

Introdução MeVisLab

Luiz Alberto Bordignon Mestrando Engenharia Elétrica
Prof. Dr. Lucas Ferrari de Oliveira

Departamento de Engenharia Elétrica
Universidade Federal do Paraná



Introdução

- MeVisLab é uma plataforma de prototipagem e desenvolvimento rápido para processamento de imagens médicas e visualização.
- Oferece maneiras fáceis de desenvolver novos algoritmos ou mudança/melhoria dos já existentes.
- Integração rápida e fácil em ambientes clínicos.
- MeVisLab inclui módulos avançados de processamento de imagens.
- Com base em MeVisLab, vários protótipos clínicos têm sido desenvolvidos, incluindo assistentes de software para neuroimagem, planejamento de cirurgias, e análise de vasos.

Utilização e requisitos

- O conhecimento prévio necessário depende do uso MeVisLab:
 - Para a criação de uma rede simples, nenhum conhecimento de programação é necessária.
 - Para criação de uma macro, é necessário conhecimento básico de Python ou JavaScript.
 - Para o desenvolvimento de módulos, conhecimento básico de C++ é necessário.
 - Para opções de visualização, é necessário algum conhecimento de processamento de imagens e computação gráfica.

Desenvolvimento

- Em MeVisLab, o desenvolvimento pode ser feito em três níveis:
 - **Nível Visual:** Programação "*plug and play*": módulos de processamento de imagem, visualização e interação podem ser combinados para redes de processamento de imagens complexas utilizando uma abordagem de programação gráfica.
 - **Nível de script:** Pode ser criado script em Python ou JavaScript para implementar funcionalidades a rede.
 - **Nível C++:** módulos de programação: Novos algoritmos pode ser facilmente integrados utilizando a linguagem C++.

Desenvolvimento em MeVisLab

- O fluxograma de trabalho para desenvolvimento de um aplicativo em MeVisLab ficaria da seguinte forma:
 - Conectar os módulos existentes para formar uma rede.
 - Desenvolver novos módulos, se necessário.
 - Construir uma interface para o usuário (GUI).
 - Criar uma macro para reutilizar uma rotina complexa.
 - Usar scripts para controlar as redes, GUIs, e macros.
 - Construir um instalador (apenas com uma licença especial).

Manipulação e processamento

- **Manipulação de imagens**

- ***ImageLoad*** : abre um arquivo de imagem armazenado em um dos seguintes formatos: DICOM, TIFF, DICOM/TIFF, RAW, LUMISYS, PNM, Analyze, PNG, JPEG.
- ***LocalImage*** : funciona como ImageLoad, mas carrega as imagens em relação à uma rede ou a caminho padrão do MeVisLab.
- ***ImageSave*** : salva uma imagem usando um dos seguintes formatos de arquivo: DICOM, TIFF, DICOM/TIFF, RAW, LUMISYS, PNM, Analyze, PNG, JPEG.

Propriedades da Imagem

- ***Info***: mostra informações sobre a imagem, como tamanho, tamanho da página, etc.
- ***MinMaxScan*** : verifica a entrada e atualiza os valores mínimos e máximos da imagem de saída.
- ***ImageStatistics*** : calcula algumas estatísticas dos *voxels* da imagem de entrada.

Processamento de imagem básico

- ***Subimage***: extrai uma sub-imagem a partir de uma imagem de entrada com base em qualquer, *voxel* início/fim. Também pode ser usado para criar uma região maior do que a imagem de entrada.
- **Arithmetic1**: executa operações aritméticas em uma imagem.
- **Arithmetic2**: executa operações aritméticas em duas imagens.
- **Mask**: aplica uma mascara sobre a imagem.
- **Testpattern**: gera uma imagem de teste.
- **AddNoise**: produz ruído em uma imagem, por exemplo ruído uniforme, ruído Gaussian, etc.

Filtros

- **Convolution:** filtros baseados no kernel padrão, como média, Gauss, Laplace ou Sobel.
- **ExtendedConvolution:** semelhante ao *Convolution* mas com tamanhos de kernel mais flexíveis.
- **Morphology** : implementa filtros de dilatação e erosão. Utilizando intervalos de limiar, o filtro pode ser aplicado seletivamente às regiões da imagem.

Segmentação

- **Threshold**: transforma a imagem de entrada em uma imagem binária, em que os voxels abaixo do limiar são ajustados para o valor mínimo da imagem, e voxels iguais ou superiores ao limiar são ajustados para o valor máximo da imagem.
- **IntervalThreshold** : filtra apenas os valores da imagem que se encontram em um determinado intervalo. Voxels fora deste intervalo pode ser ajustado para zero ou para um valor de preenchimento definido pelo usuário. Isto pode ser útil para a segmentação de objetos que deverão ter valores de cinza em um intervalo definido.

Segmentação

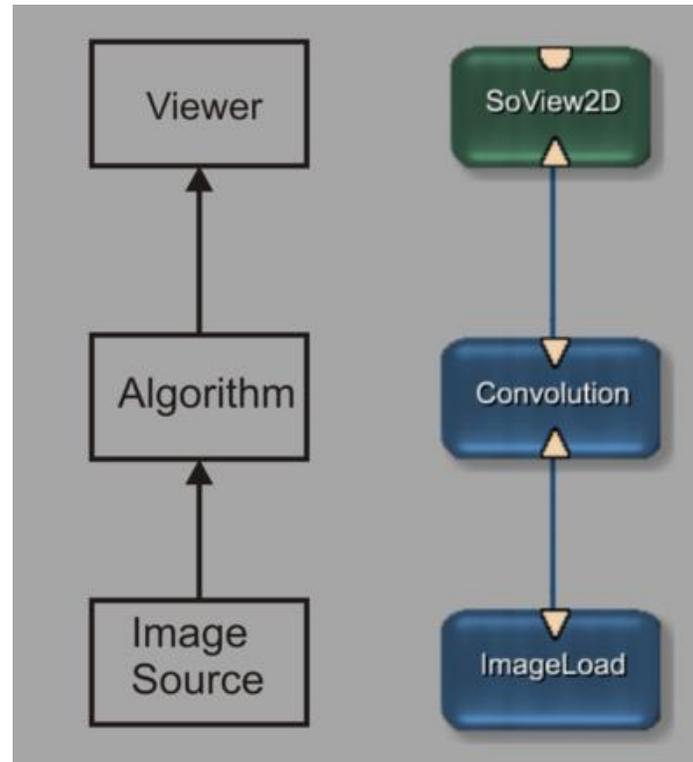
- **RegionGrowing** : fornece um limiar simples de algoritmo de crescimento de regiões.
- **RegionGrowingMacro** : amplia as opções de *RegionGrowing* adicionando um *View2D* e um editor.

Visualização 2D

- **View2D**: fornece um visualizador para as imagem em 3D como e fatias em 2D. É possível alterar a imagem ao arrastar o mouse com o botão direito pressionado.
- **SoView2D**: exibe uma fatia de uma imagem 2D.
- **View2DExtensions**: encapsula um conjunto de extensões de *viewers* que são comumente usados em conjunto com um visualizador 2D, (navegação, zoom, etc), nível/ajuste de janela e anotações de desenho.

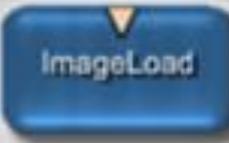
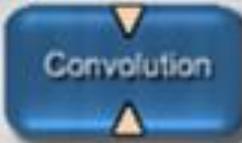
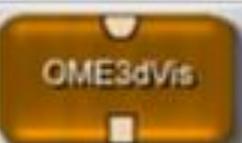
Desenvolvimento em MeVisLab

- Em MeVisLab, os algoritmos são visualizados como uma rede de módulos (gráficos).



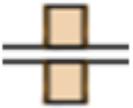
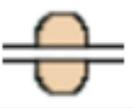
Módulos MeVisLab

- O MeVisLab trabalha com o conceito de módulos, que são representações gráficas com funções específicas.
- Os três tipos de módulos básicos (ML, inventor e macro) são distinguidos por suas cores:

Tipo	Exemplos
ML Module (azul)	 
Módulos Open Inventor (verde)	
Módulo Macro (marrom)	

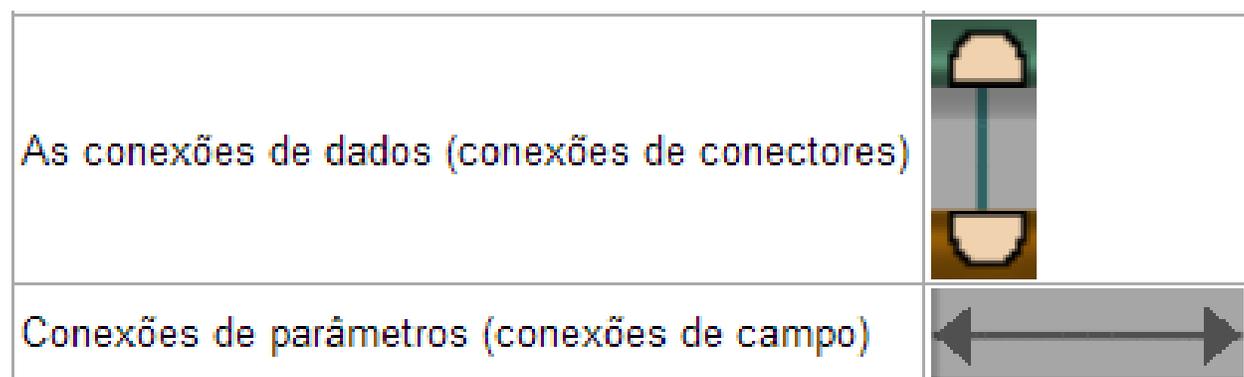
Módulos MeVisLab

- A maioria dos módulos possuem conectores. Estes representam as entradas (em baixo) e saídas (superior) dos módulos.
- Em MeVisLab, são definidos três tipos de conectores.

	quadrado	Objetos de base: os ponteiros para estruturas de dados
	triângulo	Imagens ML
	semicírculo	Cena Inventor

Módulos MeVisLab

- Ao ligar esses conectores e, portanto, estabelecer uma conexão de dados, dados de imagem ou do Open Inventor são transportados de um módulo para outros.
- Um módulo pode ser ligado a qualquer outro módulo desde que os mesmos sejam compatíveis.

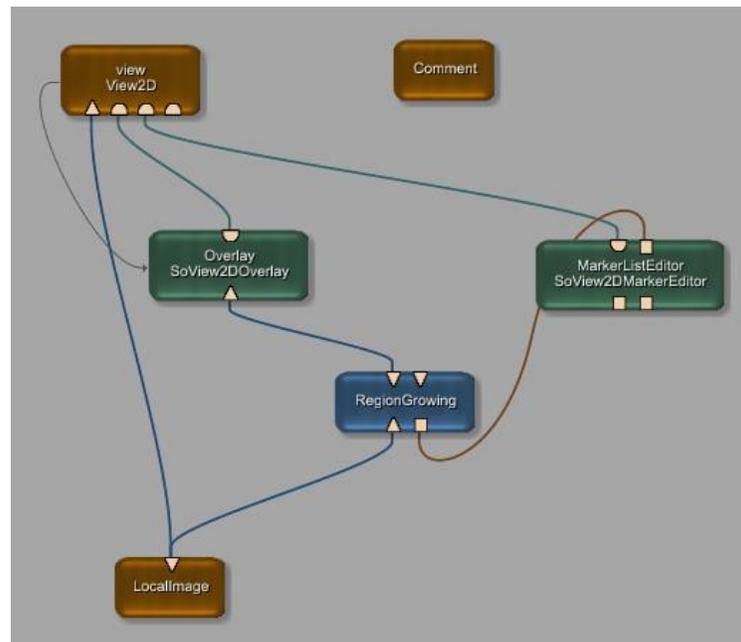


Campos

- Os campos definem a interface de um módulo.
 - Eles vêm em dois tipos:
 - 1 - In/out - ligados por conexões de dados
 - Imagens
 - Objetos
 - 2 - Campos de parâmetros - ligados por conexões de parâmetros
 - Números, strings, booleanos
 - Vetores
 - Triggers

Redes

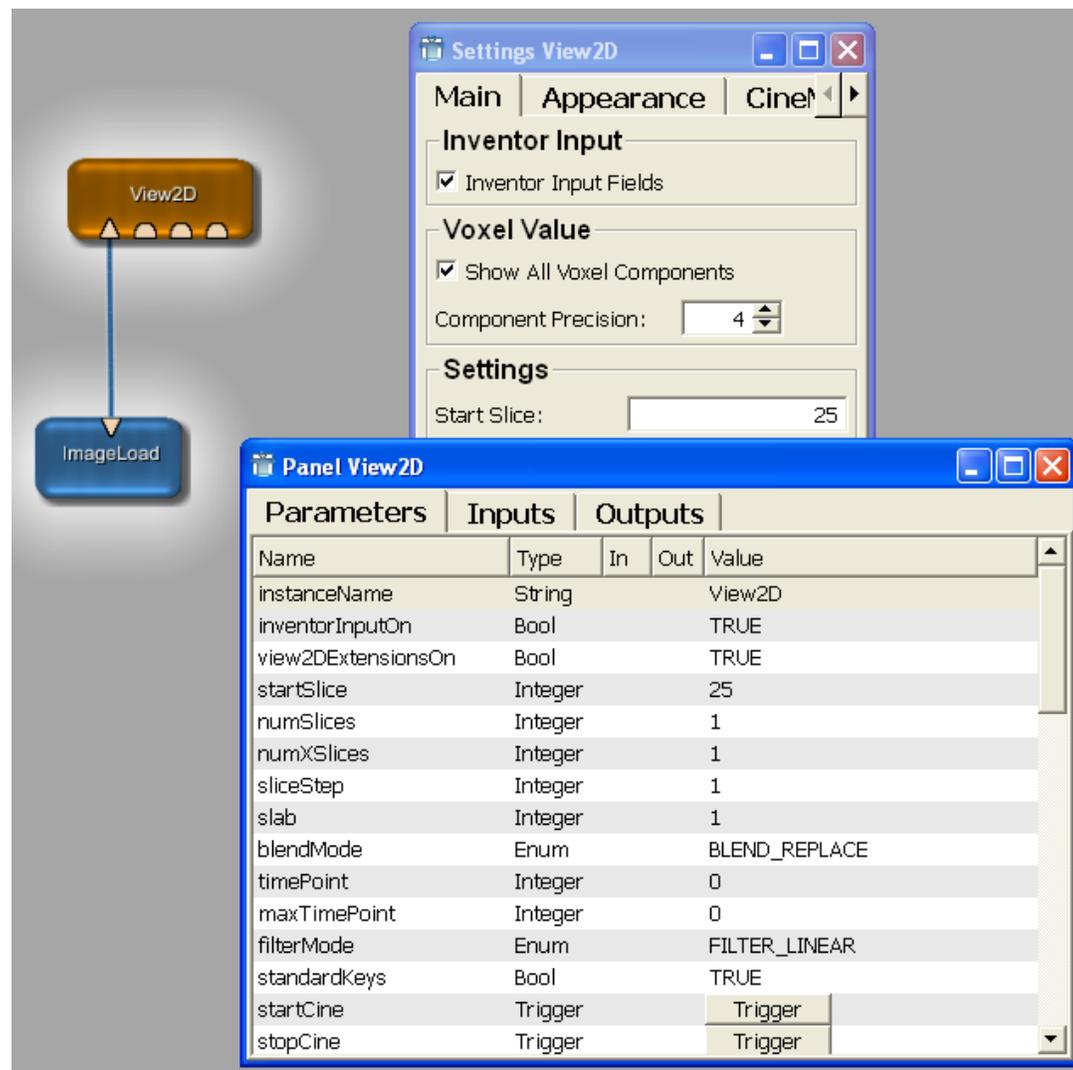
- As redes são conexões entre dois módulos com os quais você pode executar tarefas de processamento.
- As redes são editadas e salvas como .mlab, formato do MeVisLab.
- Na figura abaixo exemplo de rede e suas conexões.



Visão geral dos arquivos importantes

Tipo de Arquivo	Conteúdo
.mlab	Inclui todas as informações sobre seus módulos, conexões e configurações.
.def	Arquivo de módulo, necessário para um módulo ser adicionado à base de dados comum do MeVisLab. Também pode incluir todas as partes do script MDL
.script	Arquivo de script de MDL, normalmente inclui a definição de interface do usuário (GUI).
.mhelp	Arquivo com as descrições de todos os campos e a utilização de um módulo.
.py	Arquivo Python, utilizada para criação de scripts em módulos de macro.
.js	Arquivo JavaScript, utilizado para execução de scripts em módulos de macro.
.dcm	Parte do arquivo DCM dos arquivos DICOM importados.
.tiff	Parte do arquivo TIFF dos arquivos DICOM importados.
.mlimage	Imagem 6D salvo com todas as etiquetas DICOM.

Interfaces de controle de usuários



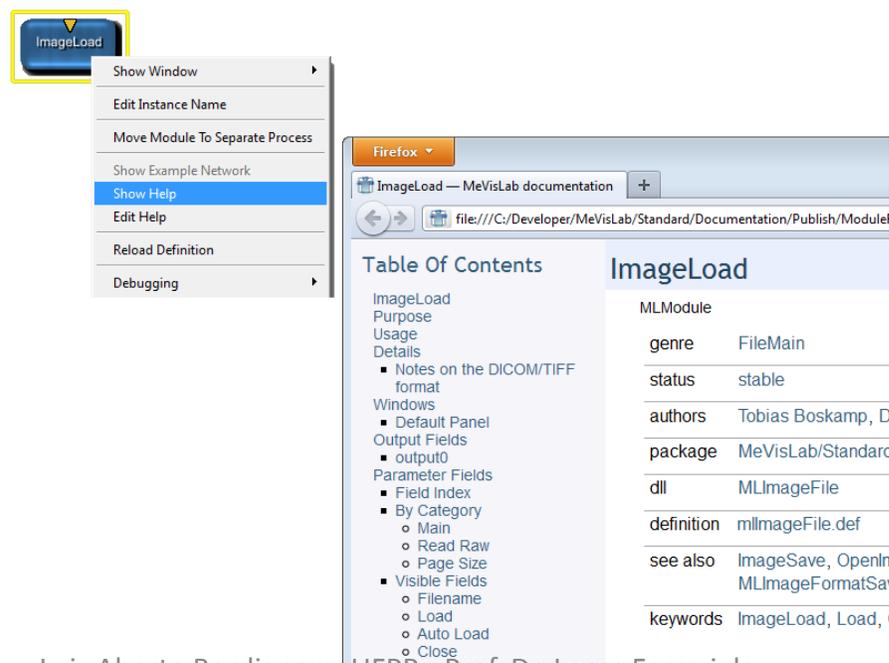
Scripting

- MeVisLab oferece interfaces de script. Os scripts podem ser implementados em Python ou JavaScript .
- **Python modules**

```
Commands {  
    source = $(LOCAL)/YourModuleName.py  
    ...  
}
```

Ajuda sobre os módulos

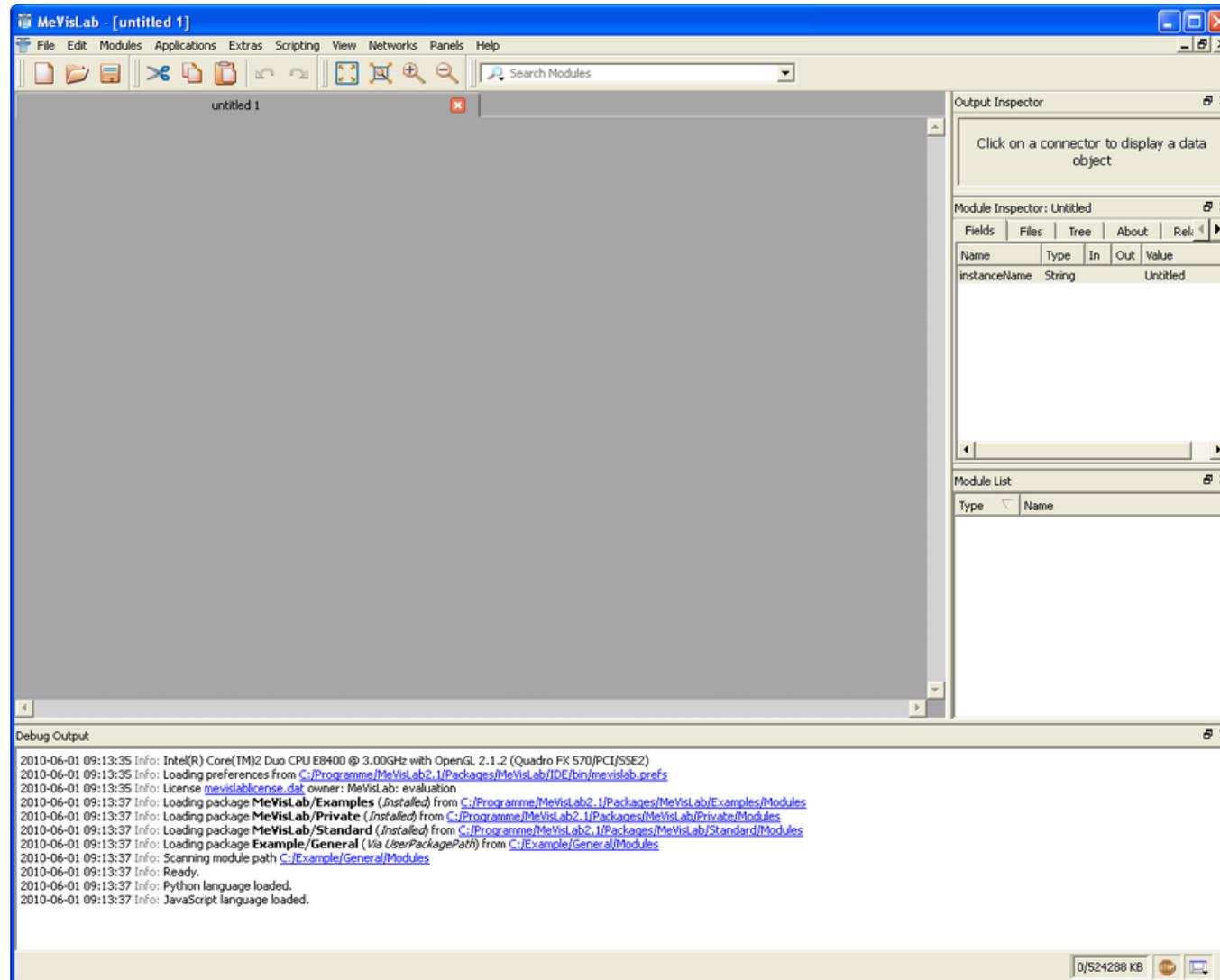
- Quando você digita o nome do módulo na busca rápida, as informações sobre o módulo é exibida.
- Selecione **Show Help** para abrir a ajuda HTML do módulo em seu navegador padrão.



