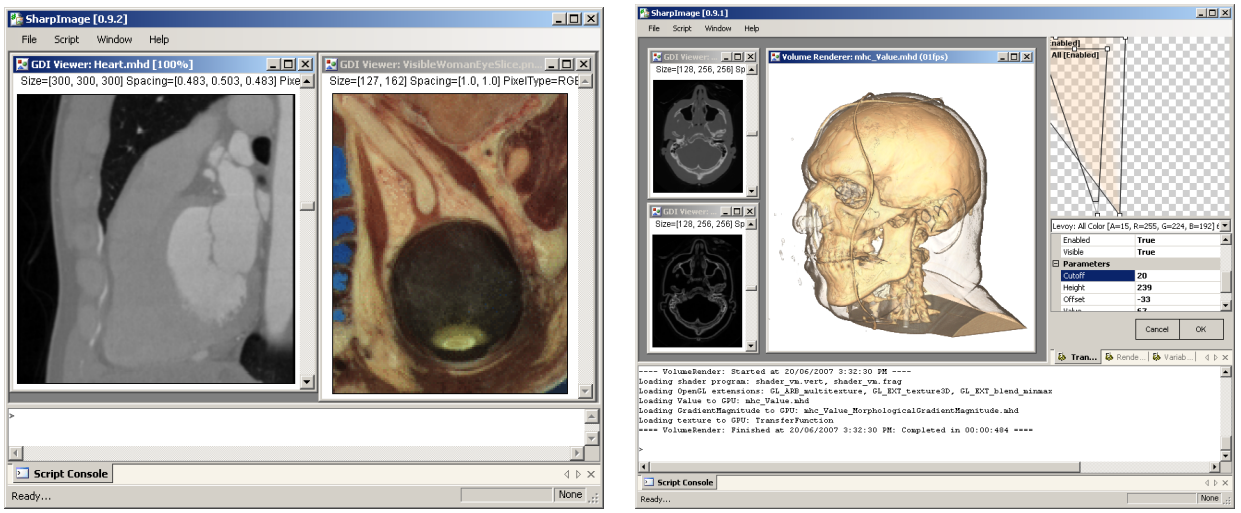
The **slice renderer** (`siGdiSliceRenderer`) uses the Windows GDI+ library to preview a single orthogonal slice though a 3-D image (or the only slice of a 2-D image). This renderer supports the following scalar and vector images: `unsigned char`, `signed short`, `float`, `double`, `RGB`, `RGBA`, `Vector`, `Covariant Vector`, and `Vector Image`. It currently uses 256 greyscales (8-bit) for display and supports multiple transparent overlays (labels).

The **volume renderer** (`siVolumeRenderer`) uses OpenGL to display images using 3-D texture mapping with fragment shaders (see [10, 7]). The user can rotate the volume using the left mouse button, translate using the right mouse button, and scale (zoom in/out) using the mouse wheel. This renderer supports scaling and biasing for multiple pixel representations including `unsigned char`, `signed short` and `float` images.

The `siVolumeRenderer` requires a graphics processing unit (GPU) supporting the following extensions:

- `GL_ARB_fragment_shader`
- `GL_ARB_shader_objects`
- `GL_ARB_multitexture`
- `GL_EXT_texture3D`
- `GL_EXT_blend_minmax`

Any good non-OEM graphics card should support these (such as the ATI Radeon 9800 release in 2003), but for optimal performance we suggest a more recent card (such as the NVIDIA GeForce 8800 GTX released in 2007).



(a) Slice Renderer (b) Volume Renderer

Figure 3: The SharpImage renderers: `siGdiSliceRenderer` and `siVolumeRenderer`.

3 Quick Start Guide

3.1 Installation

We hope to create an installation package at some stage, but at the moment each component must be installed manually:

1. Ensure the .NET Framework 2.0 is installed on your system. The .NET Framework Redistributable Package `dotnetfx.exe` can be obtained from here.
2. Install the Microsoft Visual C++ 2005 Redistributable package (`vcredist_x86.exe`) available from here. The ManagedITK assemblies which are packaged with SharpImage are compiled with Visual Studio 8 SP1, so make sure you install the SP1 redistributable package.
3. Download the SharpImage binary files (which include ManagedITK) from: http://www.insight-journal.org.
4. Unzip the binary files to a suitable directory (eg. `C:/Utils/SharpImage`).
5. Run `SharpImage.exe`.

The source-code is available for both ManagedITK and SharpImage, and can be compiled using the Visual Studio 8.0.50727.762 (SP1) solution files.

3.2 Usage

This section provides the reader with the necessary knowledge to start using SharpImage. We assume you have followed the installation process in the previous section and have a copy of SharpImage open. As a scripting environment, the user interacts by invoking scripts via the Script Console. To display the console either navigate the menu to `Script > Console`, or press `Ctrl + Shift + C`. In order to effectively use SharpImage, you need only memorise two commands: `dir` and `help`.

The `dir [pattern]` command lists all the available scripts with their name or path containing the given `pattern` string (eg. `dir ''Common''`). If no `pattern` is provided, *all* available scripts will be listed. The following example returns all the scripts related to noise suppression:

```
> dir "Denoising"
Scripting\Filtering\Denoising\AdditiveGaussianNoise
Scripting\Filtering\Denoising\Bilateral
Scripting\Filtering\Denoising\CurvatureAnisotropicDiffusion
Scripting\Filtering\Denoising\CurvatureFlow
Scripting\Filtering\Denoising\GradientAnisotropicDiffusion
Scripting\Filtering\Denoising\ImpulseNoise
Scripting\Filtering\Denoising\Mean
Scripting\Filtering\Denoising\Median
Scripting\Filtering\Denoising\MinMaxCurvatureFlow
```

Each script has a `Help` string which can be displayed by calling `help scriptname`, where `scriptname` is the name of the script you want to query (eg. `help Mean`). Each help string consists of three components: the name of the script, a description of the script, and a list of parameters (with default values indicated in brackets). If no default parameter value is shown you are required to always provide a value. For example:

```
> help CurvatureFlow
===== Help: CurvatureFlow =====
Performs denoising of the input image using curvature driven flow.
---- Parameters ----
(double) TimeStep = the finite difference time step. (0.05)
(int) Iterations = the number of iterations. (2)
```

Once you have found your desired script (with `dir`) and you have read the help information (with `help`), you are ready to invoke the script. To invoke a script type the name followed by the parameters you wish to specify (unspecified parameters will assume their default value). By default the script input is the currently selected image. For example:

```
> Open "C:/Temp/BrainProtonDensitySlice.png#F2"
> CurvatureFlow TimeStep=0.15 Iterations=8
---- CurvatureFlow: Started at 8:12:04 AM ----
Input=BrainProtonDensitySlice.png
TimeStep=0.15
Iterations=8
Applying CurvatureFlowImageFilter...Done.
Output=BrainProtonDensitySlice_CurvatureFlow.png
==== CurvatureFlow: Finished at 8:12:04 AM: Completed in 00:00:109 ====
```

4 Examples

In this section we will demonstrate some of SharpImage's core functionality. It loosely follows the organisation of the ITK Software Guide.