Ajuda sobre os módulos

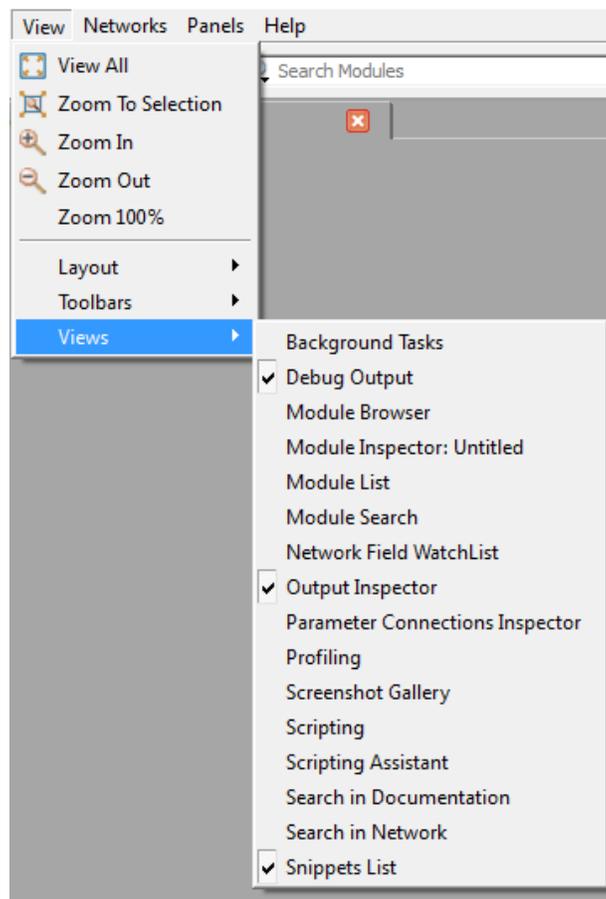
- Para a maioria dos módulos fornecidos existe um exemplo de rede que pode ser acessado com o botão direito do mouse sobre o módulo selecionado -> *Show Example Network*.

Interface do Usuário



Interface do Usuário

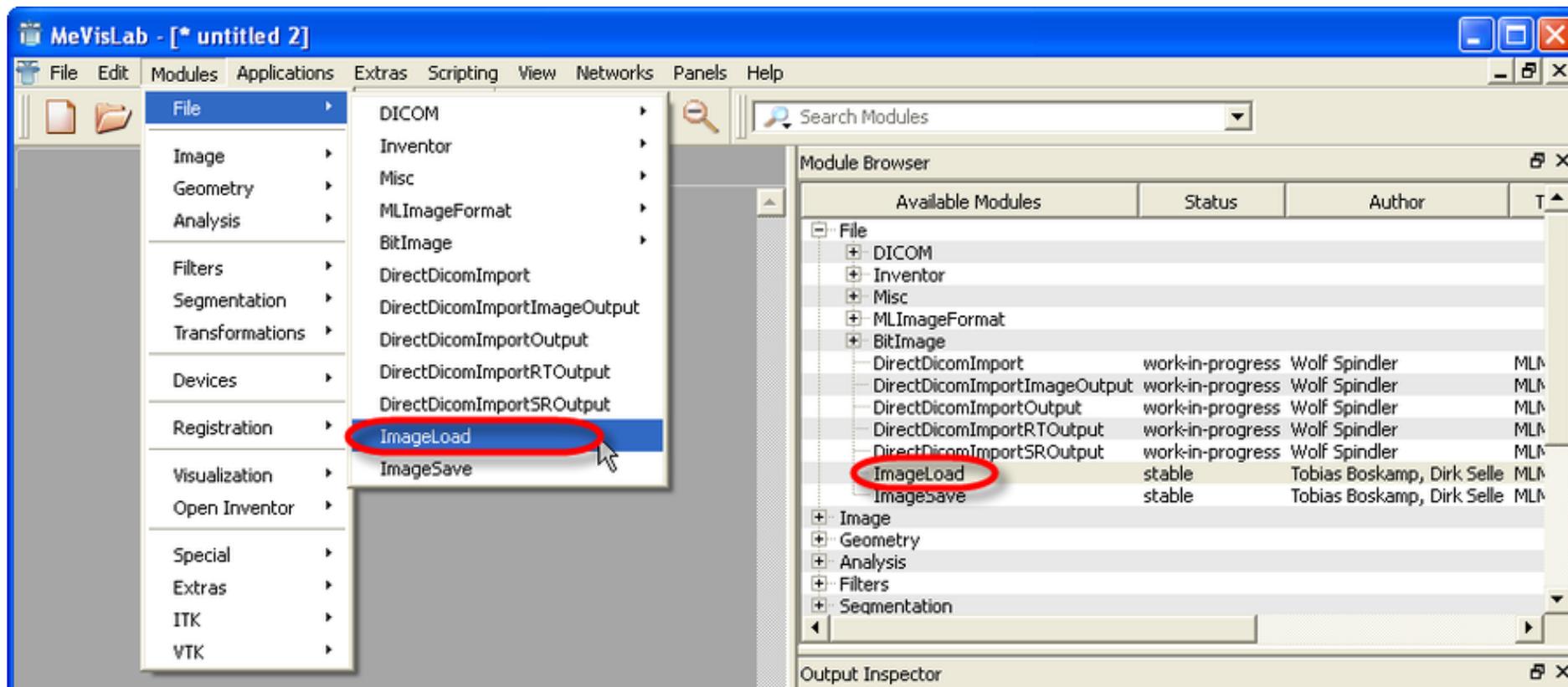
- O *layout* principal pode ser configurado pelo usuário na aba **View** -> **Views**.



Adicionando módulos

- Há várias maneiras de adicionar um módulo, por exemplo:
 - através da barra de menu, *Modules*.
 - através da barra de menu, *Quick Search*.
 - através da barra de pesquisa *Module Search*.
 - através *Module Browser*.
 - Copiar e colar a partir de outra rede.
 - por scripts.

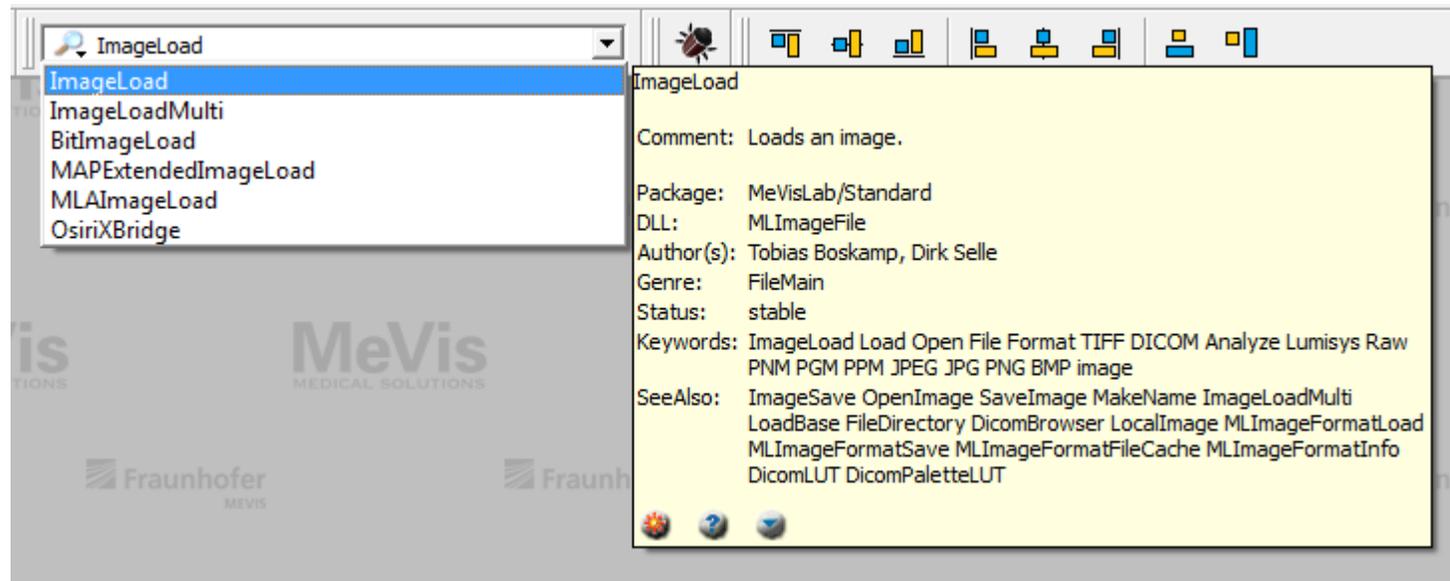
Adicionando módulos



- A maneira mais rápida de adicionar módulos a uma rede é utilizando a busca rápida na barra de menu.

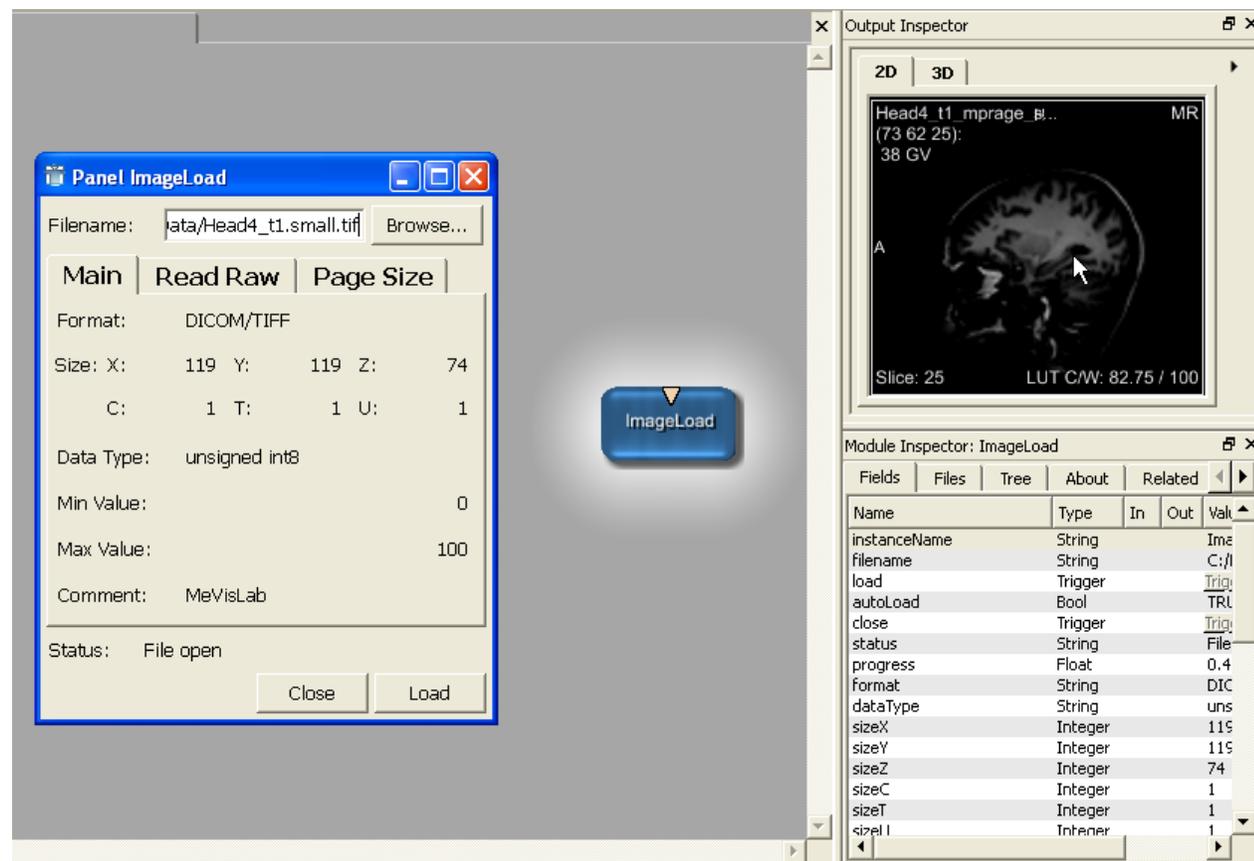
Opções de busca

- Por exemplo para procurar um módulo para carregar uma imagem, você pode digitar "load" ou "image". Como mostrado na imagem abaixo, ao lado da busca é mostrado suas informações, o que lhe permite decidir se este é o módulo correto.



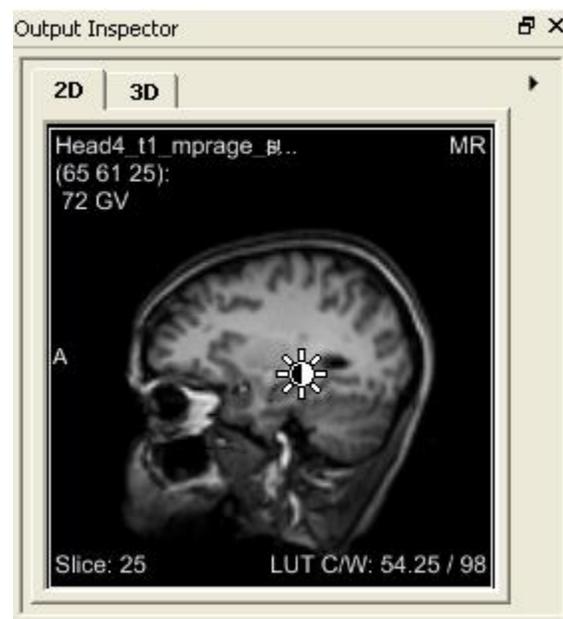
Usando o módulo *ImageLoad*

- Para carregar uma imagem basta clicar duas vezes sobre o módulo *ImageLoad*.

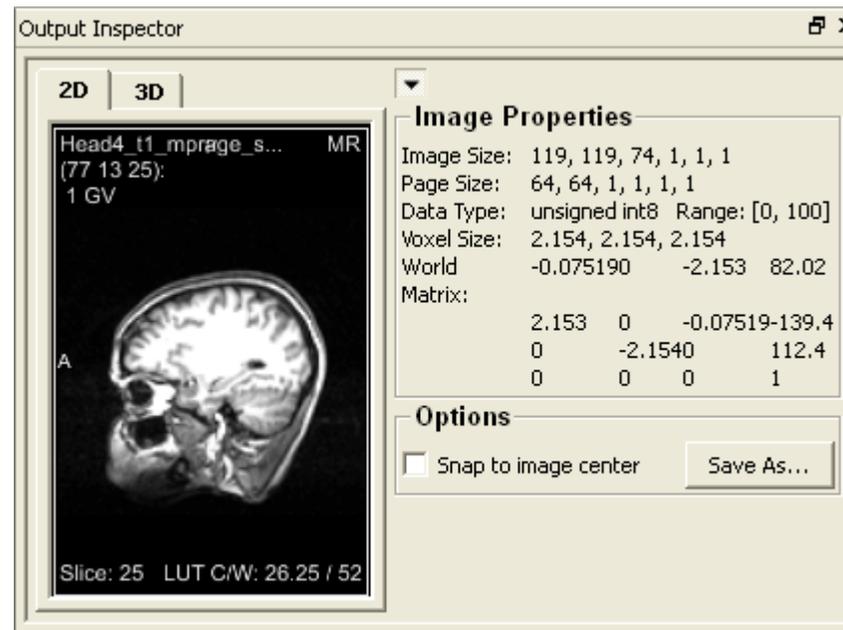
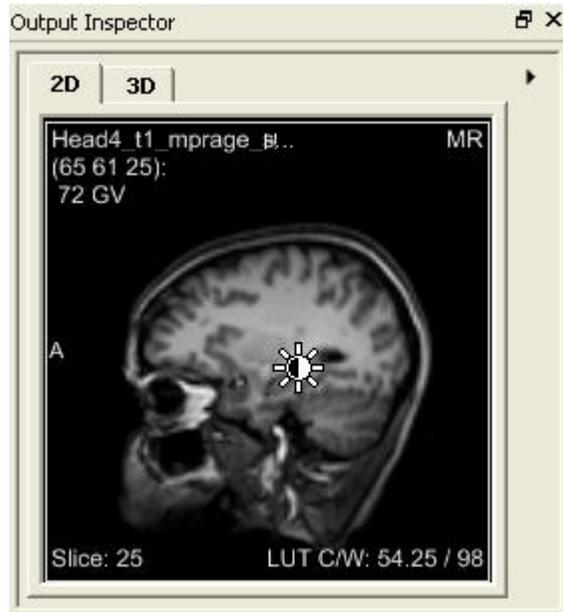


Ajustando a imagem no *ImageLoad*

- Usar o *scroll* do mouse



Output Inspector: propriedades de imagem



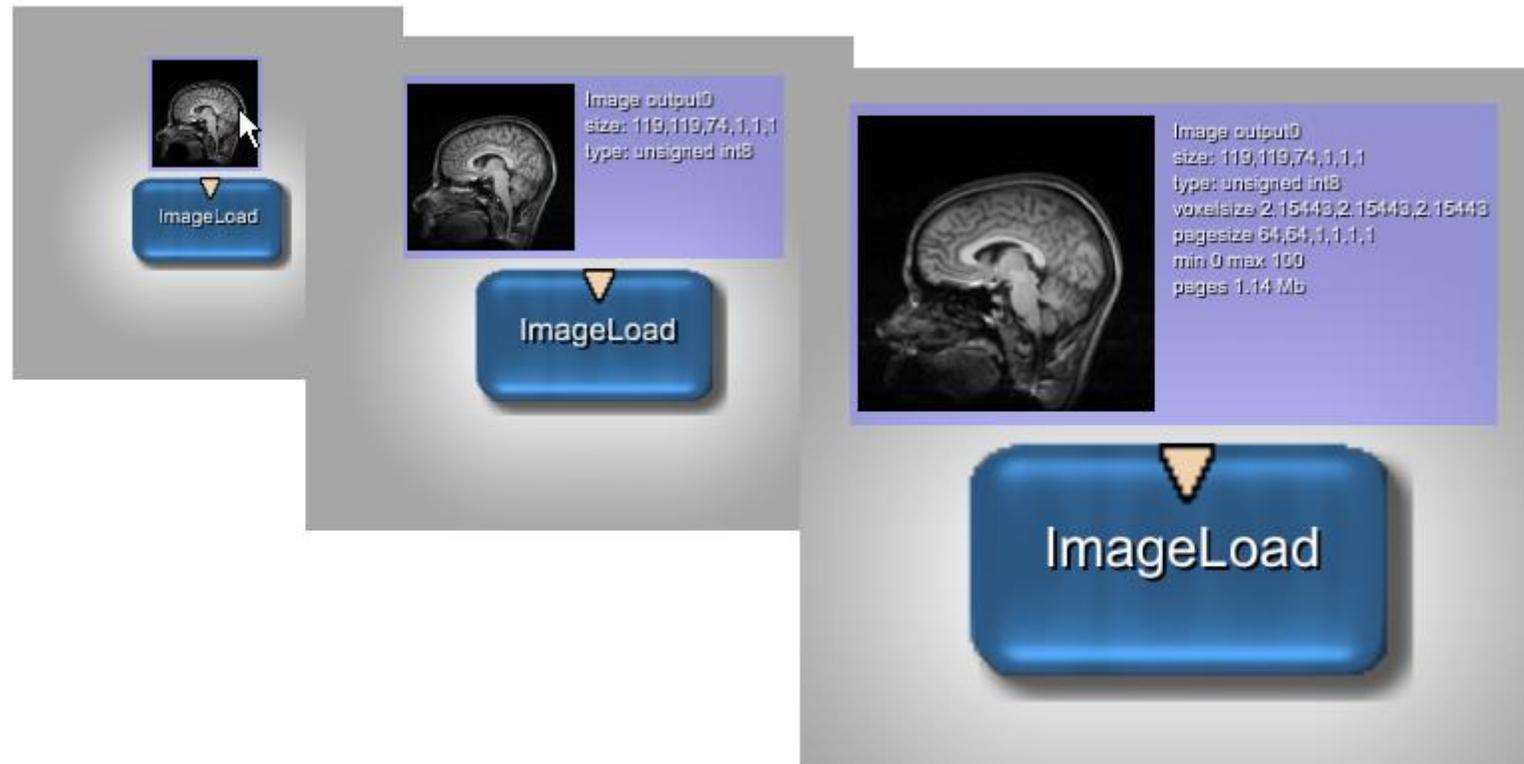
Visualização 3D

- É possível mover a imagem na visualização 3D. A orientação pode ser vista no pequeno cubo no canto inferior direito do visualizador.