4.1 Open

An image can be opened by either using a dialog or the script console. To display the dialog select `File >Open >Open Image...` (see Figure 1). The dialog can also be displayed by invoking the `Open` script with no parameters. To open an image using the `Open` script, specify the full path and type separated by a `#`:

```
> Open
> Open "C:/Temp/cthead1.png#UC2"
> Open "C:/Temp/cthead1.png#SS2"
> Open "C:/Temp/cthead1.png#F2"
> Open "C:/Temp/VisibleWomanEyeSlice.png#RGBUC2"
```

4.2 Save

Again, an image can be saved using either a dialog or the script console. To save the image using the dialog select `File >Save As...` and specify the path, name, and and pixel type (the dimensionality and number of components will handled automatically). To save an image using the `Save` script, specify the full path and type separated by a `#`:

```
> Save "C:/Temp/cthead1.png#UC2"
> Save "C:/Temp/cthead1.png#SS2"
> Save "C:/Temp/cthead1.png#F2"
```

4.3 Change Properties

It is often desirable to list and/or edit the properties of an image and as such the `Properties` script has been provided:

```
> Properties
Name=C:/Temp/cthead1.png
PixelType=Float
Size=[256, 256]
Spacing=[1.00000, 1.00000]
Origin=[0.00000, 0.00000]
Direction=1 0 0 1
Buffer=248840952
MTime=812
```

The Name, Spacing, Origin, and Direction properties can also be changed using this script (use with caution!):

```
> Properties Spacing=itkSpacing(0.5,0.5) Origin=itkPoint(5.0,5.0)
Name=C:/Temp/cthead1.png
PixelType=Float
Size=[256, 256]
*Spacing=[0.50000, 0.50000]
*Origin=[5.00000, 5.00000]
Direction=1 0 0 1
Buffer=248840952
MTime=1414
```

The `Rename` script is handy for quickly renaming an image (the names of images are important because they are used as handles for a number of different scripts):

```
> Rename "Mask1"
```

4.4 Threshold

Global threshold operations can be very effective for simple image processing tasks. The `BinaryThreshold` script sets all pixels in a given range to the given value, both specified using the command line. There are two user friendly specialisations which allow the range to be specified using scroll bars: `BinaryThresholdWithForm` which displays the result as a binary image, and `BinaryThresholdAsLabel` which displays the result as a label over the input image (see Figure 4).

```
> BinaryThreshold Lower=0 Upper=180
> BinaryThresholdWithForm
> BinaryThresholdAsLabel
```

The `Threshold` script leaves unchanged all pixels with intensity values inside the range defined by the two thresholds. Pixels with values outside this range are assigned the `OutsideValue`. Again, there is a user friendly specification allowing the range to the specified using scroll bars:

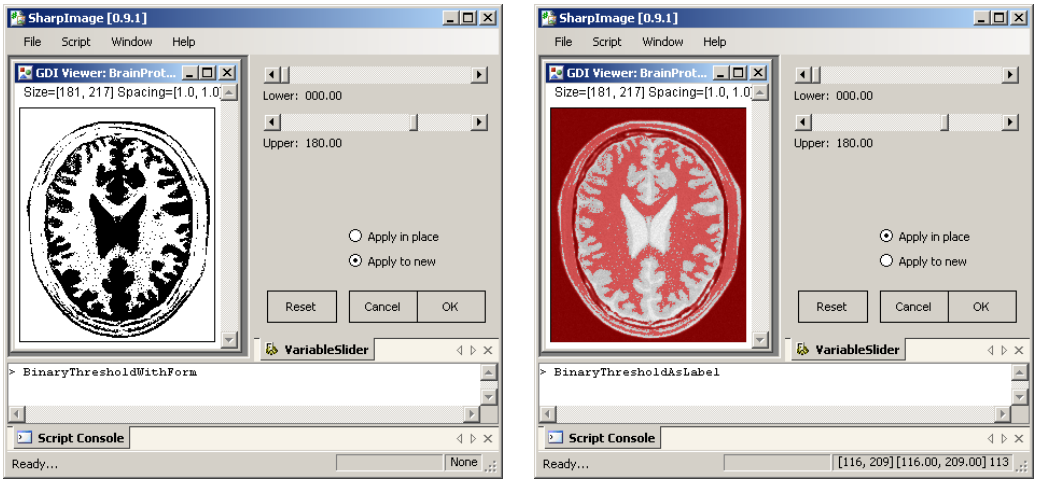
```
> Threshold Lower=0 Upper=180 OutsideValue=0
> ThresholdWithForm OutsideValue=0
```

4.5 Cast

As previously mentioned, SharpImage supports images of different pixel types. By default, the `siGdiSliceRenderer` rescales non `unsigned char` images so that the minimum and maximum values correspond to [0,255]. Figure 5 depicts the same image (consisting of two circles: left=1, right=2) displayed as `unsigned char` and `float`.

It is common to `Cast` one type to another and as such a generic script (plus a number of short-hand aliases) have been provided to facilitate this task:

```
> Cast OutputPixelType=itkPixelType.F
> CastToF
> CastToSS
> CastToUC
```



(a) `BinaryThresholdWithForm` (b) `BinaryThresholdAsLabel`

Figure 4: Binary thresholding: `BinaryThresholdWithForm` displays the result as a binary image, whereas `BinaryThresholdAsLabel` displays the result as a label.

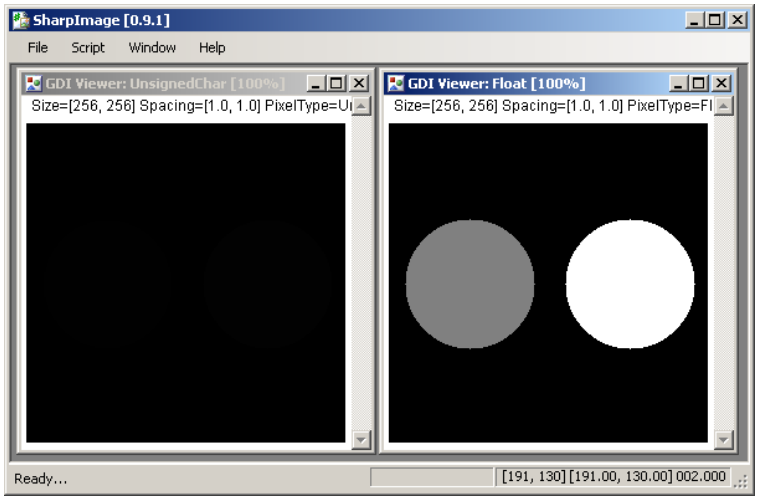


Figure 5: This figure depicts the same image with `unsigned char` pixel type (left) and `float` pixel type (right). The `float` image is automatically rescaled to consume the entire [0,255] range. The `unsigned char` image must be explicitly rescaled using the `RescaleIntenityToUC` command.

4.6 Intensity Mapping

The `RescaleIntensity` script linearly scales the pixel values in such a way that the minimum and maximum values of the input are mapped to minimum and maximum values provided by the user:

```
> RescaleIntensity OutputMinimum=0 OutputMaximum=100
> RescaleIntensity OutputMaximum=100.0 OutputPixelType=itkPixelType.F
```

A number of aliases exist for quickly rescaling an image to use an appropriate range of a given pixel type. `RescaleIntensityToF` rescales the image to [0.0,1.0], `RescaleIntensityToSS` rescales the image to [-32768,32767], and `RescaleIntensityToUC` rescales the image to [0,255]:

```
> RescaleIntensityToF
> RescaleIntensityToSS
> RescaleIntensityToUC
```

The `ShiftScale` script applies a linear intensity mapping by firstly adding `Shift` to each pixel, then multiplying each pixel by `Scale`:

```
> ShiftScale Shift=100.0 Scale=0.5
```

The `Sigmoid` script applies a non-linear intensity mapping useful for focusing attention on a particular set of values and progressively attenuating the values outside that range:

```
> Sigmoid Alpha=10 Beta=170 OutputMinimum=10 OutputMaximum=240
```

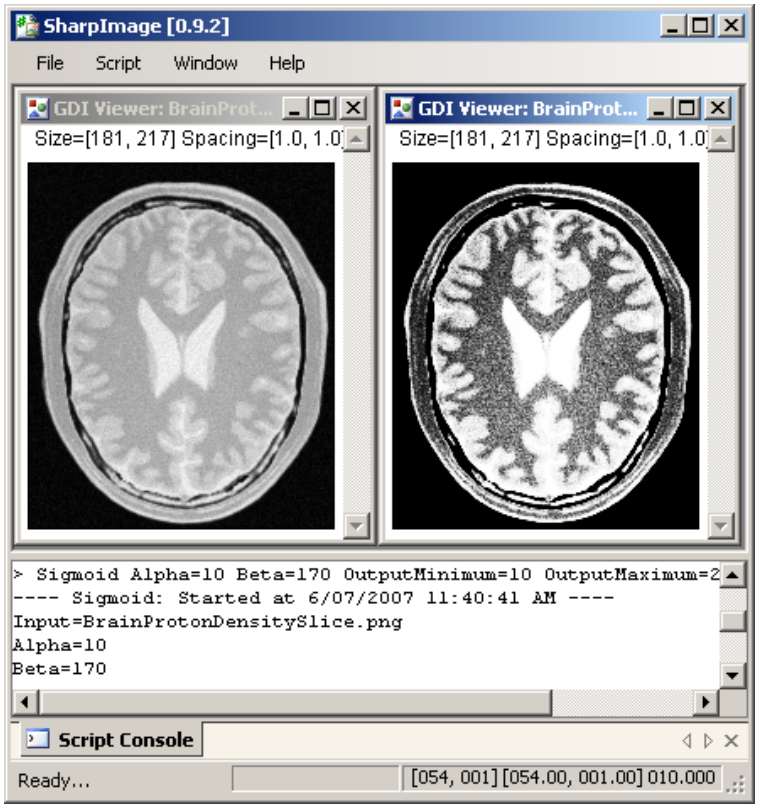


Figure 6: This figure depicts the result of applying the `Sigmoid` script.

4.7 Gradients

Gradient vectors are useful for a variety of image processing tasks, including simple edge detection. SharpImage provides a number scripts for computing the gradient and gradient magnitude. Gradient scripts produce multi-component (vector) images, of which a single channel is shown at one time. To change the channel, select the renderer and type `0`, `1`, or `2`.

```
> Gradient
> GradientRecursiveGaussian Sigma=1.0
```

A number of scripts are also provided to computing the gradient magnitude, a useful edge measure:

```
> GradientMagnitude
> GradientMagnitudeRecursiveGaussian Sigma=1.0
```

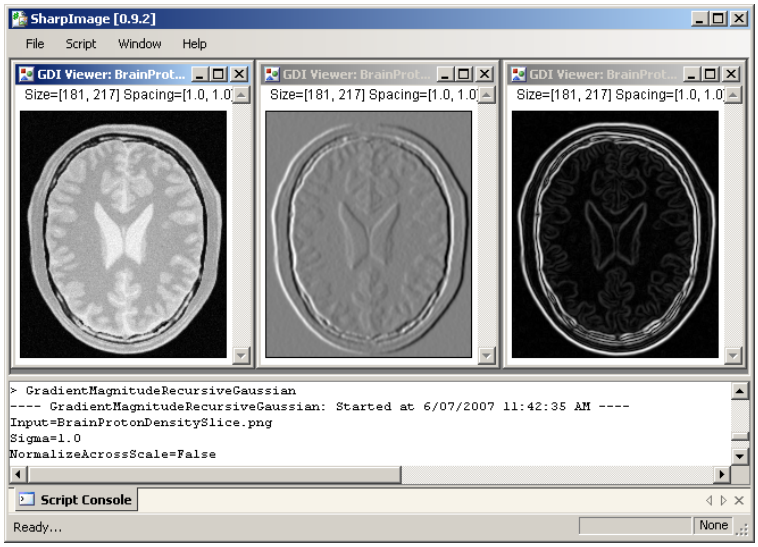


Figure 7: This figure depicts the results of applying the `GradientRecursiveGaussian` and `GradientMagnitudeRecursiveGaussian` scripts with `Sigma=1.0`.

4.8 Mathematical Morphology

Morphological filtering is a powerful technique which analyses images based on shape using a structuring element. ITK currently provides two streams of morphological filters: those for binary and those for greyscale images. SharpImage brings both of these together into single scripts using the `Operation` parameter: if `Operation="Binary"` then the binary variant is used, else if `Operation="Grayscale"` the greyscale variant is used (the default is always greyscale). The structuring element can be specified using the `itkFlatStructuringElement` static constructors: `Ball`, `Box`, `Annulus`. The default type is the `Ball` (circle in 2-D and sphere in 3-D).

```
> MorphologicalErode
> MorphologicalDilate
> MorphologicalOpen Operation="Binary"
> MorphologicalClose Operation="Binary"
> MorphologicalErode KernelRadius=itkSize(3,3)
> MorphologicalDilate Kernel=itkFlatStructuringElement.Ball(itkSize(2,2))
> MorphologicalOpen Kernel=itkFlatStructuringElement.Box(itkSize(2,2))
> MorphologicalClose Kernel=itkFlatStructuringElement.Annulus(itkSize(4,4),2)
```

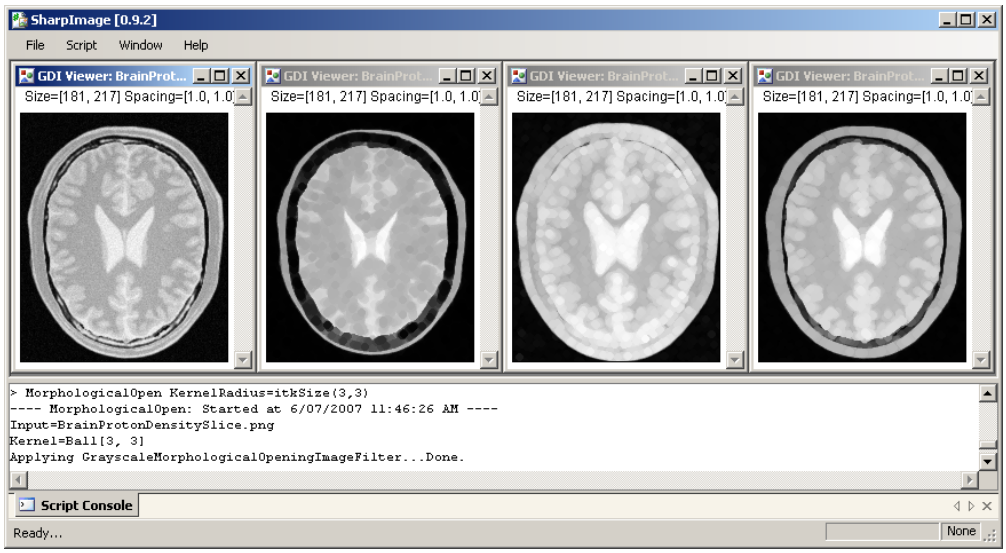


Figure 8: This figure depicts different morphological operations.