Zoom in/out

- Na área de trabalho do MeVisLab é possível ajustar o *zoom* dos módulos através do *scroll* do mouse.



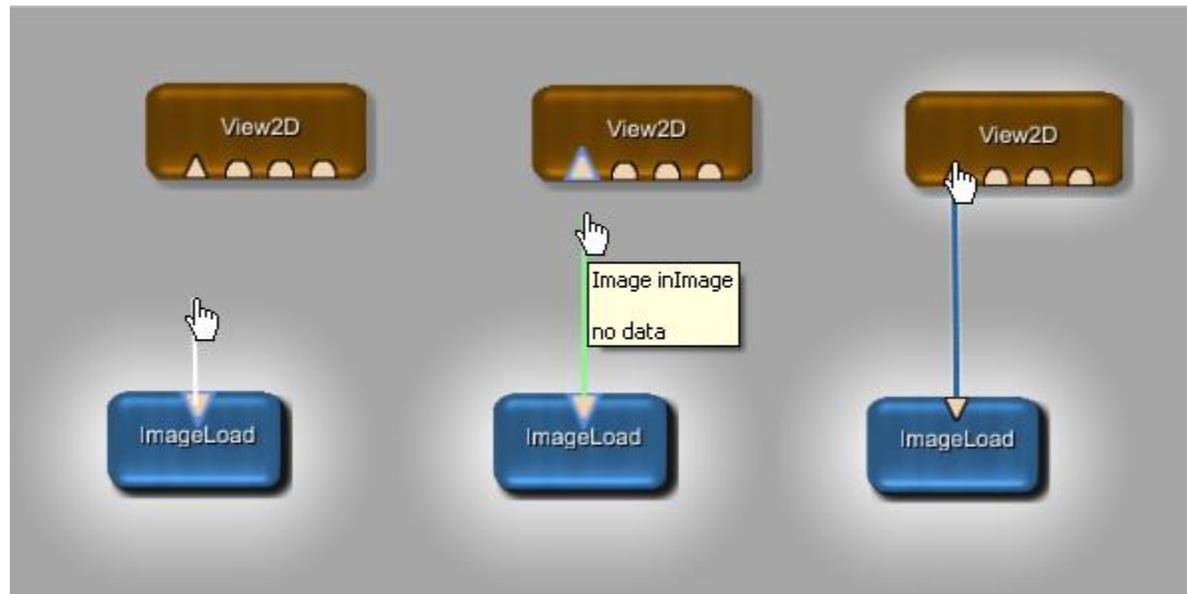
Adicionando o módulo View2D

- Os módulos view2D e view3D são frequentemente utilizados para visualização dos resultados do processamento das imagens.

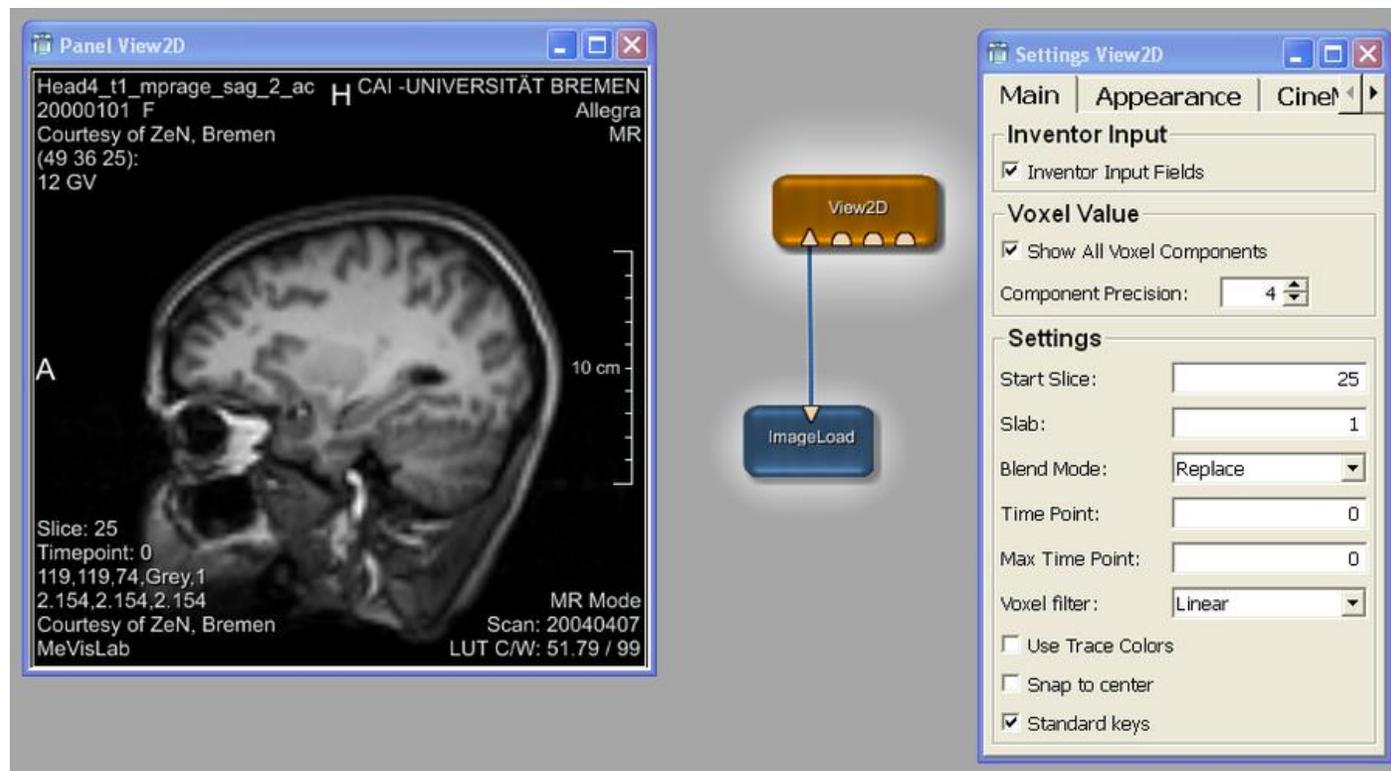
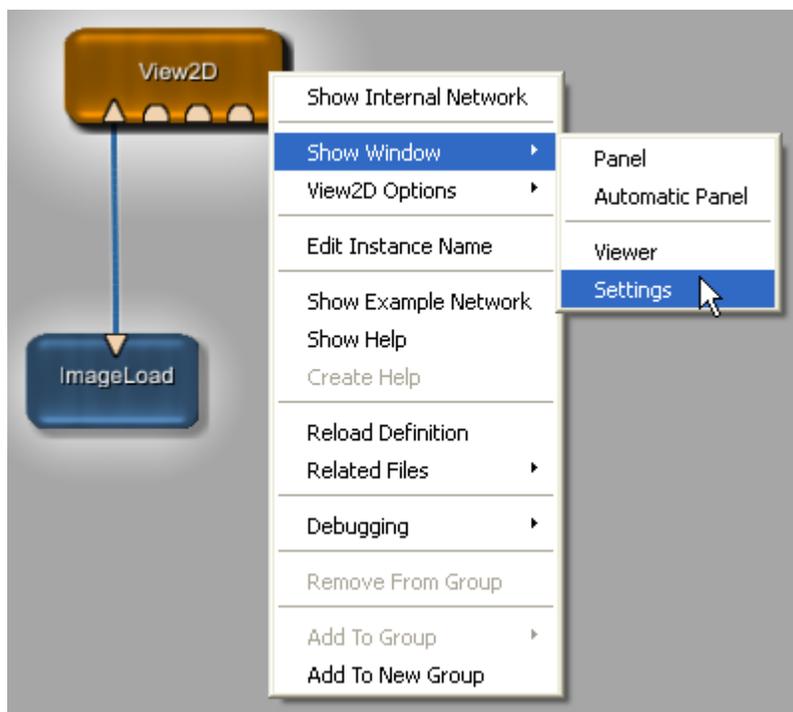


Configurando a conexão

- O módulo *View2D* tem um conector de entrada para imagem, e três entradas do *Inventor*.

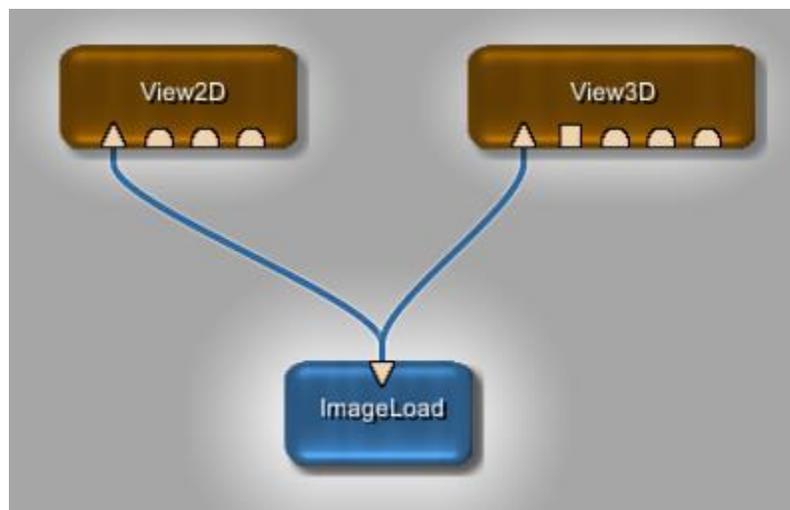


Abrindo o painel de configurações de View2D



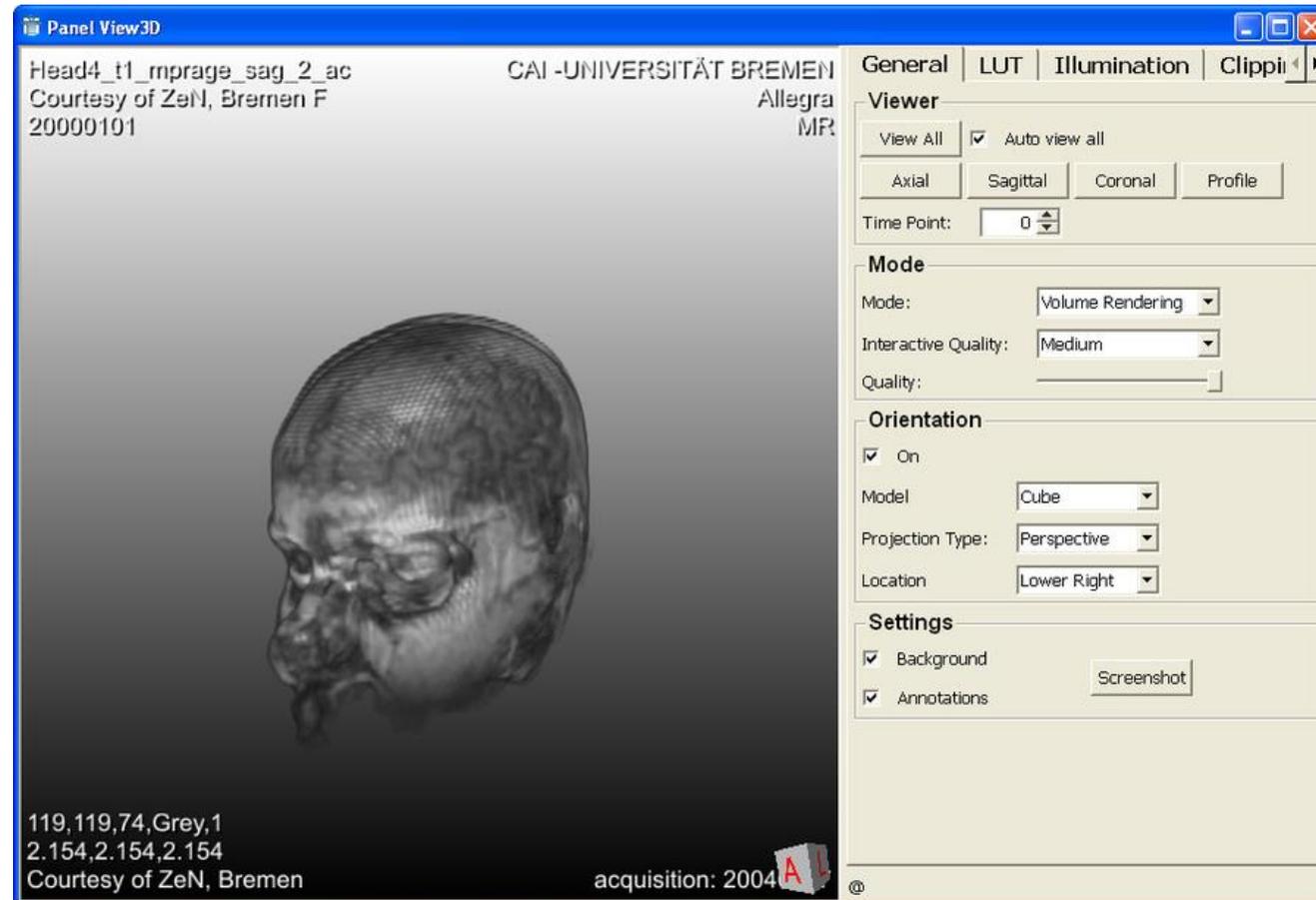
Adicionando o módulo View3D

- Adicionando o módulo *view3D* temos acesso a uma cena em 3D da nossa imagem.



Painel View3D

- O *View3D* tem varias configurações, onde você pode definir detalhes de exibição ou mesmo gravar um filme.

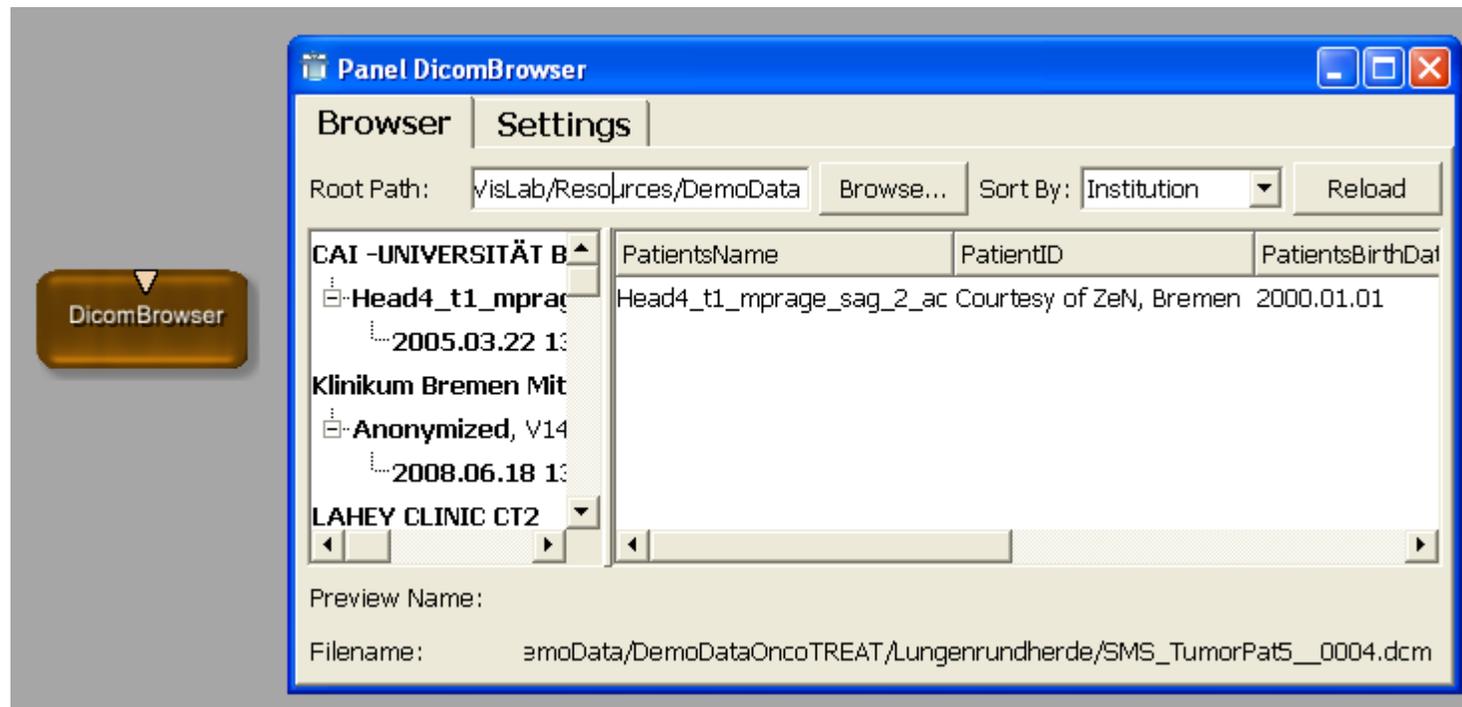


Formas alternativas para carregar imagens

- Além da maneira descrita acima, existem variações.
- Uma opção é arrastar a imagem (DICOM ou TIFF) para o *workspace*, assim um módulo *ImageLoad* é criado automaticamente.

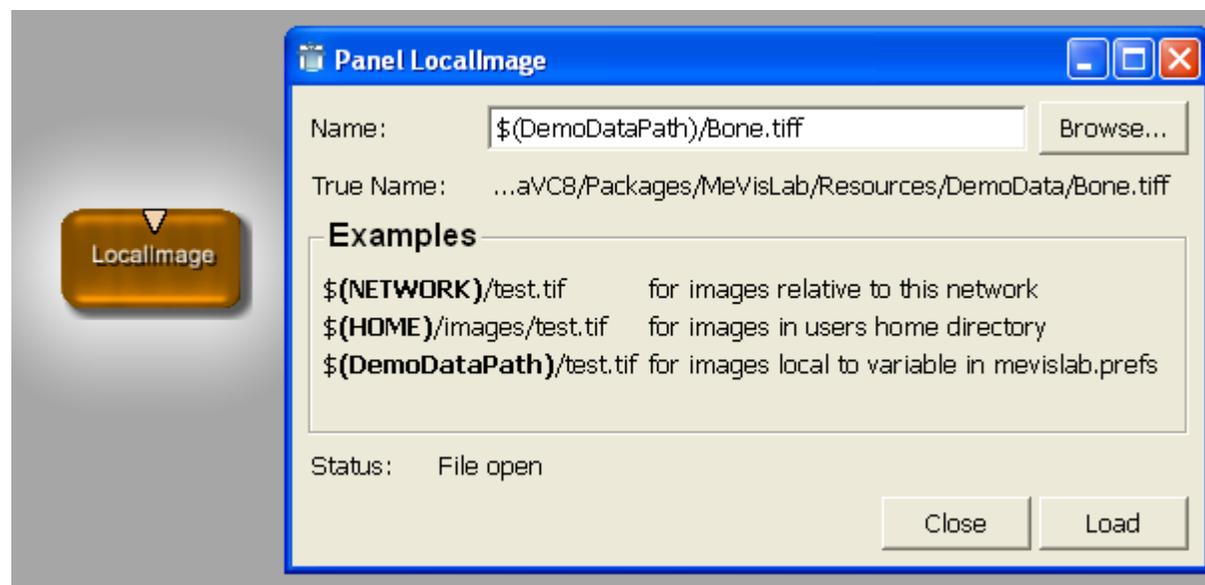
Adicionando imagens via DICOM Navegador

- Utilizando o *DicomBrowser*, imagens DICOM podem ser classificadas com tags, como por exemplo, instituição, paciente, modalidade e etc.



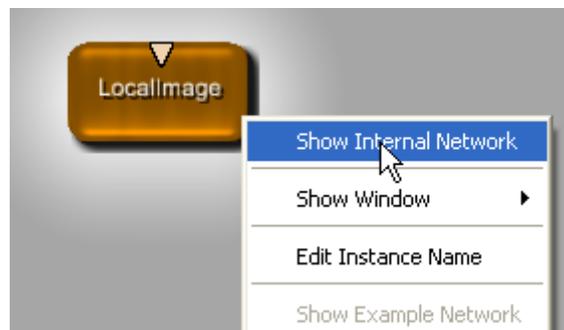
Usando o módulo *LocalImage*

- A diferença para esse módulo é apenas no script que oferece diferentes cominhos ao invés de caminhos absolutos no arquivo (*ImageLoad*).

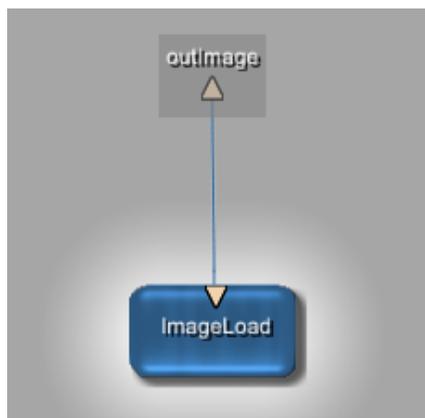


Usando o módulo *LocalImage*

- Módulos de macro são uma combinação de uma rede interna e um script. Você pode abrir a rede interna através do menu de contexto do módulo.

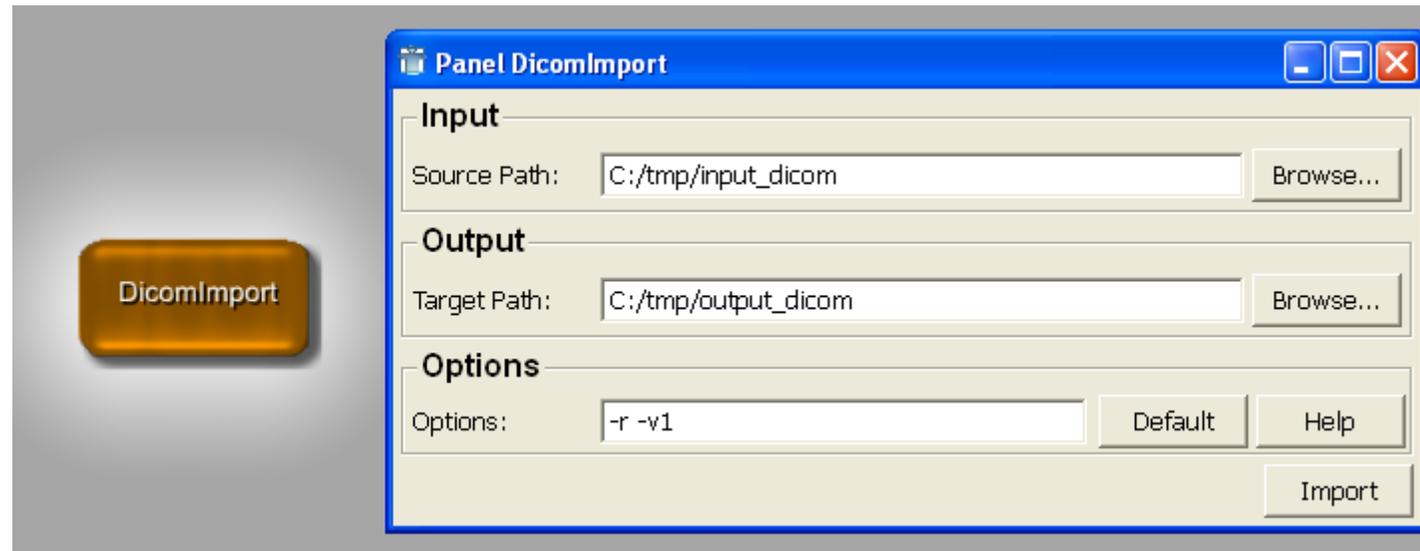


- No caso do *LocalImage* a rede interna consiste de uma *ImageLoad* apenas.

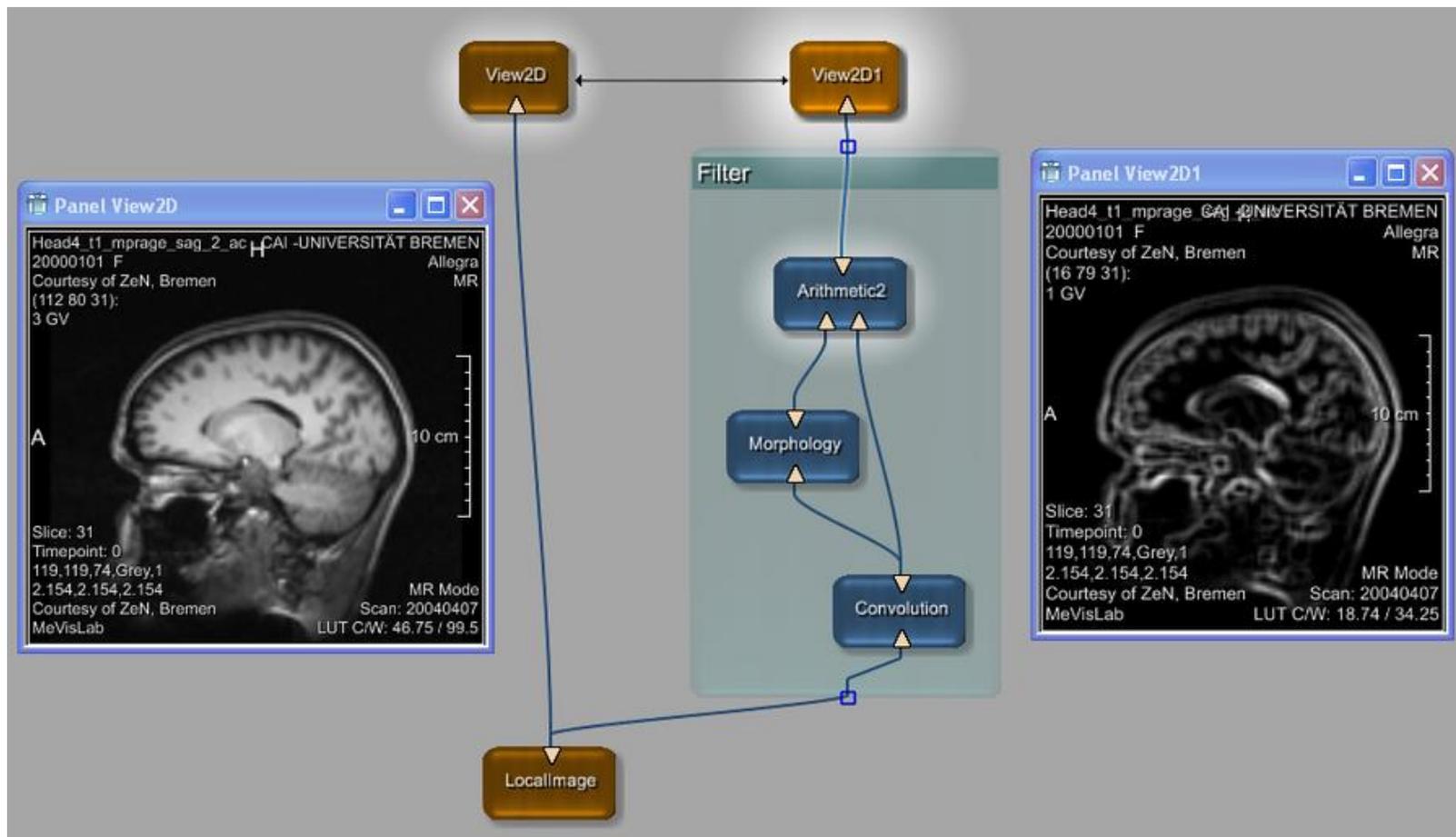


Importação DICOM Images

- A importação DICOM é fornecida pelo modulo *DicomImport*.



Implementação de um filtro de contorno

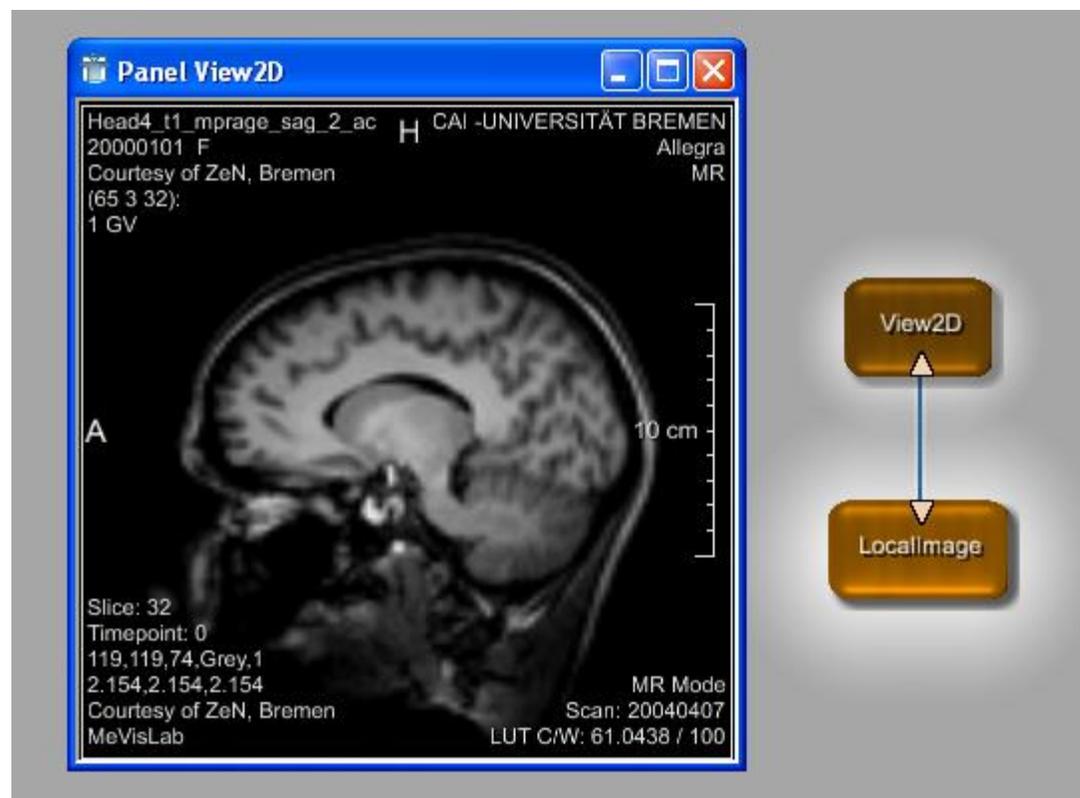


A implementação de um filtro de contorno

- Primeiramente precisamos carregar uma imagem de entrada.
- 1 – Criar uma nova rede.
- 2 – Adicionar *LocalImage* para carregar uma imagem.
- 3 – Adicionar um módulo *View2D* para visualizar a imagem carregada.

A implementação de um filtro de contorno

- Exemplo



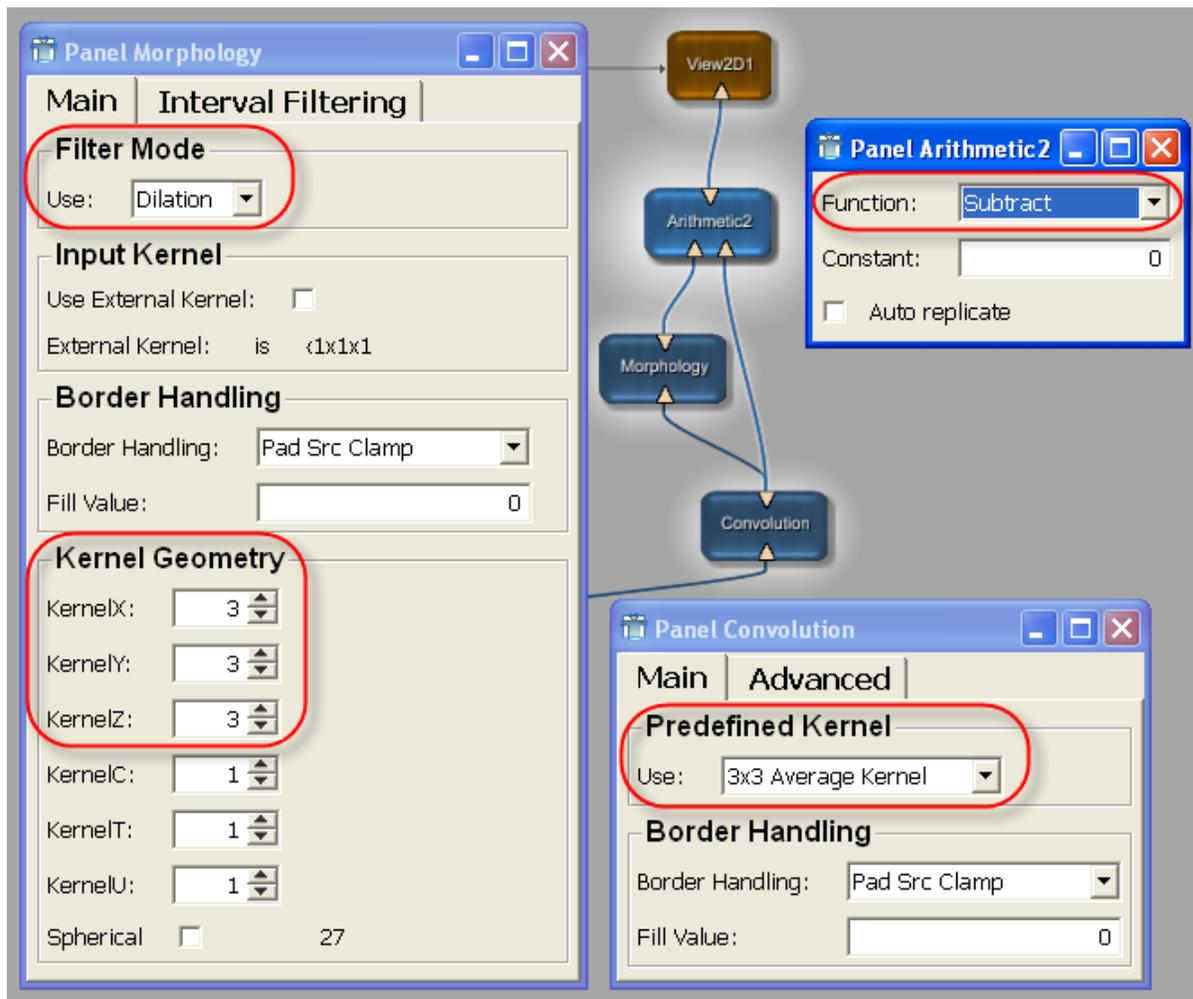
A implementação de um filtro de contorno

- Aplicar um filtro de média na imagem
- Dilatar a imagem utilizando um kernel morfológico
- Subtrair a imagem suavizada da imagem dilatada

A implementação de um filtro de contorno

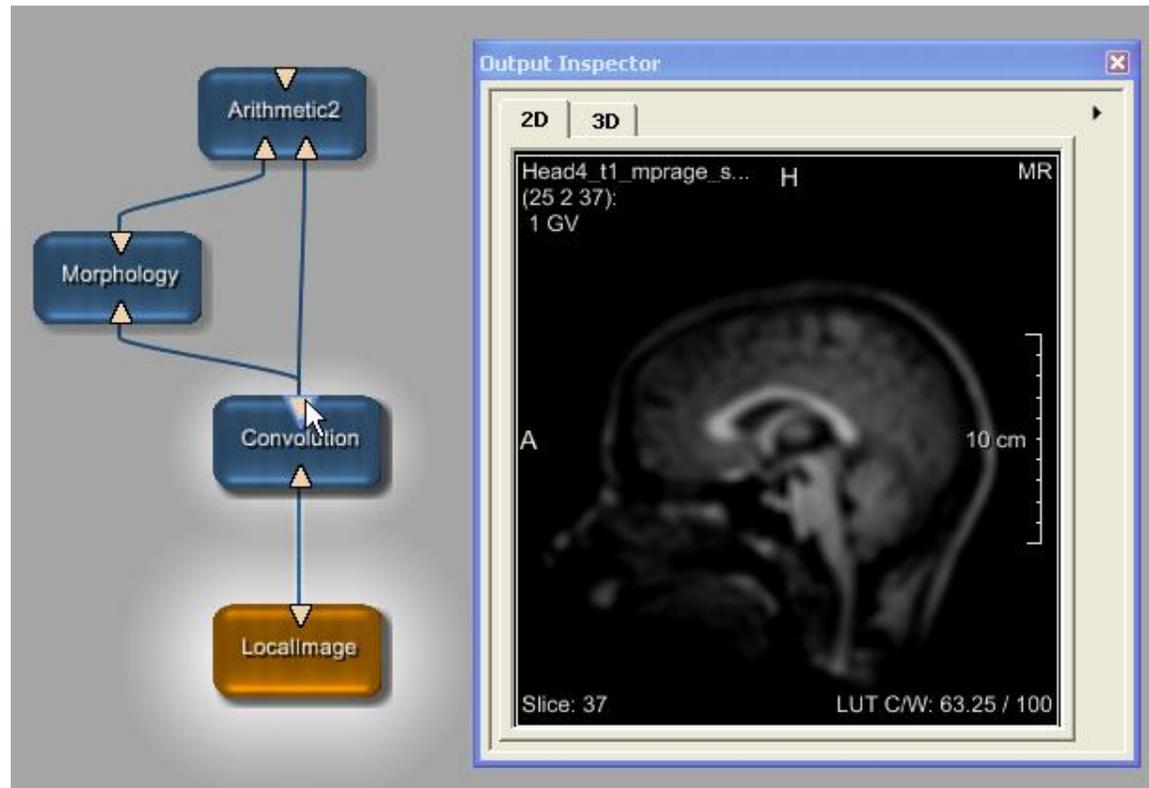
- Adicione os módulos *Convolution*, *Morphology* e *Arithmetic2* à rede
- Modules -> Filters -> Kernel -> Convolution
- Modules -> Filters -> Morphology -> Morphology
- Modules -> Analysis -> Arithmetic -> Binary -> Arithmetic2

Ajustar os parâmetros do filtro

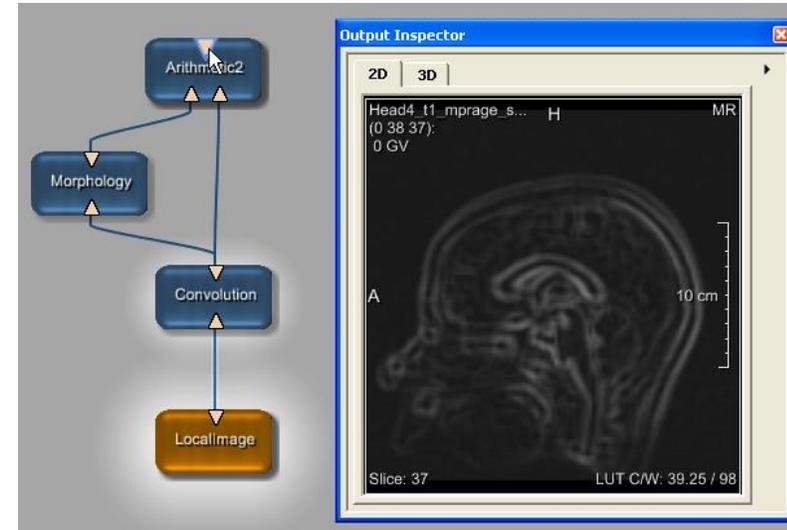
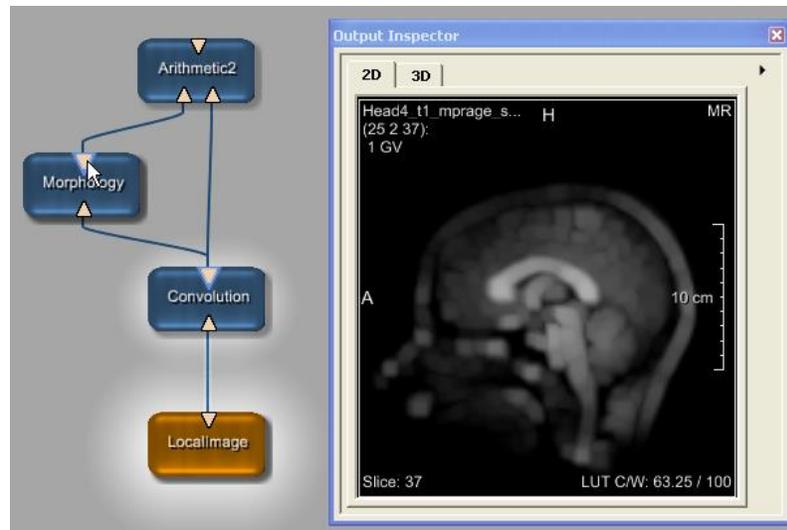
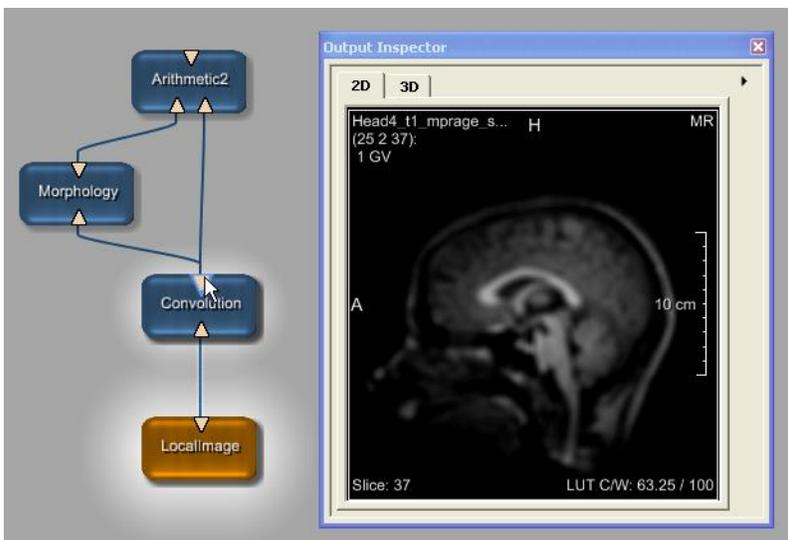