4.9 Noise

SharpImage has a number of scripts for both adding and suppressing noise. Gavin Baker has implemented additive Gaussian and impulse noise scripts for ITK (see his webpage) which are available using the `AdditiveGaussianNoise` and `ImpulseNoise` scripts:

```
> AdditiveGaussianNoise Mean=0.0 StdDev=20.0
> ImpulseNoise Probability=0.1
```

A number of naïve, neighbourhood, and edge-preserving denoising scripts are also provided:

```
> SmoothingRecursiveGaussian Sigma=1.0
> Mean Radius=2
> Median Radius=2
> GradientAnisotropicDiffusion Conductance=1.5 Iterations=4
> CurvatureAnisotropicDiffusion Conductance=1.5 Iterations=4
> Bilateral DomainSigma=[5.0,5.0] RangeSigma=5.0
> CurvatureFlow TimeStep=0.15 Iterations=4
```

4.10 Pixel Math

Per-pixel operations perform logical or mathematical operations on each pixel in an image. ITK implements nearly twenty different pixel math operations which have been drawn together in SharpImage using two simple scripts: `UnaryPixelMath` and `BinaryPixelMath`. As their names suggest, `UnaryPixelMath` facilitates operations on single images, while `BinaryPixelMath` operates on two images. The `UnaryPixelMath` script supports the following operations: `Abs`, `Exp`, `Log`, `Not`, `Negate`, `Sqrt`, or `Square`. The `BinaryPixelMath` script supports the following operations: `Add`, `Sub`, `Mult`, `Div`, `And`, `Or`, `Xor`, `Max`, `Min`, `Mask`, or `SquaredDiff`.

The following example masks an image using a binary image:

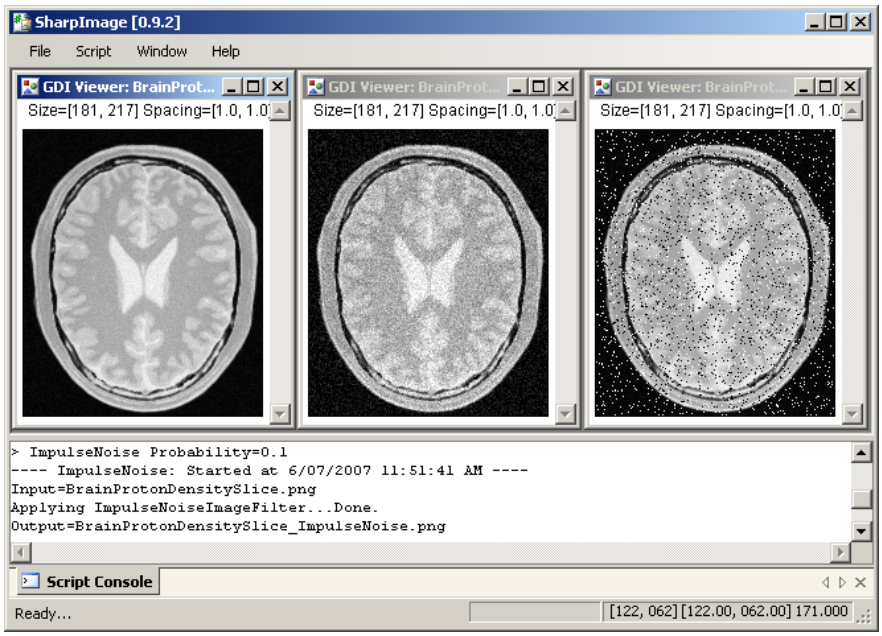


Figure 9: This figure depicts results from adding noise to an image.

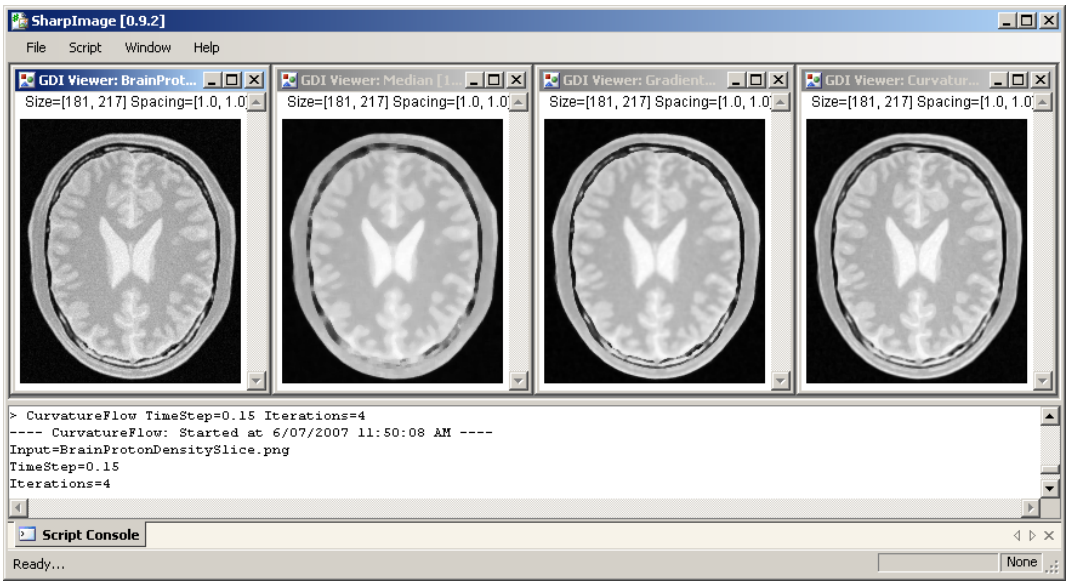
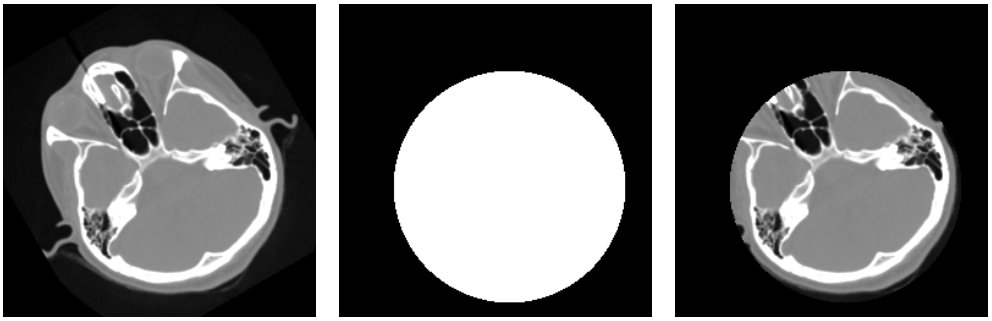


Figure 10: This figure depicts results from the `Median`, `GradientAnisotropicDiffusion`, and `CurvatureFlow` denoising filters.

```
> Open "C:/Temp/cthead1.png#F2"
> Open "C:/Temp/mask1.png#F2"
> BinaryPixelMath Operation="Mask" Input1="cthead" Input2="mask"
```



(a) Input1 (b) Input2 (c) Result

Figure 11: A demonstration of the `BinaryPixelMath Mask` operation.