Resultado

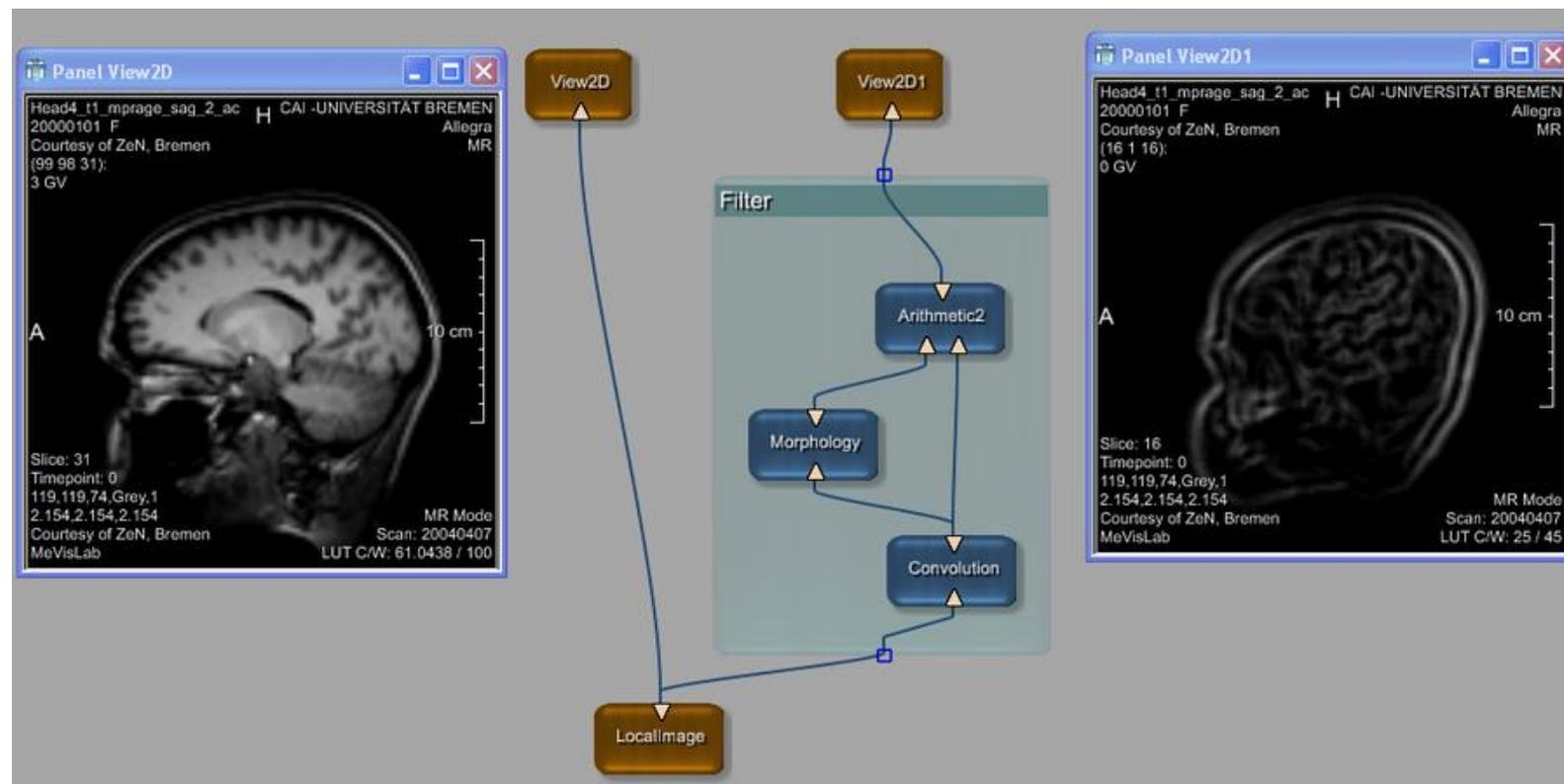
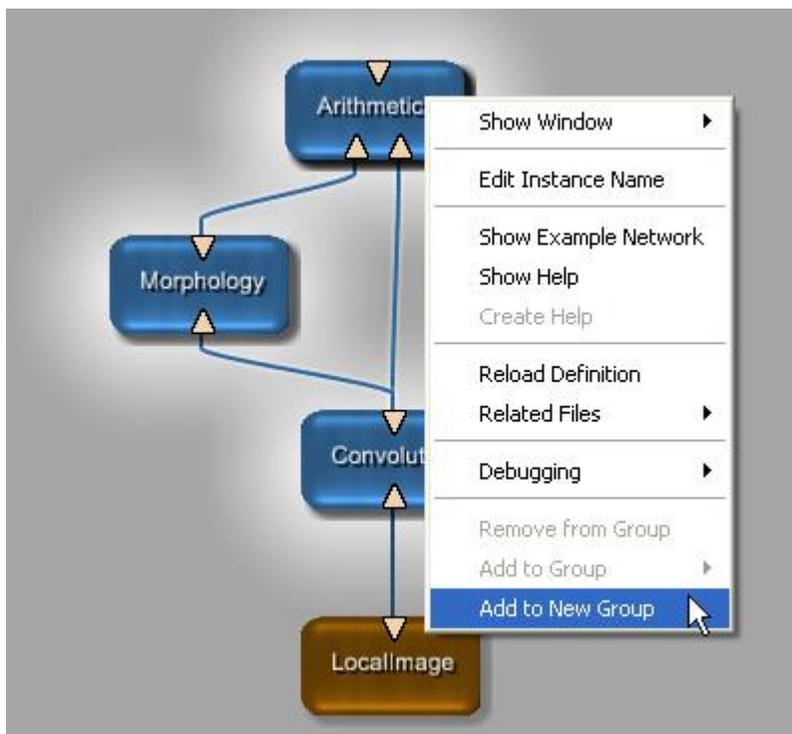
- Clique cada conector para seguir o processamento de imagem.



Resultado



Criando um grupo



Conexão de parâmetro para Sincronização

- Além de conexões de dados entre as entradas e saídas dos módulos (Imagem, Inventor e conectores Base) também é possível conectar campos dos módulos através de uma conexão de parâmetros. Os valores das áreas ligadas estão sincronizados, o que significa que a alteração do valor de um campo irá alterar todas as áreas ligadas a este campo.

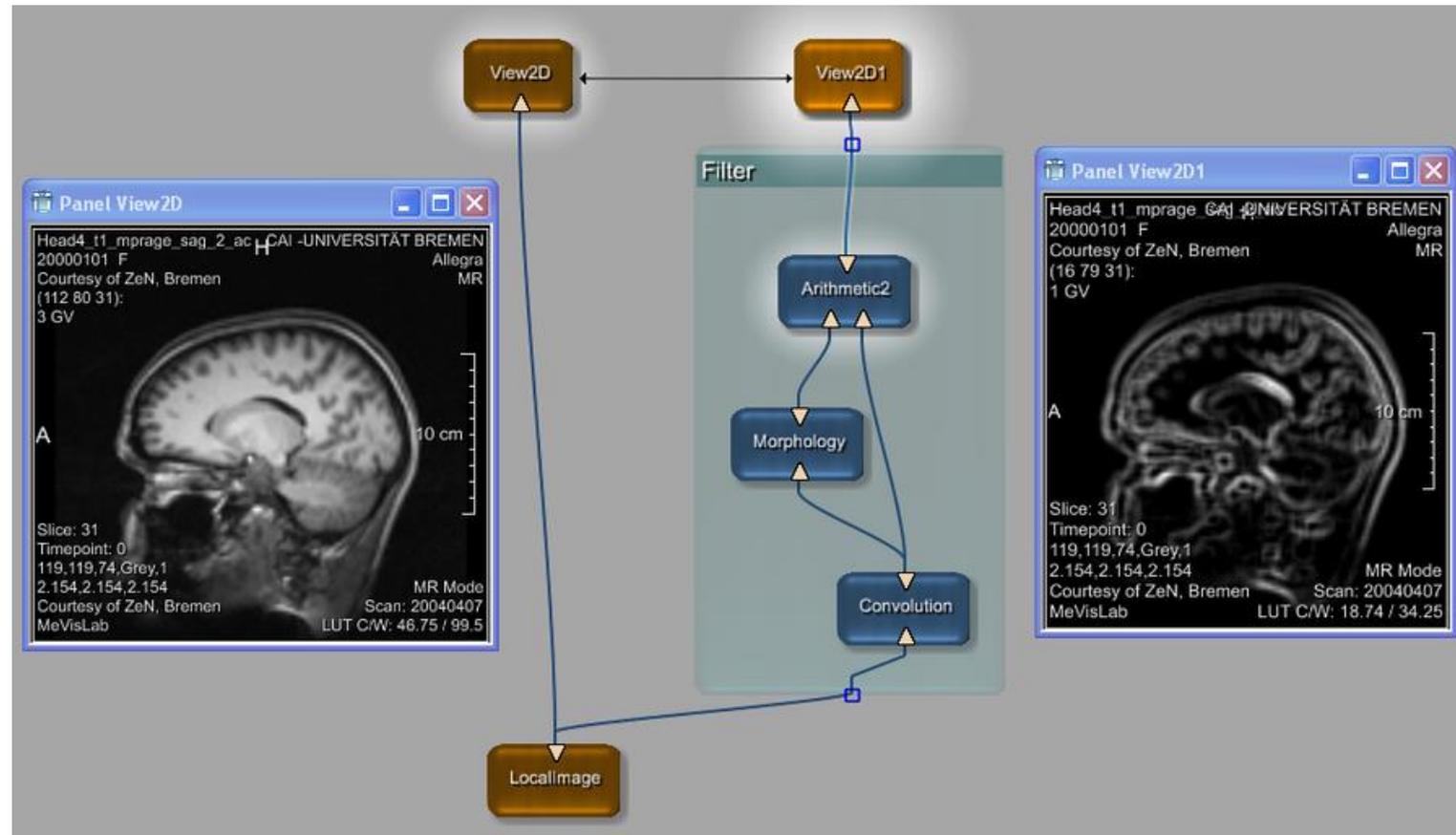
Panel View2D					Panel View2D1				
Parameters		Inputs		Outputs	Parameters		Inputs		Outputs
Name	Type	In	Out	Value	Name	Type	In	Out	Value
instanceName	String			View2D	instanceName	String			View2D1
inventorInputOn	Bool			FALSE	inventorInputOn	Bool			FALSE
view2DExtensionsOn	Bool			TRUE	view2DExtensionsOn	Bool			TRUE
startSlice	Integer			31	startSlice	Integer			31
numSlices	Integer			1	numSlices	Integer			1
numXSlices	Integer			1	numXSlices	Integer			1

Unidirecional
Bidirecional

Varias saída apenas uma entrada

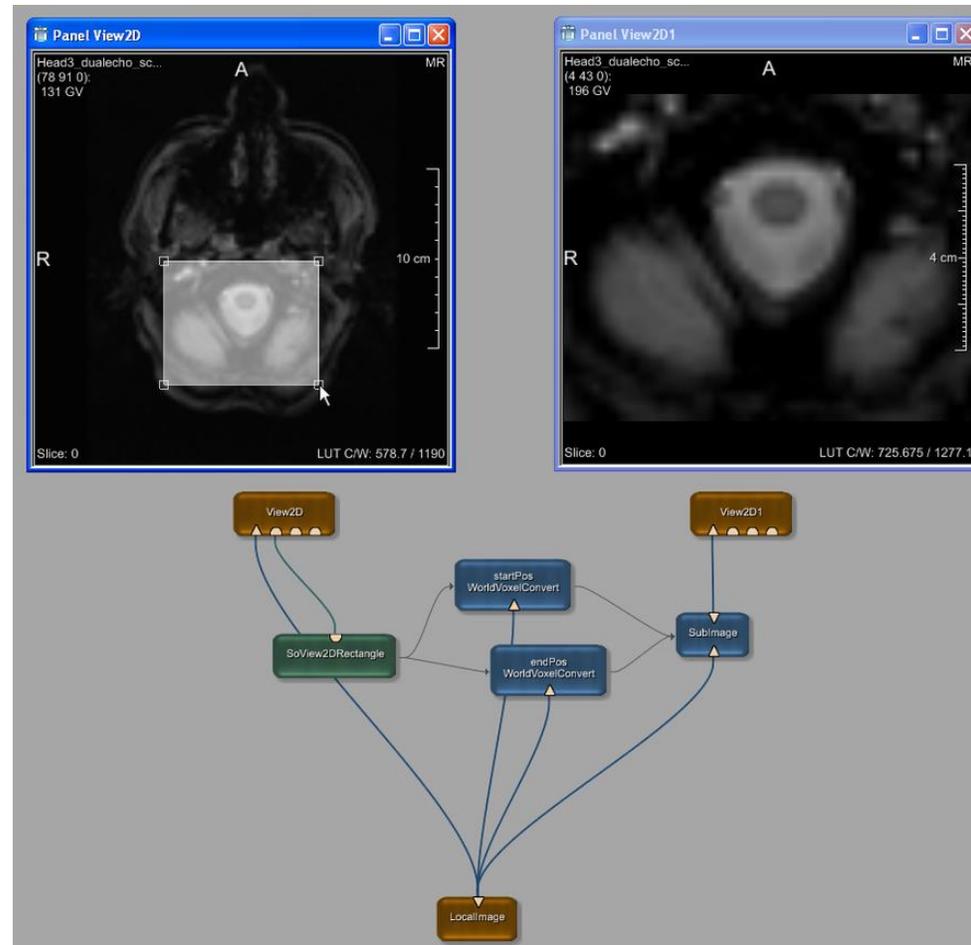
Resultado

- Como resultado temos a mesma fatia da imagem em ambos os *Viewers*.



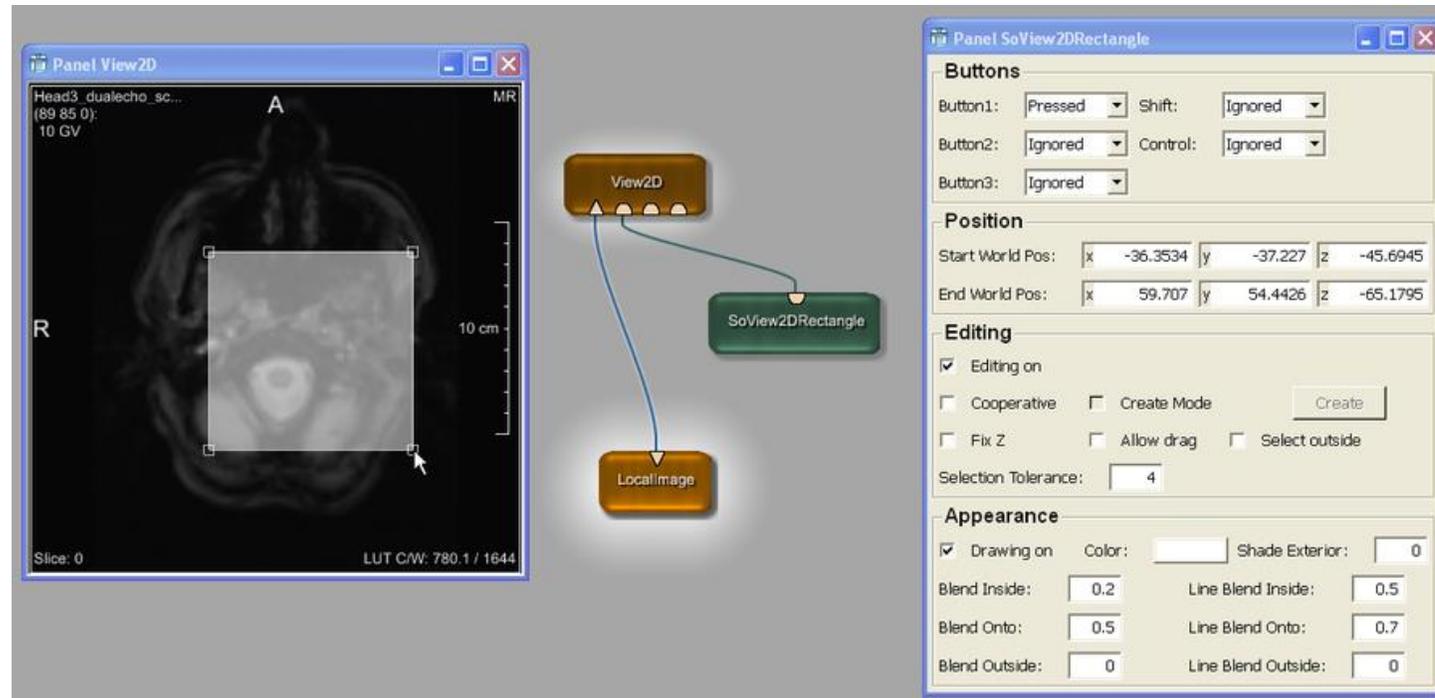
Definindo uma região de interesse

- Criar uma rede que permite a seleção de uma região 2D no primeiro *viewer*, então a região selecionada é exibida como uma sub-imagem no segundo *viewer*.



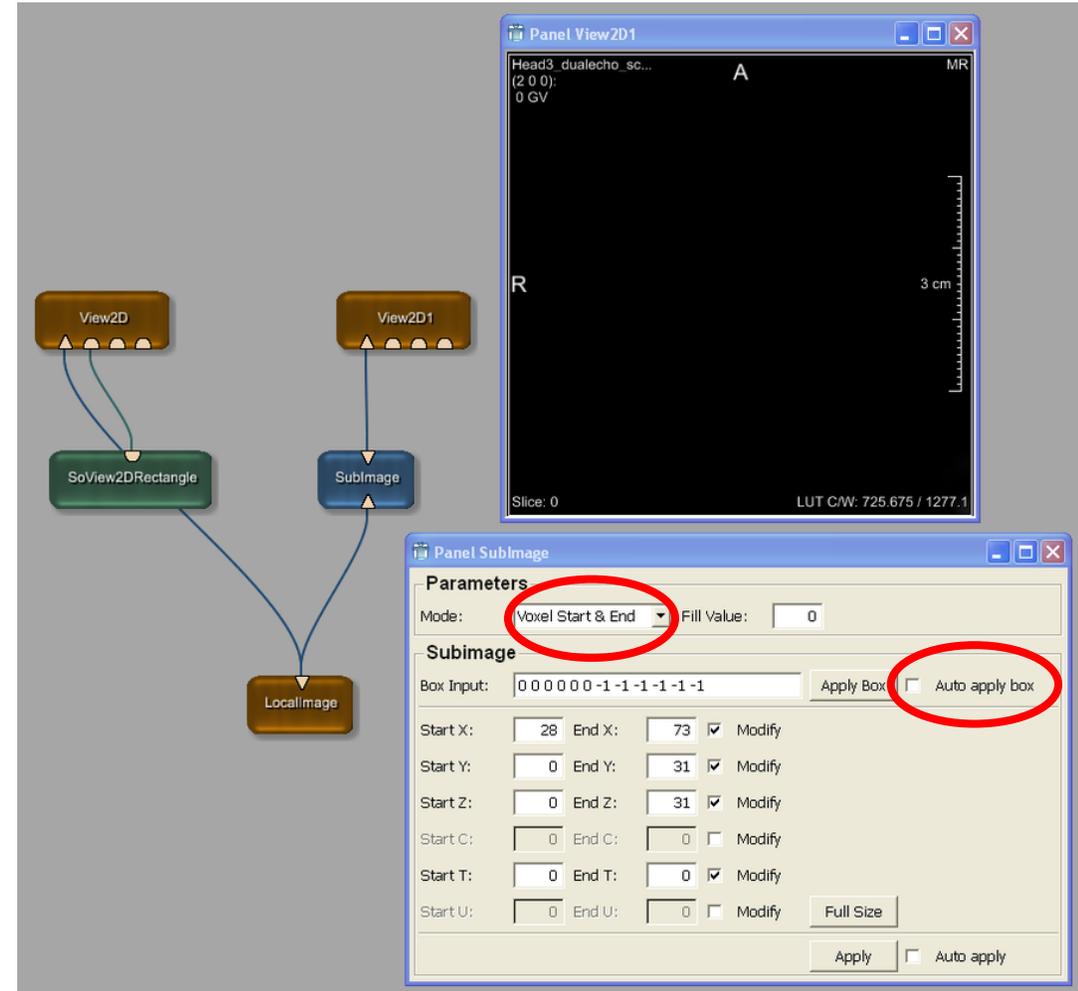
Criando um visualizador com um *Selection Rectangle*

- A primeira parte é a construção de uma rede simples com um módulo *ImageLoad*, um *Viewer*, e um módulo que permite desenhar um retângulo de seleção.



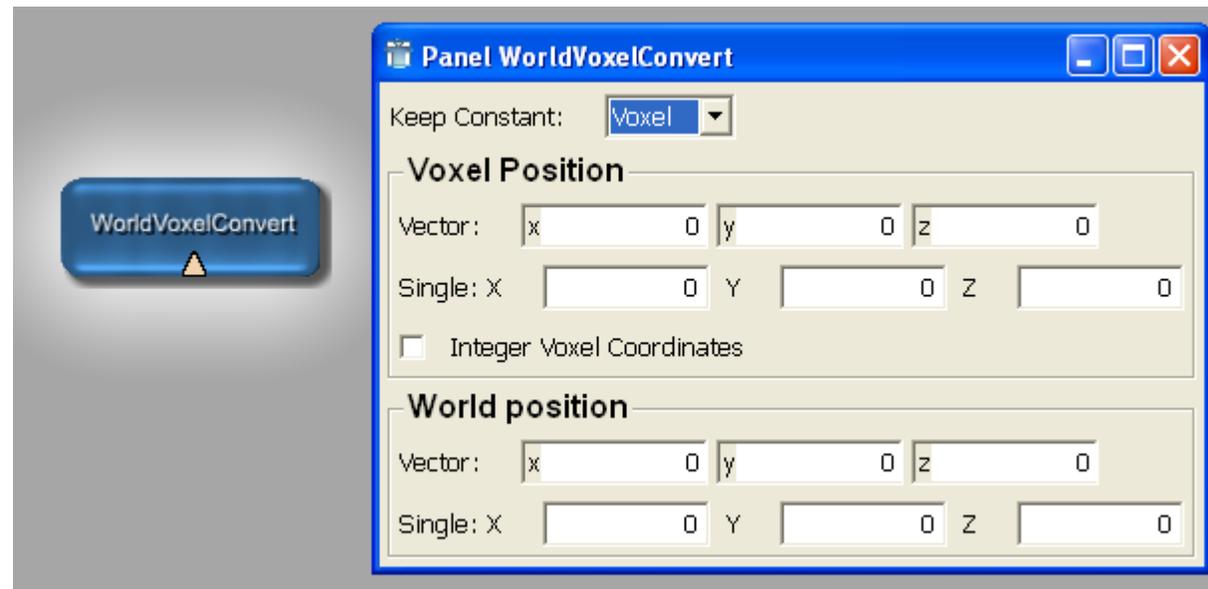
Adicionando um segundo *Viewer* para a sub- imagem

- Adicionar uma módulo de *SubImage*
- E outro módulo *View2D*



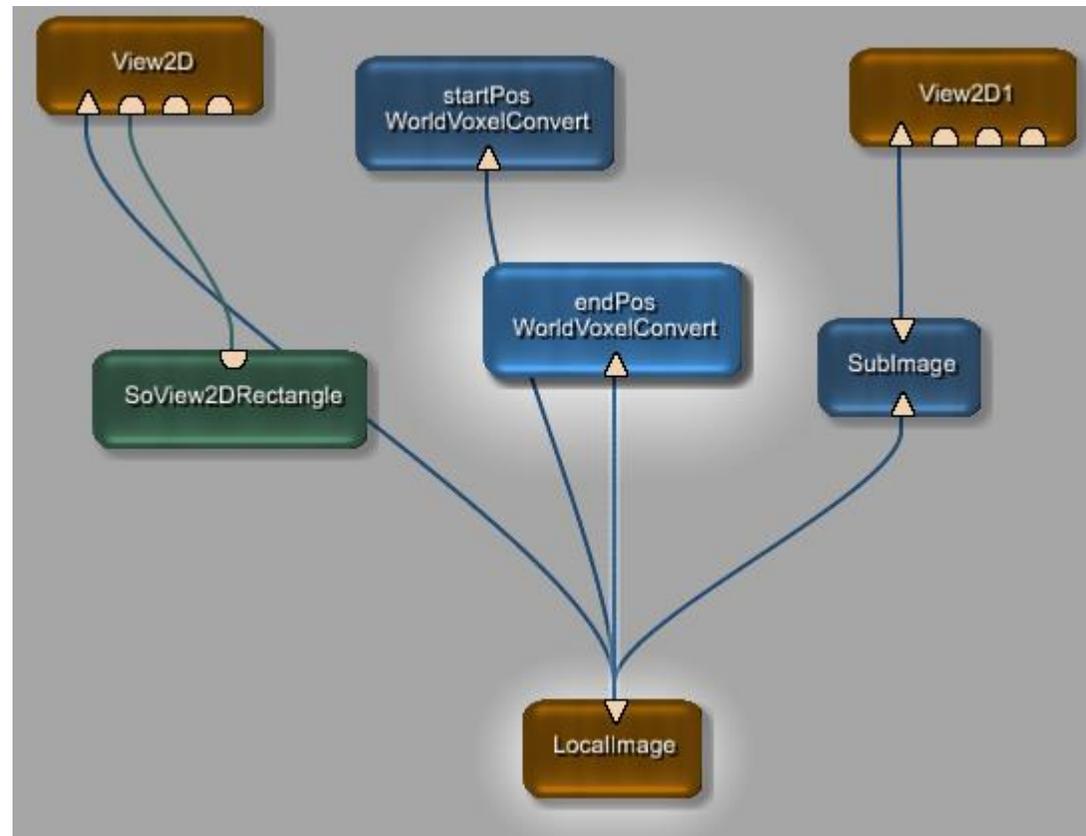
Painel WorldVoxelConvert

- Para tais tarefas de tradução, existem vários módulos que convertem valores de um tipo para o outro.



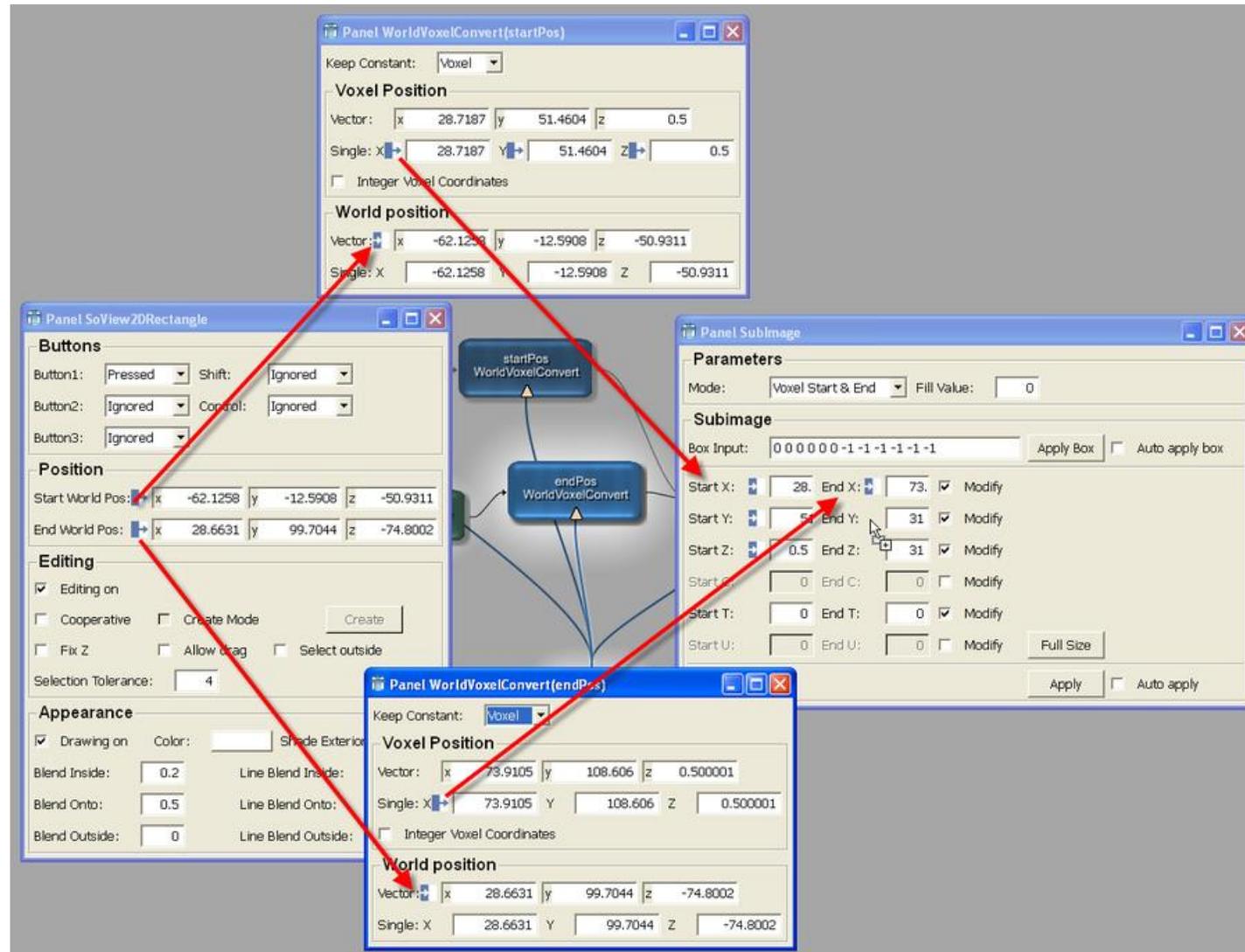
WorldVoxelConvert

- No nosso caso, precisamos de duas conversões, para o início e o fim.

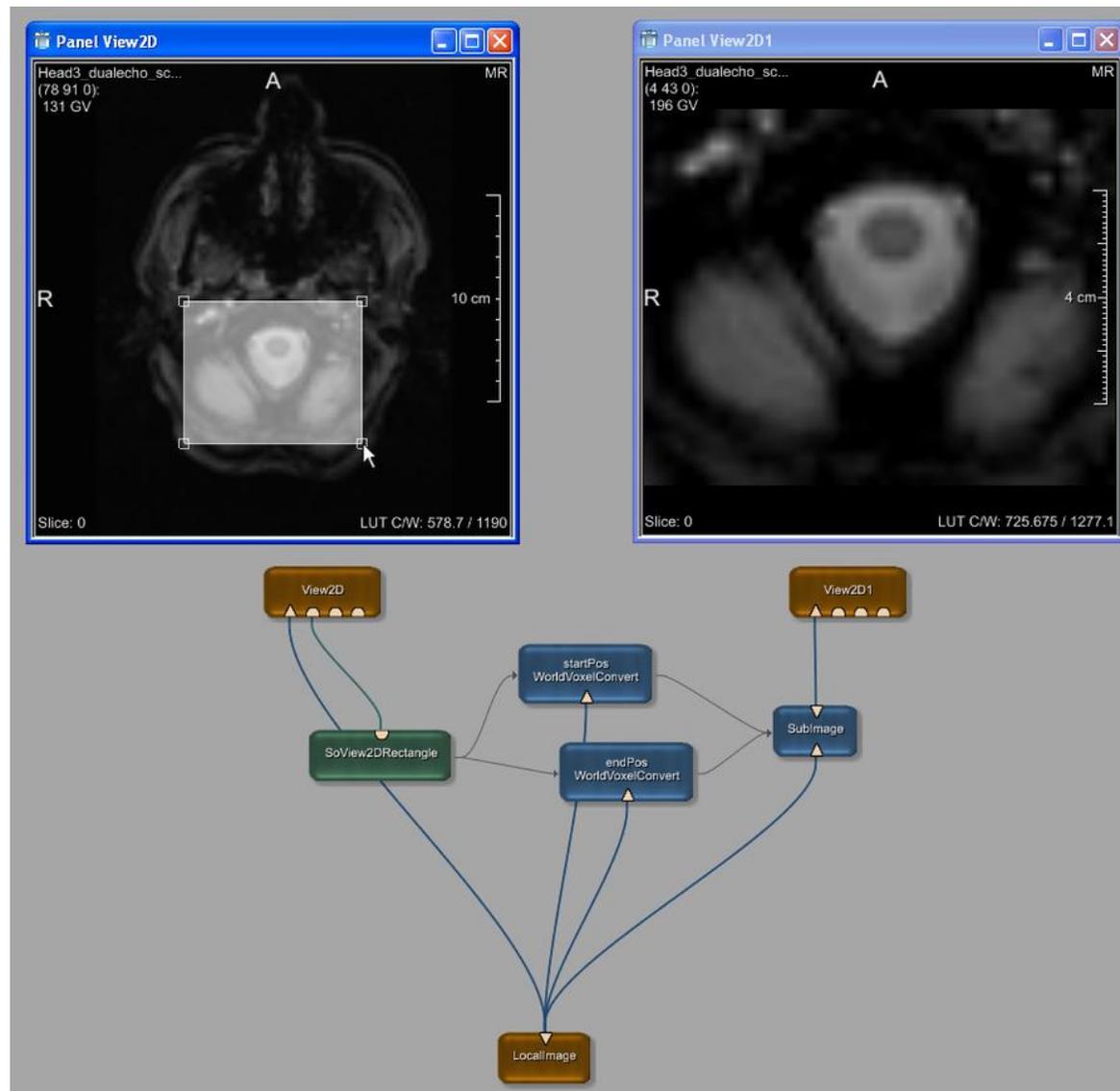


Renomear os módulos para facilitar o entendimento

Ligando conexões entre parâmetros



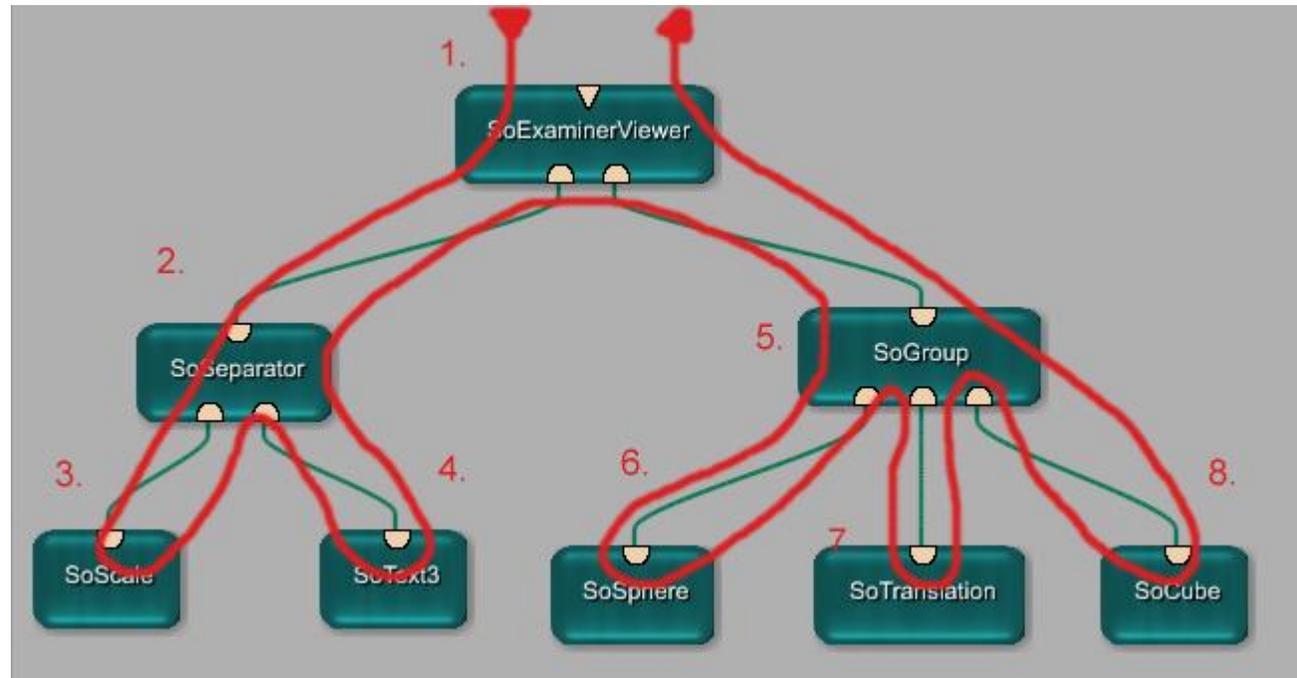
Resultado



Introdução ao Open Inventor

- Open Inventor é uma toolkit 3D orientada a objetos desenvolvida pela Silicon Graphics (SGI) que oferece uma solução abrangente para os problemas de programação de gráficos interativos.

Orientação do processo Open Inventor

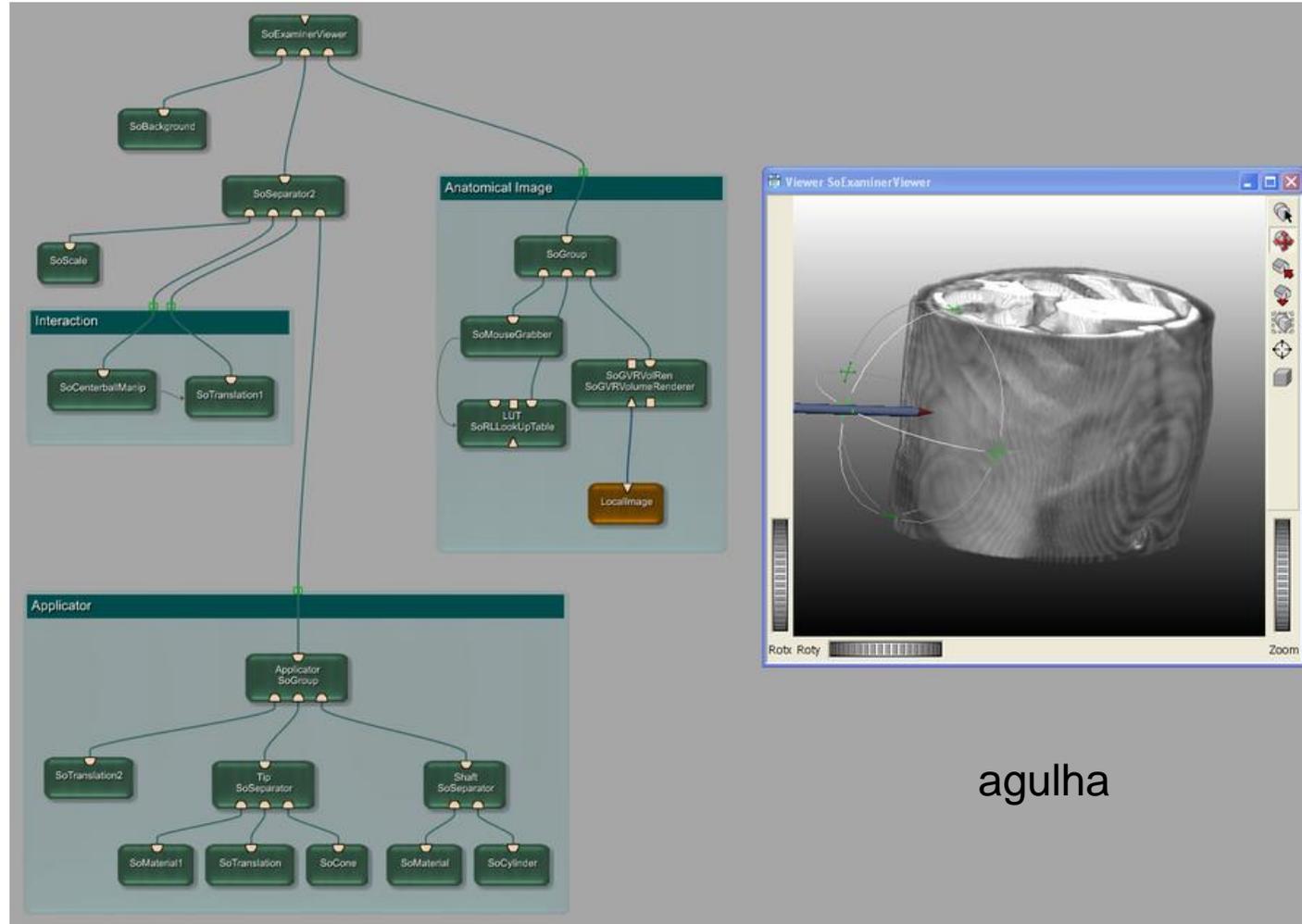


Ordem de processamento

- Mudanças de campo em módulos ML são executados de forma síncrona: a alteração no campo leva a uma execução imediata chamando seu método `handleNotification(Field*)`.
- Mudanças de campo em módulos Open Inventor são executadas de forma assíncrona: As mudanças de campo são armazenados em uma fila de atraso. Em geral, não é conhecida quando esta fila será processada. O processamento pode ser executada chamando `MLAB.processInventorQueue()`.