4.11 Region growing

The `ConnectedThreshold` script performs region growing from a seed specified by clicking on the input image:

```
> ConnectedThreshold Lower=400 Upper=2000
```

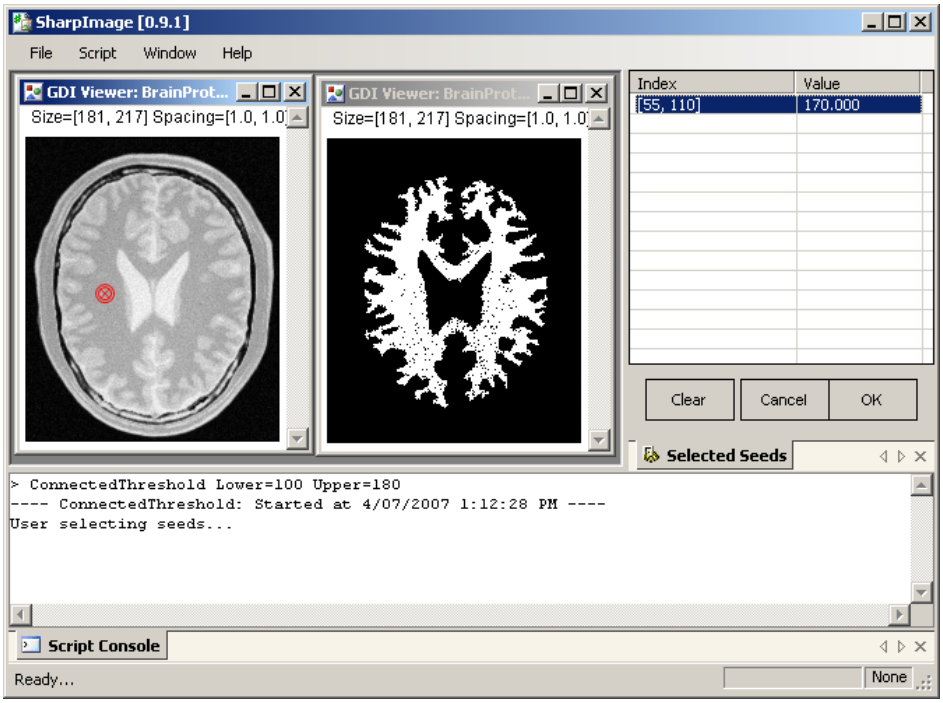


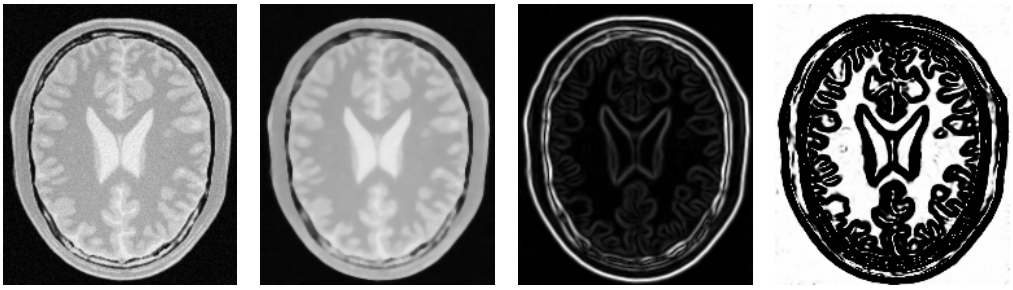
Figure 12: `ConnectedThreshold` region growing from a user specified seed point.

4.12 Active Contours

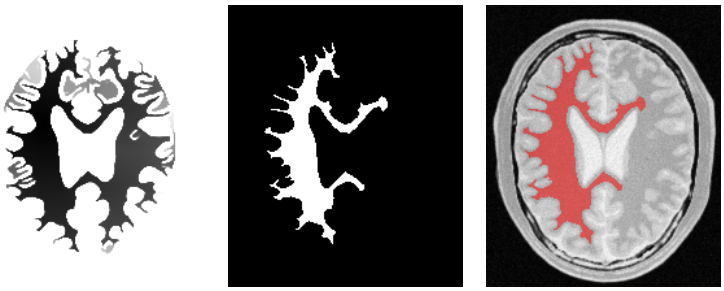
SharpImage provides a `FastMarching` script to compute the positive motion (expansion) of an active contour. The `FastMarchingImageFilter` example given in the ITK Software Guide [6,

Sec. 9.3.1] can be reproduced using six simple commands (plus the seed point specified using the mouse) (see Figure 13):

```
> Open "C:/Temp/BrainProtonDensitySlice.png#F2"
> CurvatureAnisotropicDiffusion TimeStep=0.125 Iterations=5 Conductance=9.0
> GradientMagnitudeRecursiveGaussian Sigma=1.0
> Sigmoid OutputMinimum=0.0 OutputMaximum=1.0 Alpha=-0.3 Beta=2.0
> FastMarching
Added trial point=[56, 92]
> BinaryThreshold Lower=0.0 Upper=100.0
> Select "Brain"
> AddLabel "Threshold"
```



(a) Input (b) Smooth (c) Gradient (d) Sigmoid



(e) Arrival (f) Threshold (g) Label

Figure 13: The `FastMarchingImageFilter` example from the ITK Software Guide [6, Sec. 9.3.1] can be reproduced using six simple commands.

An interactive script for computing an edge-based speed image for 2-D or 3-D images is also provided in `LevelSetSpeed` (see Figure 14):

```
> Open "C:/Temp/BrainProtonDensitySlice.png#F2"
> LevelSetSpeed OutputMinimum=-1.0 OutputMaximum=1.0
```

4.13 Volume Rendering

The `siVolumeRenderer` implements 3-D texture mapping using fragment shaders written in OpenGL Shading Language (GLSL). It expects at least one input image (a 'value' image), but also supports gradient / gradient magnitude images. To invoke the renderer using a given value and gradient magnitude image use the `VolumeRender` script in the following manner:

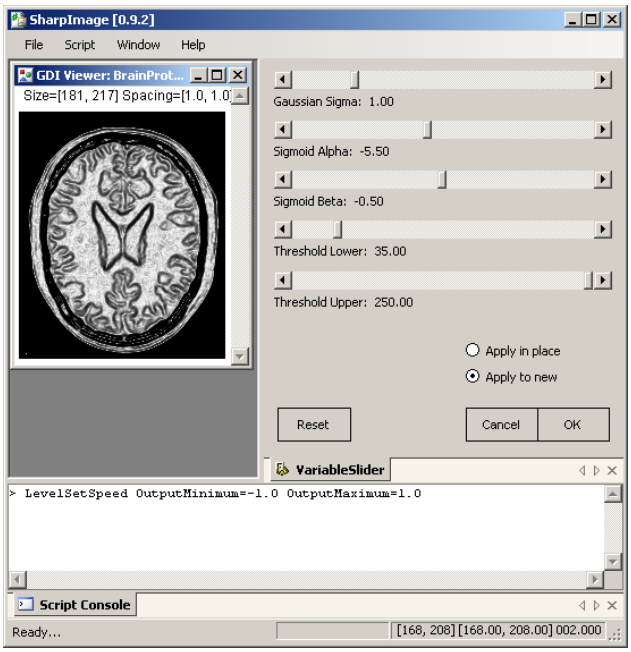


Figure 14: The `LevelSetSpeed` script in action.

```
> Open "C:/Temp/engine.mhd#UC3"
> GradientMagnitude
> VolumeRender Value="engine" Gradient="Magnitude"
```