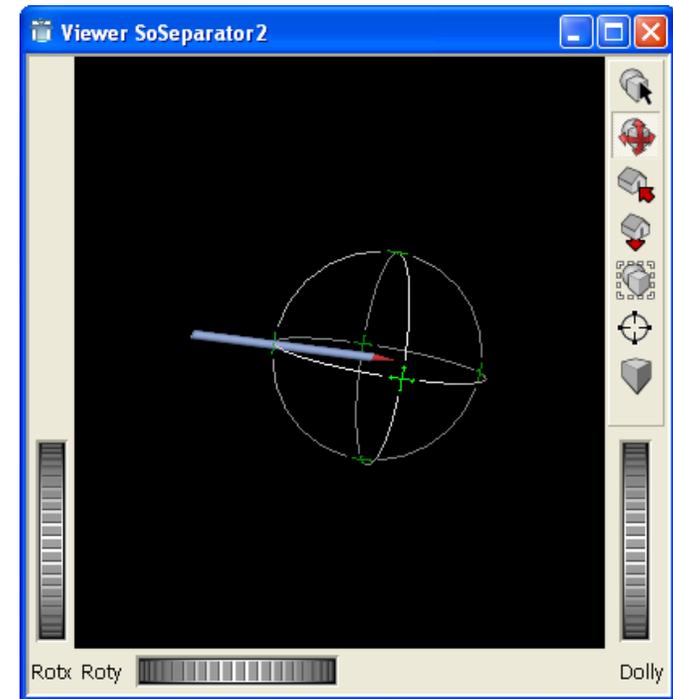
Criando um Open Inventor Scene

- Abaixo exemplo do modelo que iremos criar



A agulha

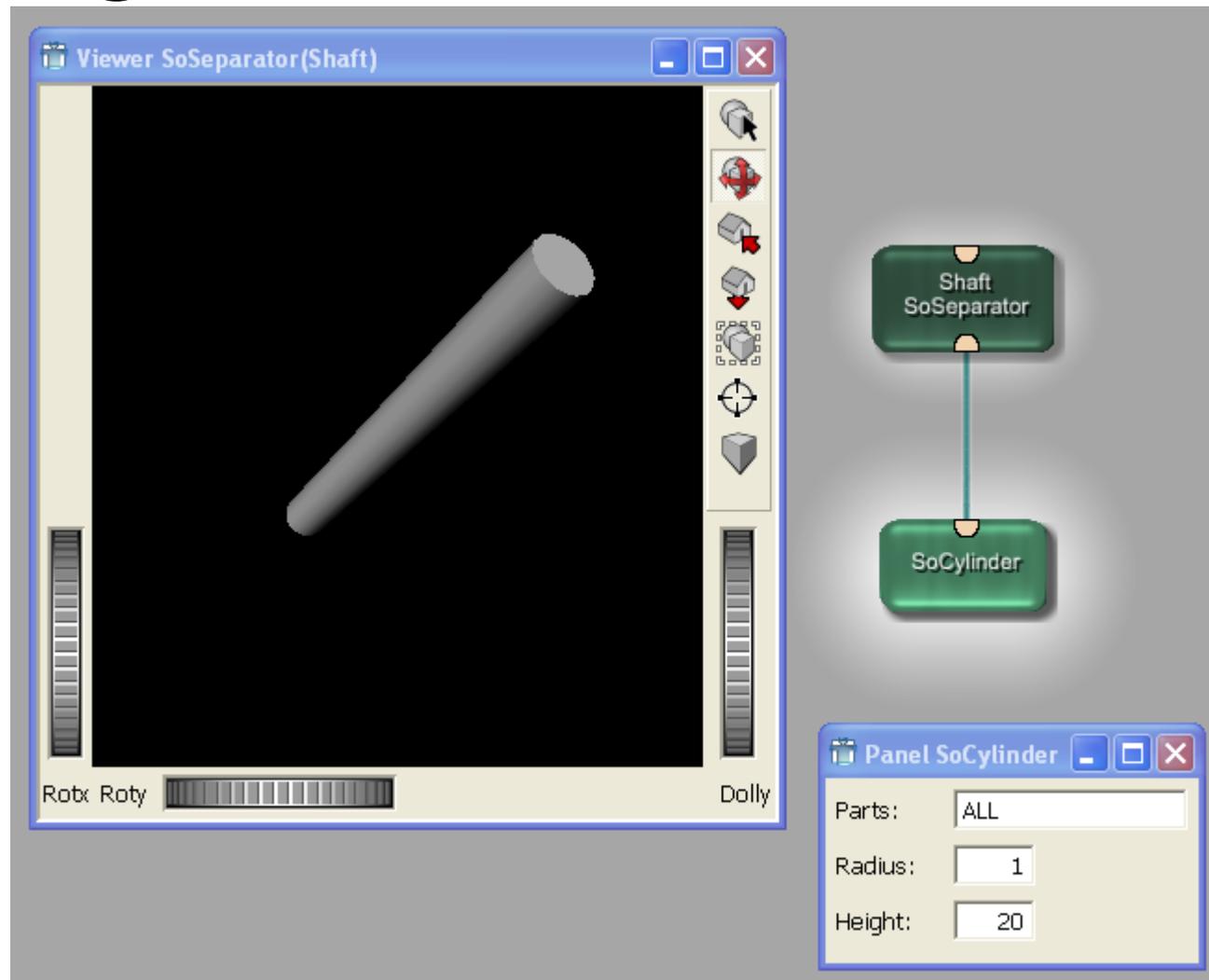
- O agulha deve ser capaz de ser movido dentro do visualizador e também ser capaz de ser reposicionado.
- Os dados devem ser exibidos no modo 3D.



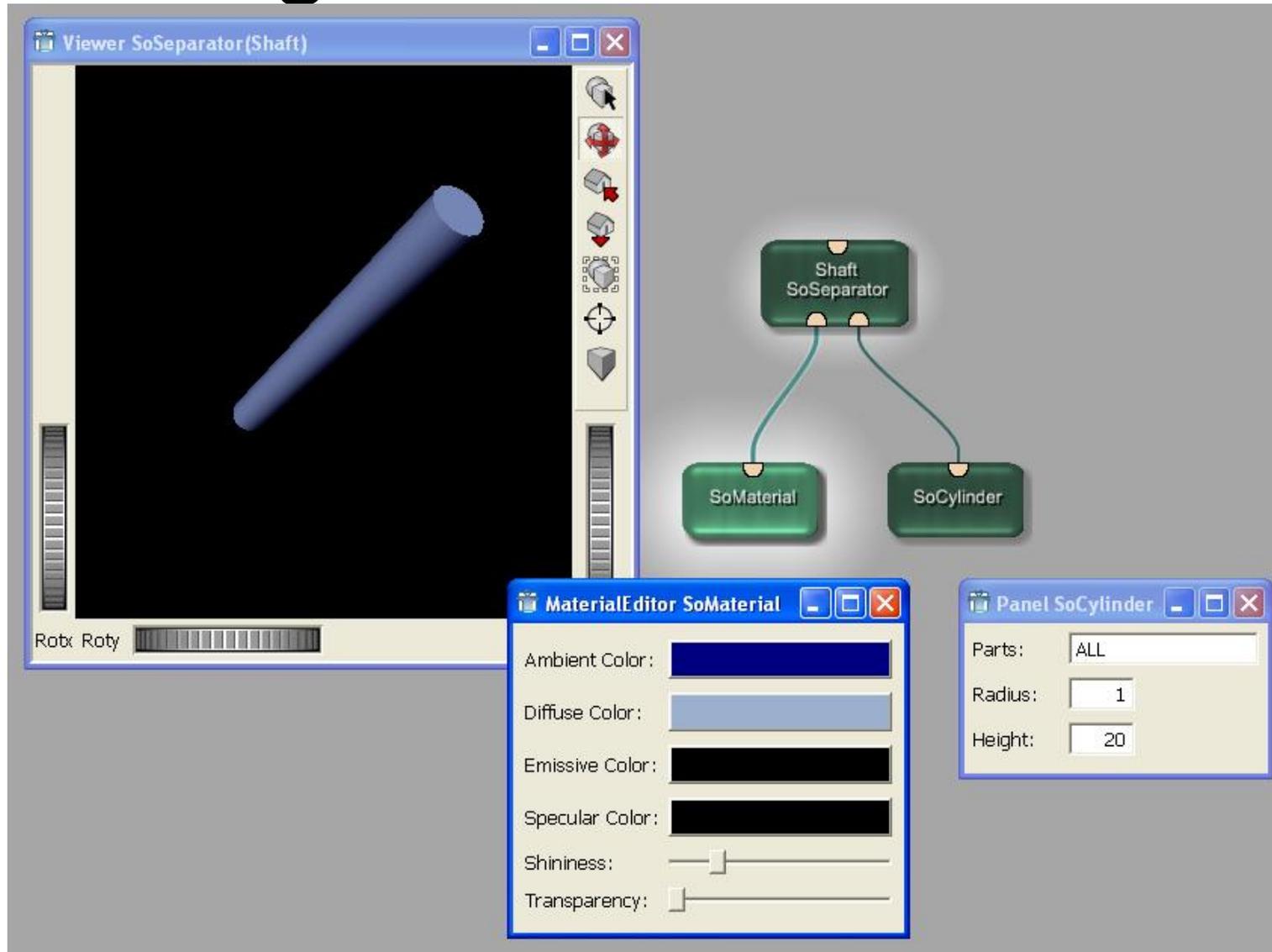
Criando a agulha

- Como primeiro elemento, temos o eixo da agulha. Adicionando um módulo *SoCylinder*.
- Como queremos manter o eixo da agulha e a ponta basicamente independentes, já é possível adicionar um módulo *SoSeparator* que vem com um visualizador embutido.