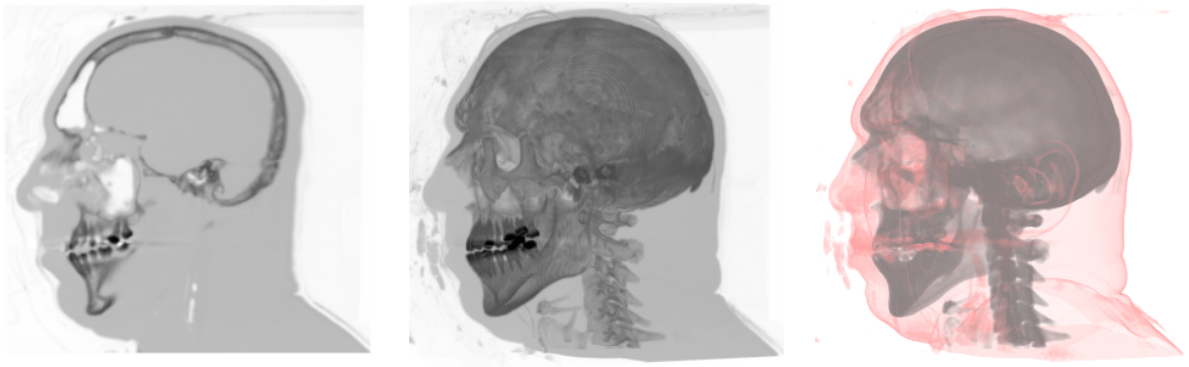
Once the script has finished, a new `siVolumeRenderer` is displayed along with a number of forms for specifying the renderer properties and transfer function. A histogram can be generated for the transfer function editor background using the `ValueEdgeHistogram` script:

```
> Open "C:/Temp/engine.mhd#F3"
> GradientMagnitude
> ValueEdgeHistogram Value="engine" Edge="Magnitude"
```

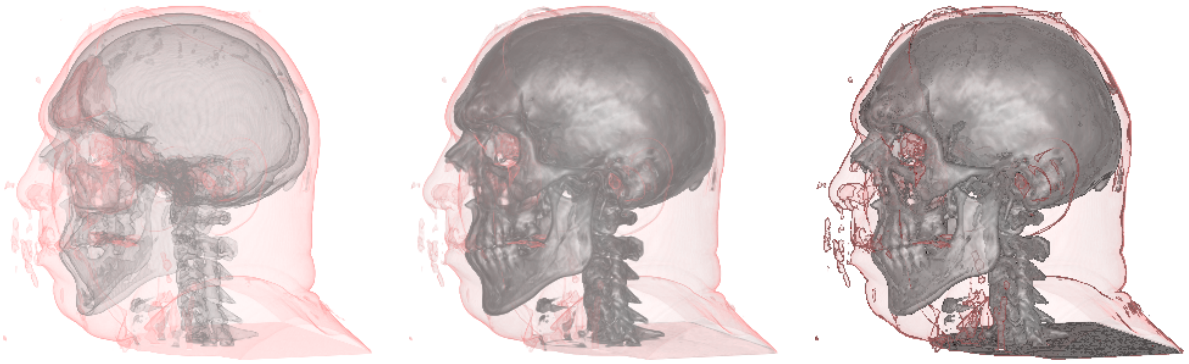
To change the fragment shader select the “Renderer Editor” tab, select the “FragmentProgramPath” property, click the “...” button and navigate the desired fragment shader (a vertex shader of the same name but different extension must also exist in the same directory). A number of existing fragment shaders are provided, but custom shaders can be easily added for prototyping illustration techniques.

The simplest shader is `shader-v-copy.frag` which copies the image value to the fragment and is useful for maximum intensity projections (see Figure 15(b)):

```
    float value = texture3D( sam3Tex1, gl_TexCoord[1] ).a;

    // Discard if no contribution
    // Set color
    gl_FragColor = vec4( value, value, value, 1.0 );
}
```



(a) Oblique Section (b) Minimum Intensity Projection (c) Value



(d) Value-Magnitude (e) Phong (f) Silhouette

Figure 15: Results from different fragment shaders and renderer settings.

Typically a 'transfer function' is employed to transfer opacity and colour information to each potential pixel ('fragment'). A simple transfer function is realised in `shader-v.frag` by indexing a 1-D lookup table (LUT) using only the image value (see Figure 15(c)):

```
    float value = texture3D( sam3Tex1, gl_TexCoord[1] ).a;

    // Interpolate transfer function
    vec3 pos3Tf = vec3( value, 0.0, 0.0 );
    vec4 val4Tf = texture3D( sam3Tex0, pos3Tf );

    // Discard if no contribution
    // Set color
    gl_FragColor = val4Tf;
}
```

A more complex transfer function is realised in `shader-vm-noadjust.frag` by indexing a 2-D LUT using image value and gradient magnitude [7] (see Figure 15(d)):

```
    float grad  = texture3D( sam3Tex2, gl_TexCoord[2] ).a;

    // Interpolate transfer function
```

```
    vec3 pos3Tf = vec3( value, 1.0-grad, 0.0 );
    vec4 val4Tf = texture3D( sam3Tex0, pos3Tf );

    // Discard if no contribution
    if (val4Tf.a <= 0.0) discard;
    gl_FragColor = val4Tf;
}
```

Unfortunately, as shown in Figure 16, this approach does not cater for changes in the sampling rate (the ratio of the distance between the interpolating proxy geometry). Weiler et al. [12] showed that the transparency of the fragment (α_0) must be adjusted according the sampling rate (r_s) as follows:

$$\alpha_k = 1 - [1 - \alpha_0]^{\frac{1}{r_s}} \tag{1}$$

This adjustment is implemented in `shader-vm.frag`:

```
{
    return 1.0 - pow( 1.0 - a,  1.0 / fSamplingRate );
}

void main()
```

The previous fragment shader implements what is sometimes called 'gradient magnitude opacity modulation'. However, true direct volume rendering typically implies the use of an illumination model, such as the Phong reflection model. The Phong model is a simplified *local* model consisting of three reflection terms (ambient, diffuse, and specular):

$$I = k_a i_a + \sum_{\text{lights}} \left[k_d i_d (N \cdot L) + k_s i_s (R \cdot V)^n \right] \tag{2}$$

where I is the resultant light intensity, k_x are constants, i_x are the light intensities of the respective components, N is the normal vector, L the vector to the light source, R the reflection vector, V the viewing vector, and n the specular exponent. For our implementation in `shader-vm-phong.frag`[1] (see Figure 15(e)) we compute the normal using an online central difference method, assume unity constants, and use a single light positioned along the viewing vector (ie. $V = L$):

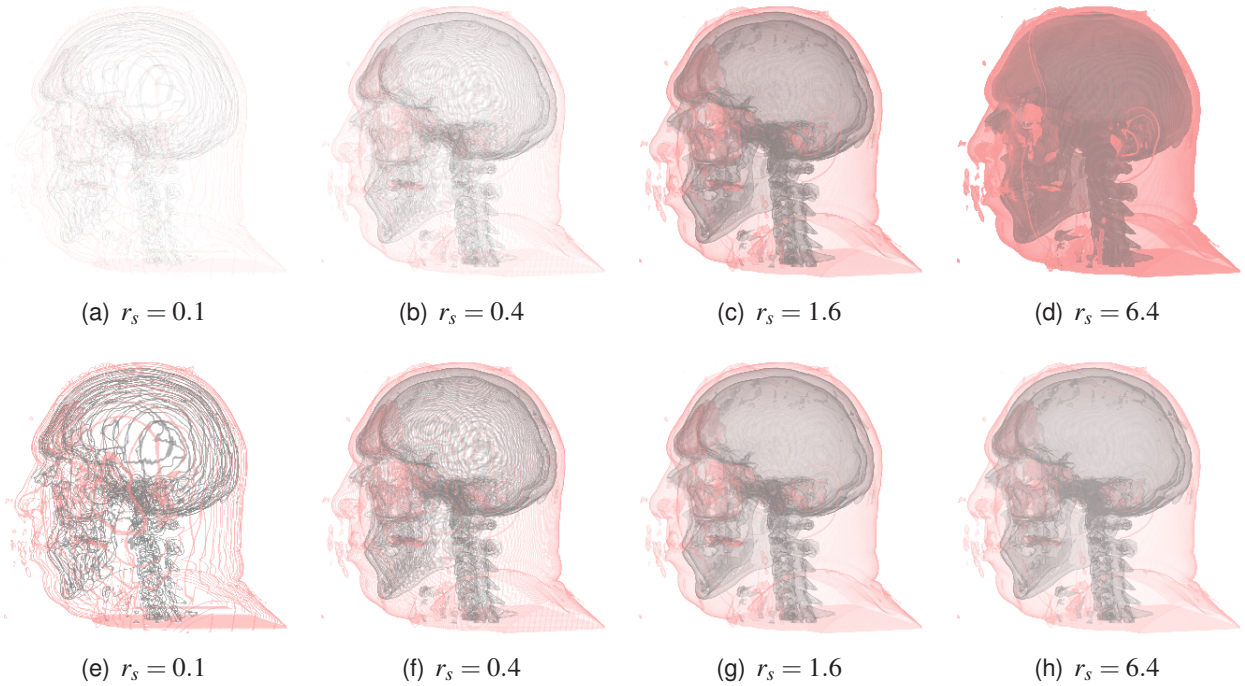
```
{
    vec4 gradient = vec4( 0.0 );
    for ( int i=0; i<3; i++ )
    {
        vec3 pos3L = vec3( gl_TexCoord[1] ); pos3L[i] -= fNormalOffset;
        vec3 pos3R = vec3( gl_TexCoord[1] ); pos3R[i] += fNormalOffset;
        float valL = texture3D( sam3Tex1, pos3L ).a;
        float valR = texture3D( sam3Tex1, pos3R ).a;
        gradient[i] = (valL - valR) / 2.0;
    }
```

[1] This shader is written for readability rather than speed or size. If you have an older graphics card we recommend using `shader-vm-phong-simple.frag`.

```
    return gradient;
}

// Apply the Phong lighting model
vec4 phong( vec4 vec4N, gl_LightSourceParameters light )
{
    // Compute light/view vector
    vec4 vec4L = vec4( 0.0, 0.0, -1.0, 0.0 );
    vec4L *= gl_ModelViewMatrix;
    vec4L = normalize( vec4(vec4L.x, vec4L.y, -vec4L.z, 0.0) );

    // Compute terms
    float fLdotN = dot( vec4L, vec4N );
    vec4 vec4R = normalize( (2.0 * fLdotN * vec4N) - vec4L );
    float fRdotL = abs( dot(vec4R, vec4L) );

    // Compute contributions
    vec4 vec4A = light.ambient;
    vec4 vec4D = light.diffuse  * abs( fLdotN );
    vec4 vec4S = light.specular * pow( fRdotL, light.spotExponent );

    // Add and return
    return (vec4A + vec4D + vec4S);
}

void main()
{
    // Interpolate images
    float value = vec4( texture3D(sam3Tex1, gl_TexCoord[1]) ).a;
    float grad  = vec4( texture3D(sam3Tex2, gl_TexCoord[2]) ).a;
    vec4 vec4N = normalize( vec4G );

    // Apply phong lighting
    gl_FragColor = val4Tf * phong( vec4N, gl_LightSource[0] );
}
```

Rheingans and Ebert [11] introduced a number of volume illustration techniques for enhancing boundaries, silhouettes, and regions. Some of these are realised in `shader-vm-enhance.frag` (see Figure 15(f)):

```
{
    vec4 vec4V = vec4( 0.0, 0.0, -1.0, 0.0 );
    vec4V *= gl_ModelViewMatrix;
    vec4V = normalize( vec4(vec4V.x, vec4V.y, -vec4V.z, 0.0) );
    return dot( vec4N, vec4V );
}

// Compute the silhouette enhancement
float enhanceSilhouette( vec4 vec4N )
{
    if (fFactor1 <= 0.0) return 0.0;
    return pow( 1.0 - abs( computeNdotV(vec4N) ), fFactor1 );
}
```



(a) $r_s = 0.1$ (b) $r_s = 0.4$ (c) $r_s = 1.6$ (d) $r_s = 6.4$

(e) $r_s = 0.1$ (f) $r_s = 0.4$ (g) $r_s = 1.6$ (h) $r_s = 6.4$

Figure 16: A comparison of value-magnitude volume renderings generated with different sampling rates. First row: no alpha adjustment. Second row: with alpha adjustment.

```
14
15 // Compute the boundary enhancement
16 float enhanceBoundary( float value )
17 {
18     if (fFactor1 <= 0.0) return 1.0;
19     return pow( value, fFactor1 );
20 }
21
22 void main()
```

4.14 Extending SharpImage

Although SharpImage has a decent collection of image processing scripts (thanks to ITK), its real strength lies in the ability to be easily extended. To add a new form for use with SharpImage follow these steps:

1. Open Visual Studio and select **File** > **New Project**.
2. Select **Windows Application**, enter the name and location, and click OK.
3. Right click on the project References, and select **Add Reference...**. Select the Browse tab, navigate to the SharpImage folder, select `SharpImage.exe`, and click OK.

4. In the Solution Explorer, right click on the **project name** and select **Properties**. Change the Output type from Windows Application to **Class Library**. Save the project and close the Properties window.

5. In the Solution Explorer, right click on `Program.cs` and select **Delete**.

6. In the Solution Explorer, right click Form1.cs and select **Rename**. Enter the desired name, such as siFormCustom1.cs.

7. In the Solution Explorer, right click on the form and select **View Code**. Change the name of the form in the class declaration and make it extend `SharpImage.Forms.siFormTool`. Also change the default constructor name.

8. In the Solution Explorer, right click on the form and select **View Designer**. Edit the form as desired. For this example we changed the form text in the Properties explorer and added a check box.

9. In the Solution Explorer, right click on the form, select View Code, and **add code** as desired. For this example we added a property named `RunFilter` which returns the checkbox Checked property.

10. Select the build type and **compile** the library.

11. **Create a Scripting/Custom folder** in the SharpImage installation directory. Copy the class library to this folder.

12. **Create a script** in this folder which uses the form (see below).

13. **Invoke the script** from SharpImage.

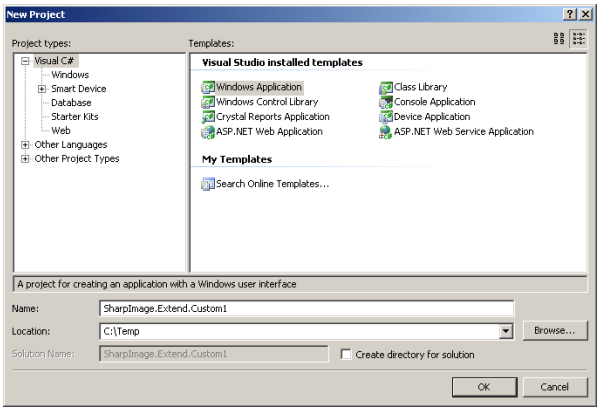
```
#=========================================================================
# Module:    Custom1.py
#=========================================================================

# Import the base script class
import ImageToImageScript
from ImageToImageScript import *

# Add reference and import required libraries
clr.AddReference("ManagedITK.Image.Cast")
clr.AddReference("SharpImage.Extension.Custom1")
from itk import *
from SharpImage.Extension.Custom1 import *

class Custom1Script(ImageToImageScriptObject):
    # ------------------------------------------------------------------------
    Name = "Custom1"
    Help = """Insight Journal custom example.
              A form is displayed asking if the filter should be run."""
    Parameters = """None"""
    Form = None
    RunFilter = False
    # ------------------------------------------------------------------------

    def Run(self):
        """ Run the script. """
        self.Initialise()
```

```
28
29      def Initialise(self):
30          """ Initialise the environment for running this script. """
31
32          # ...
33
34          # Show the form
35          self.Form = siFormCustom1( )
36          self.Form.Continue += self.Continue
37          self.Form.Cancel += self.Cancel
38          self.ParentApplication.AddTool( self.Form )
39
40      def ThreadedDoWork(self):
41          """ Perform the main functions of the script on a background thread. """
42          try:
43              # Setup
44              self.ParentApplication.SetApplicationAsWorking()
45
46              # Check if we are running the filter
47              if (self.Form.RunFilter):
48                  # Run the filter
49                  # ...
50              else:
51                  # Don't run the filter
52                  #...
53
54          except Exception, ex:
55              self.HandleException( ex )
56              self.FinishedWork( False )
57              self.Finalise( )
```

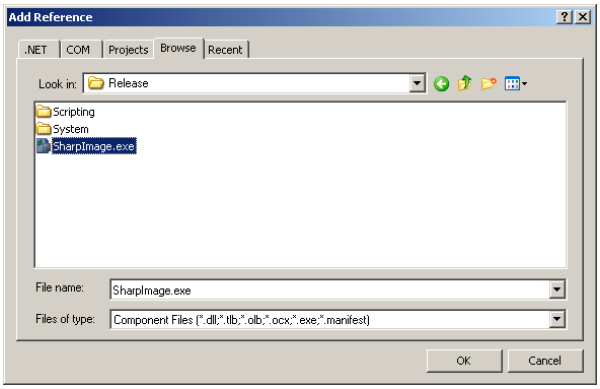
5 Conclusion

This paper described an image processing prototyping environment based on ITK. It currently supports two renderers (a slice renderer and a volume renderer) and allows various scripts to be called from a command console. New scripts and forms can be added to the environment at run-time very easily. We have found the environment useful for a range of image processing tasks including prototyping volume illustrations. For suggestions or bugs, feel free to contact us[2].

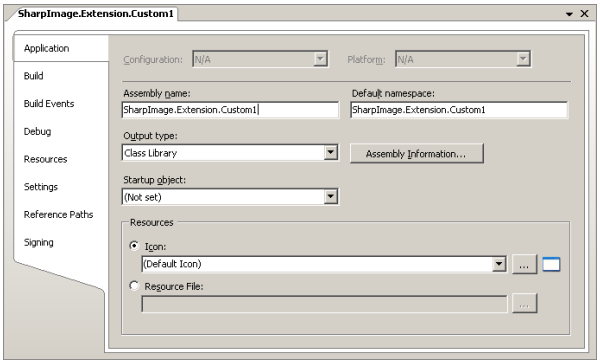
[2]Corresponding author: Dan Mueller: d.mueller@qut.edu.au or dan.muel@gmail.com.



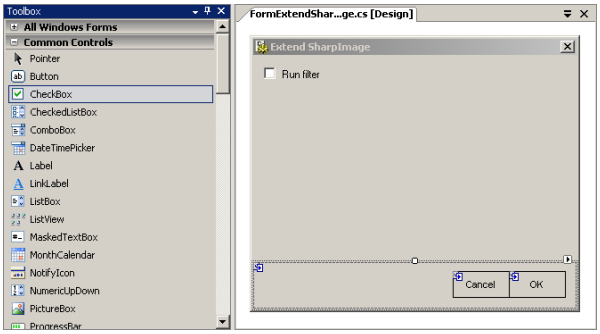
(a) Create a new project



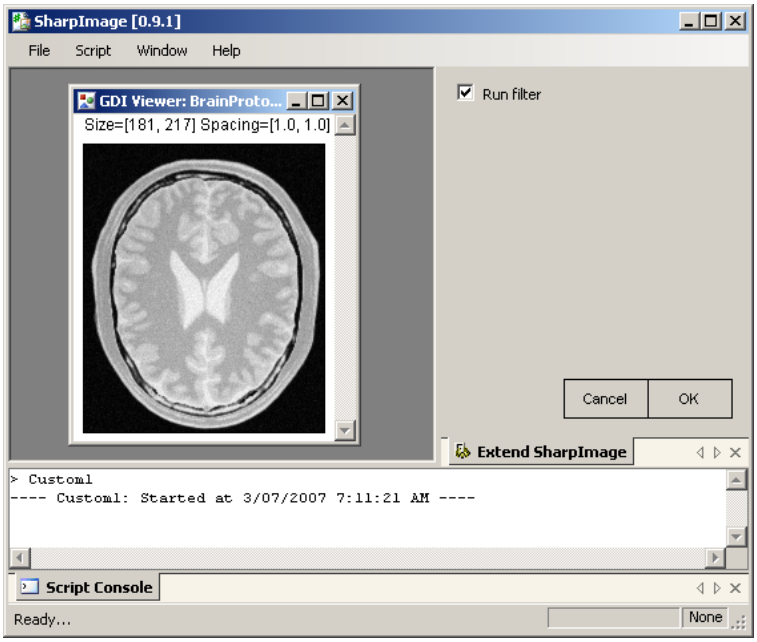
(b) Add reference



(c) Edit project properties



(d) Design the form



(e) Invoke the script

Figure 17: Steps for extending SharpImage.

References

[1] MeVisLab. Technical report, MeVis, 2007, Available online: http://www.mevislab.de/index.php?id=mevislabmain. 1

[2] SCIRun. Technical report, Scientific Computing and Imaging Institute, University of Utah, 2007, Available online: http://software.sci.utah.edu/scirun.html. 1

[3] Slicer 3.0. Technical report, NA-MIC, 2007, Available online: http://www.na-mic.org/Wiki/index.php/Slicer3. 1

[4] VolView 2.0 Users Guide. Technical report, Kitware, Inc, 2007, Available online: http://www.volview.com/VolView/VolView20Help.pdf. 1

[5] V. Chu and G. Hamarneh. MATITK: Extending MATLAB with ITK. *The Insight Journal*, December, 2005. 1

[6] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. The ITK Software Guide: The Insight Segmentation and Registration Toolkit. Technical report, Kitware, Inc, 2007. 4.12, 13

[7] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002. 2.3, 4.13

[8] G. Lehmann, Z. Pincus, and B. Regrain. WrapITK: Enhanced languages support for the Insight Toolkit. *The Insight Journal*, June, 2006. 1

[9] D. Mueller. ManagedITK: .NET wrappers for ITK. *The Insight Journal*, June, 2007. 1

[10] C. Rezk-Salama. *Volume rendering techniques for general purpose graphics hardware*. PhD dissertation, University Erlangen-Nuremberg, 2001. 2.3

[11] P. Rheingans and D. Ebert. Volume illustration: nonphotorealistic rendering of volume models. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):253–264, 2001. 4.13

[12] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering via 3D textures. In *Volume Visualization and Graphics*, pages 147–152. IEEE, 2000. 4.13

[13] 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