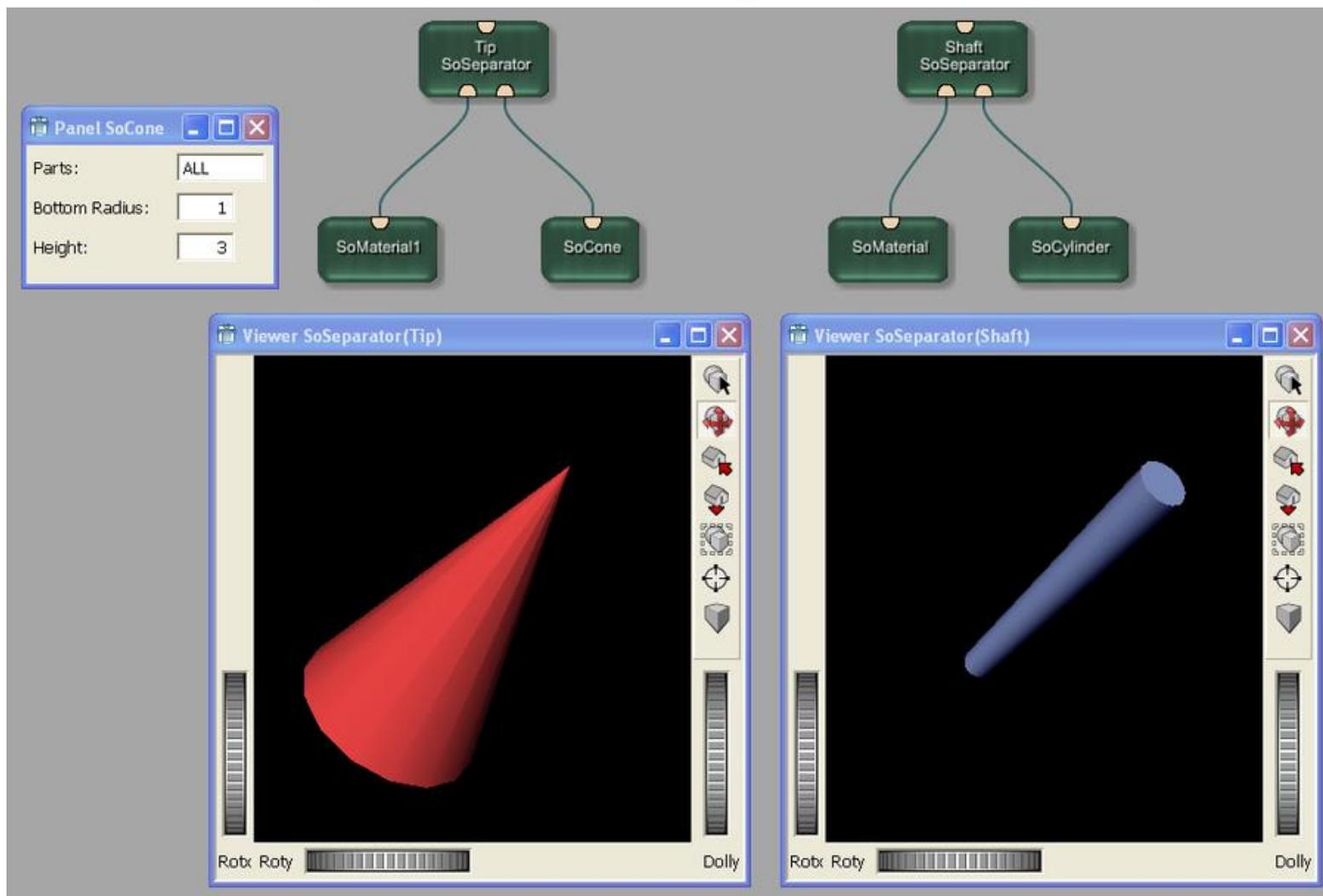
Criando a agulha



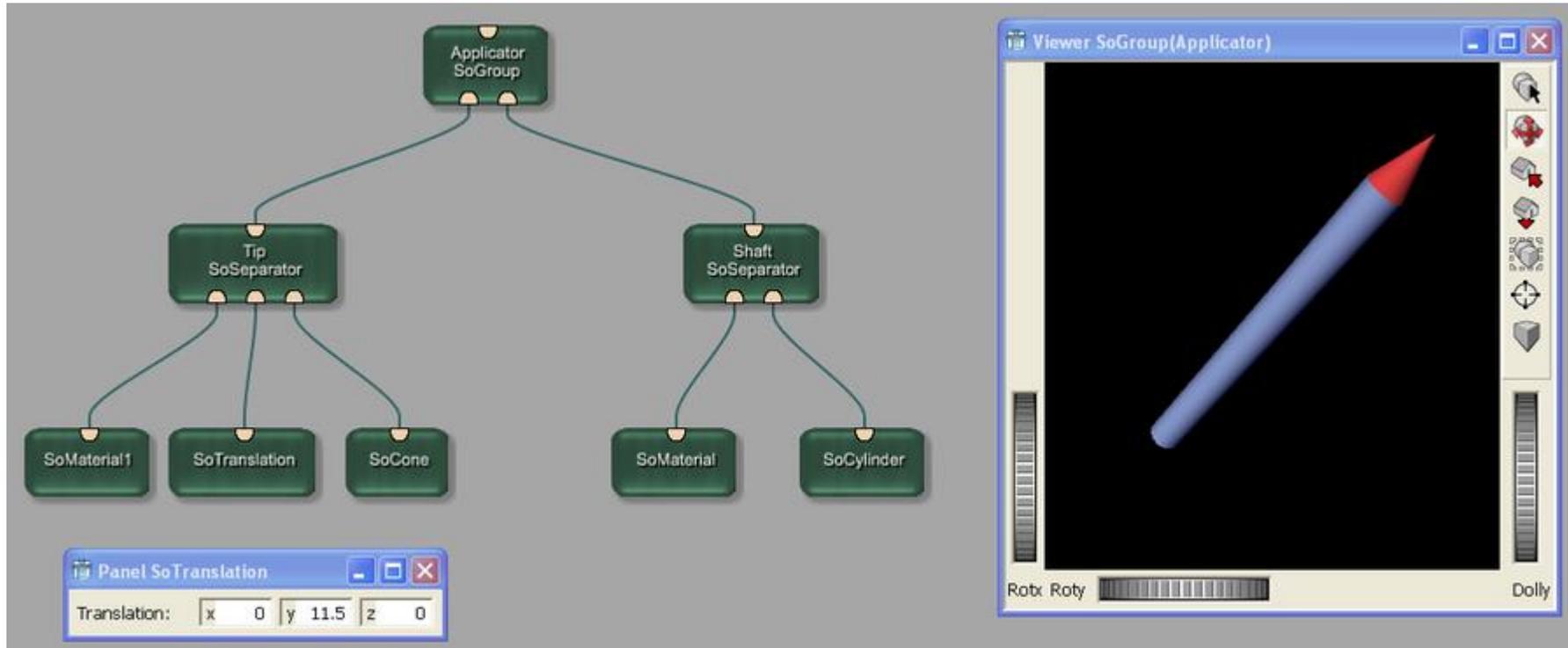
Colorindo a agulha



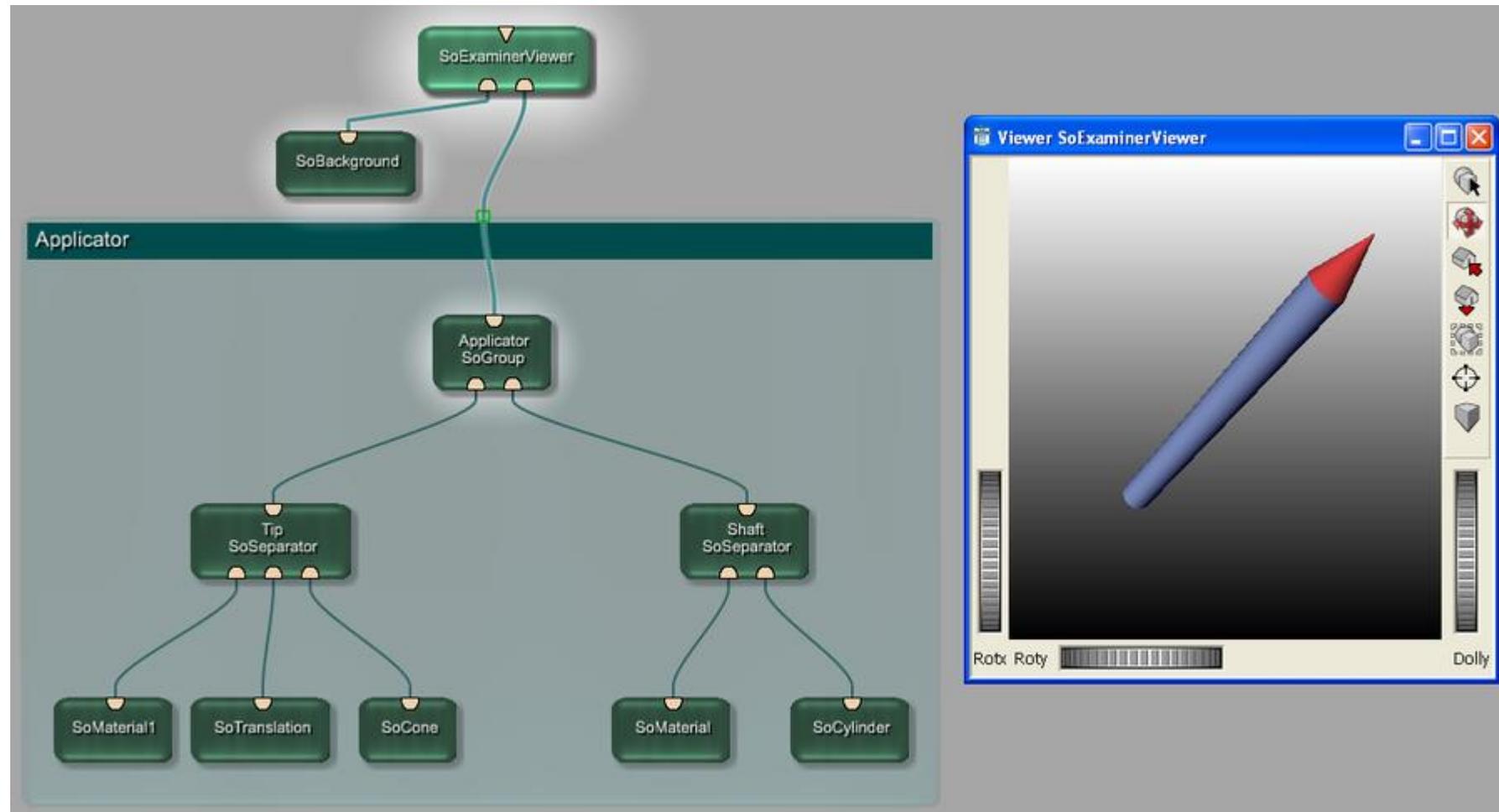
Adicionando a ponta da agulha



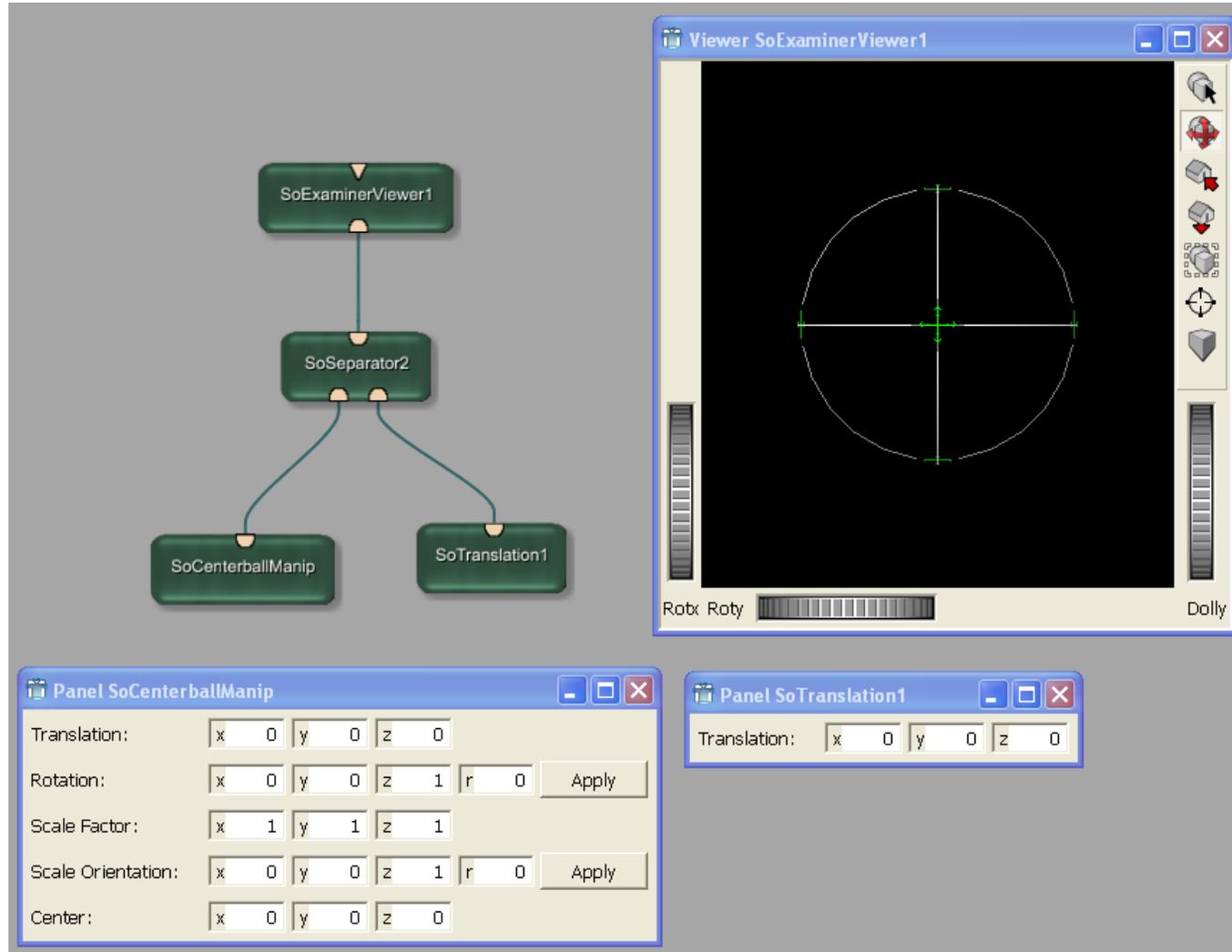
Convertendo e agrupando



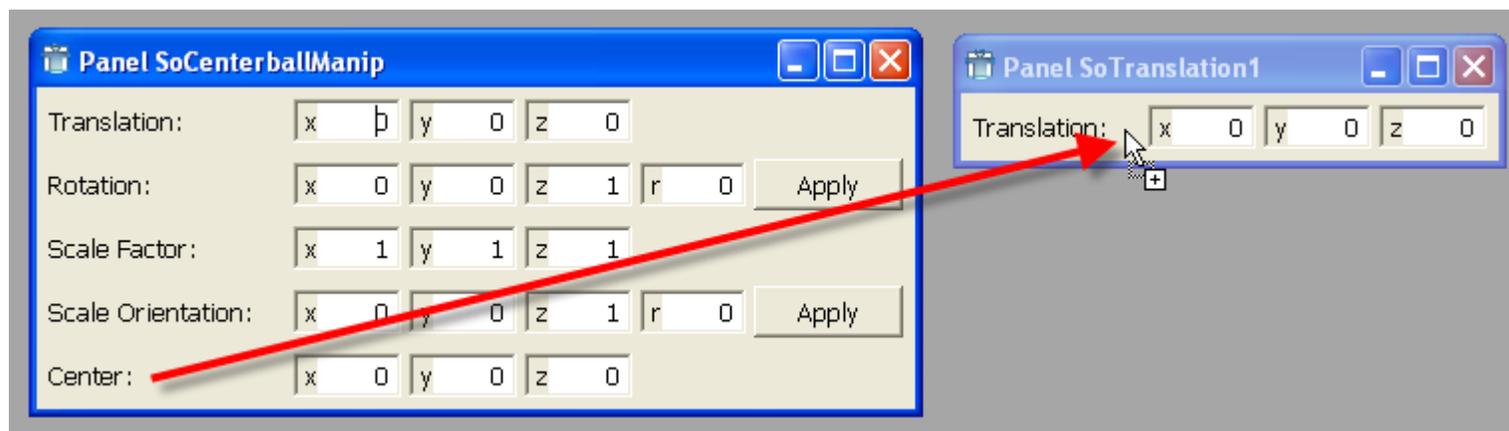
Finalizando



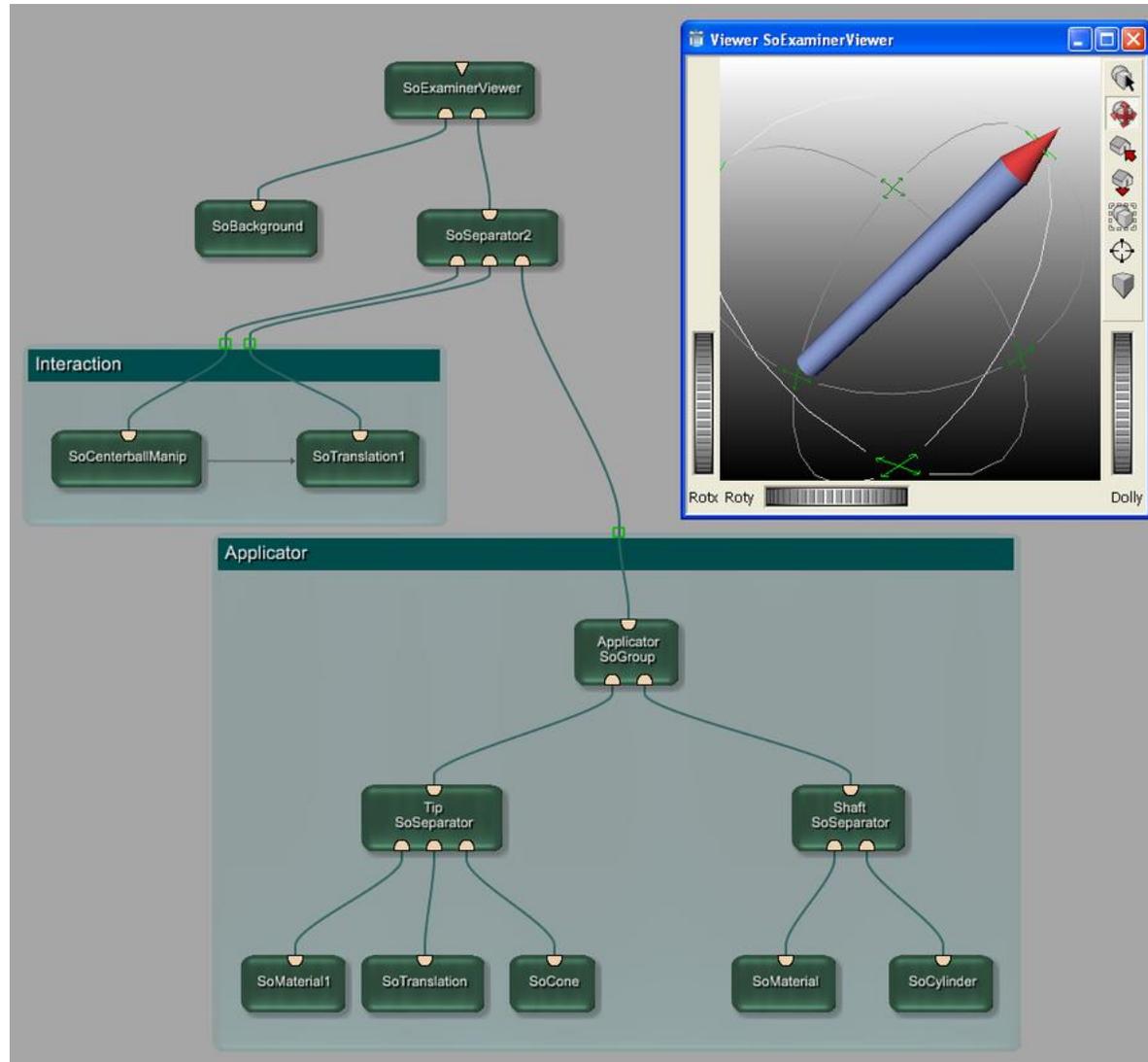
Criando o Interaction



Conectando Parâmetros

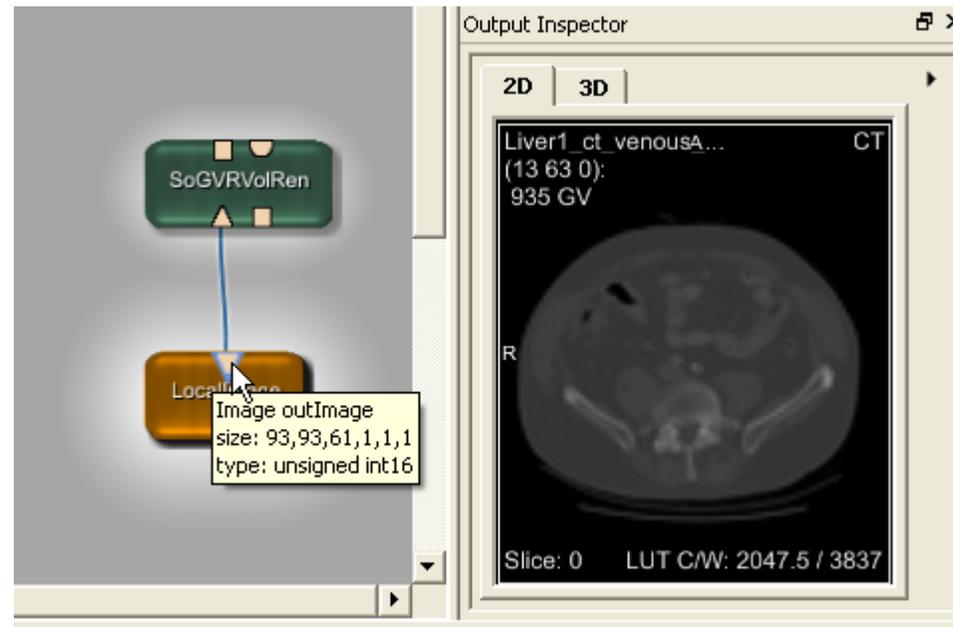


Combinando a interação e a agulha

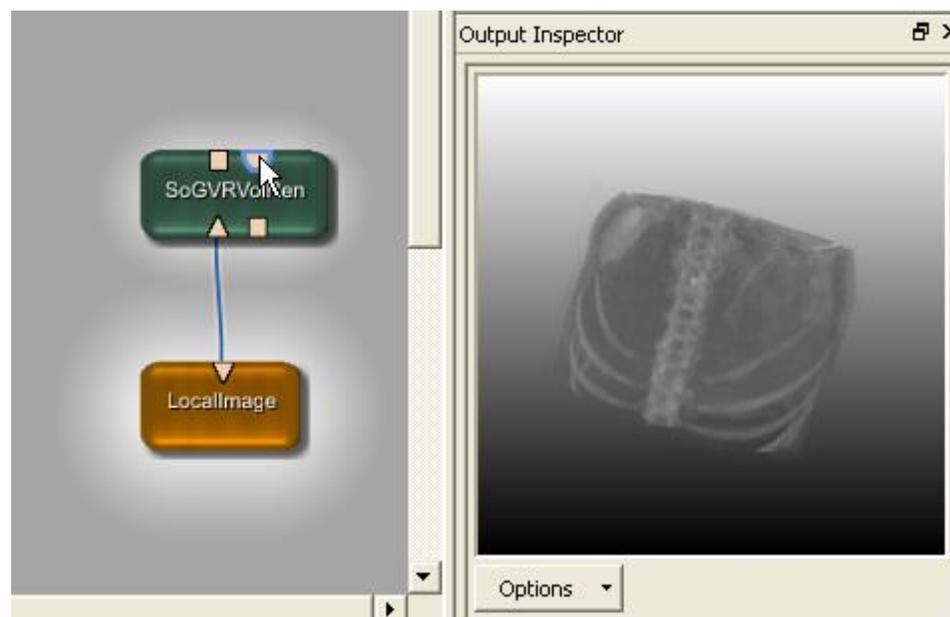


Criando a imagem

- Precisamos da imagem 3D em que o aplicador deve ser posicionado.

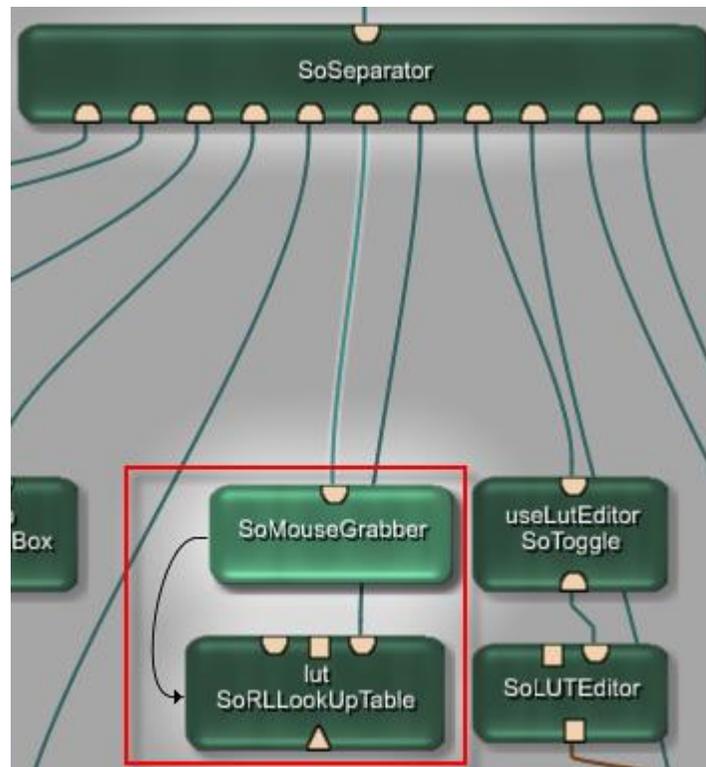


Adicionando o GigaVoxel Renderer



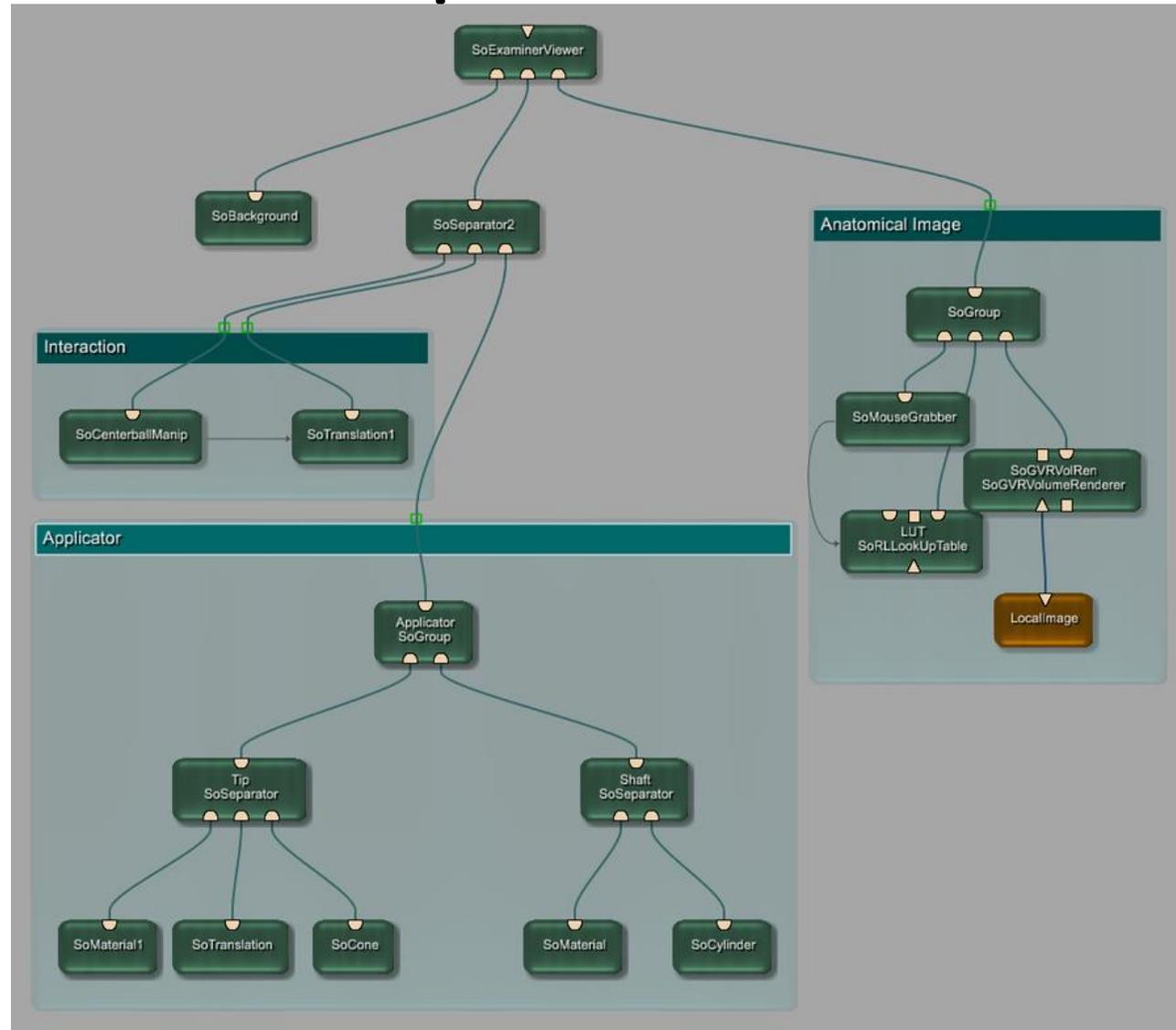
GigaVoxel Renderer

Copiando os Módulos de janela de View3D

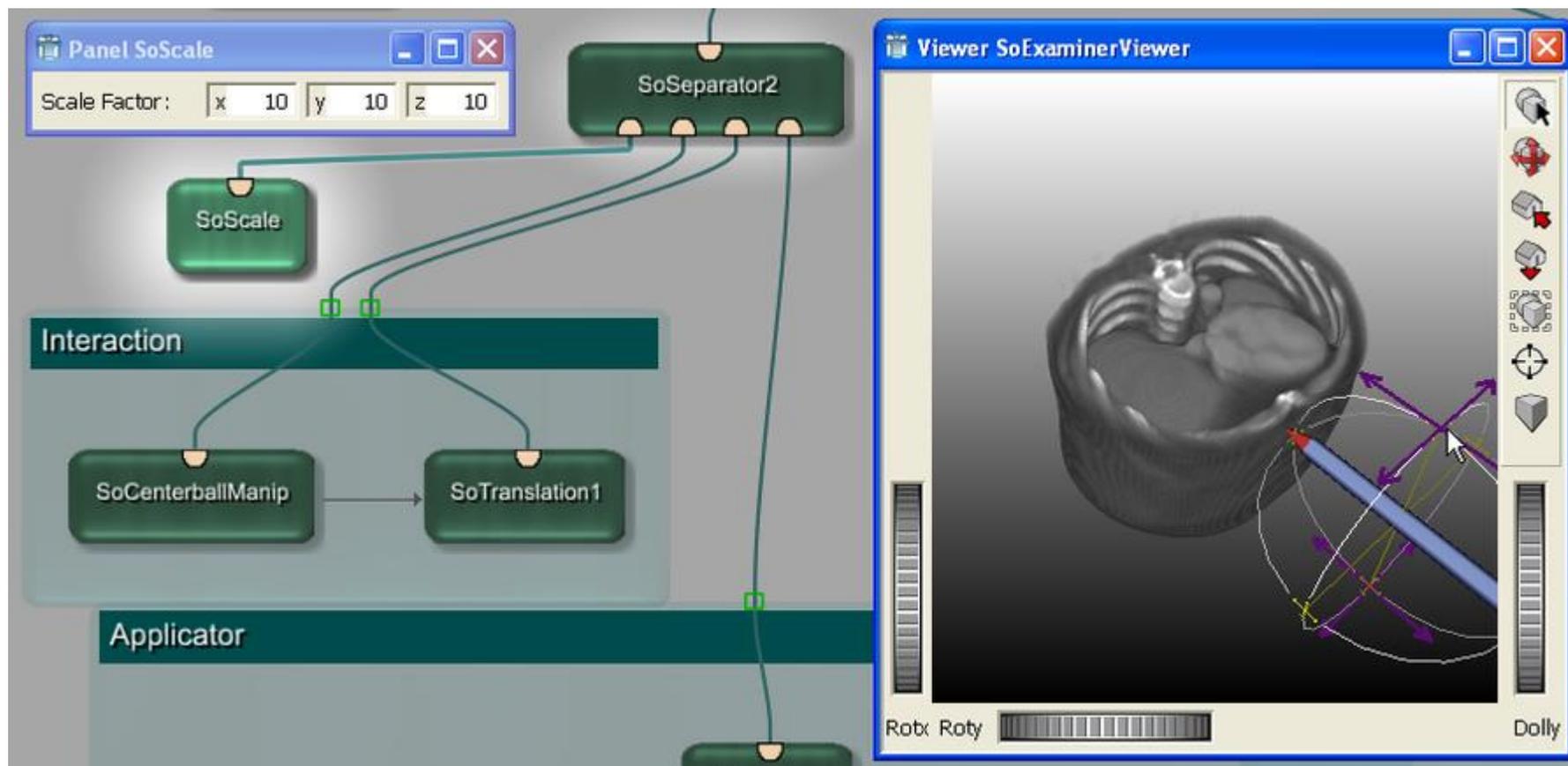


Podemos copiar do módulo *View3D* através da rede interna

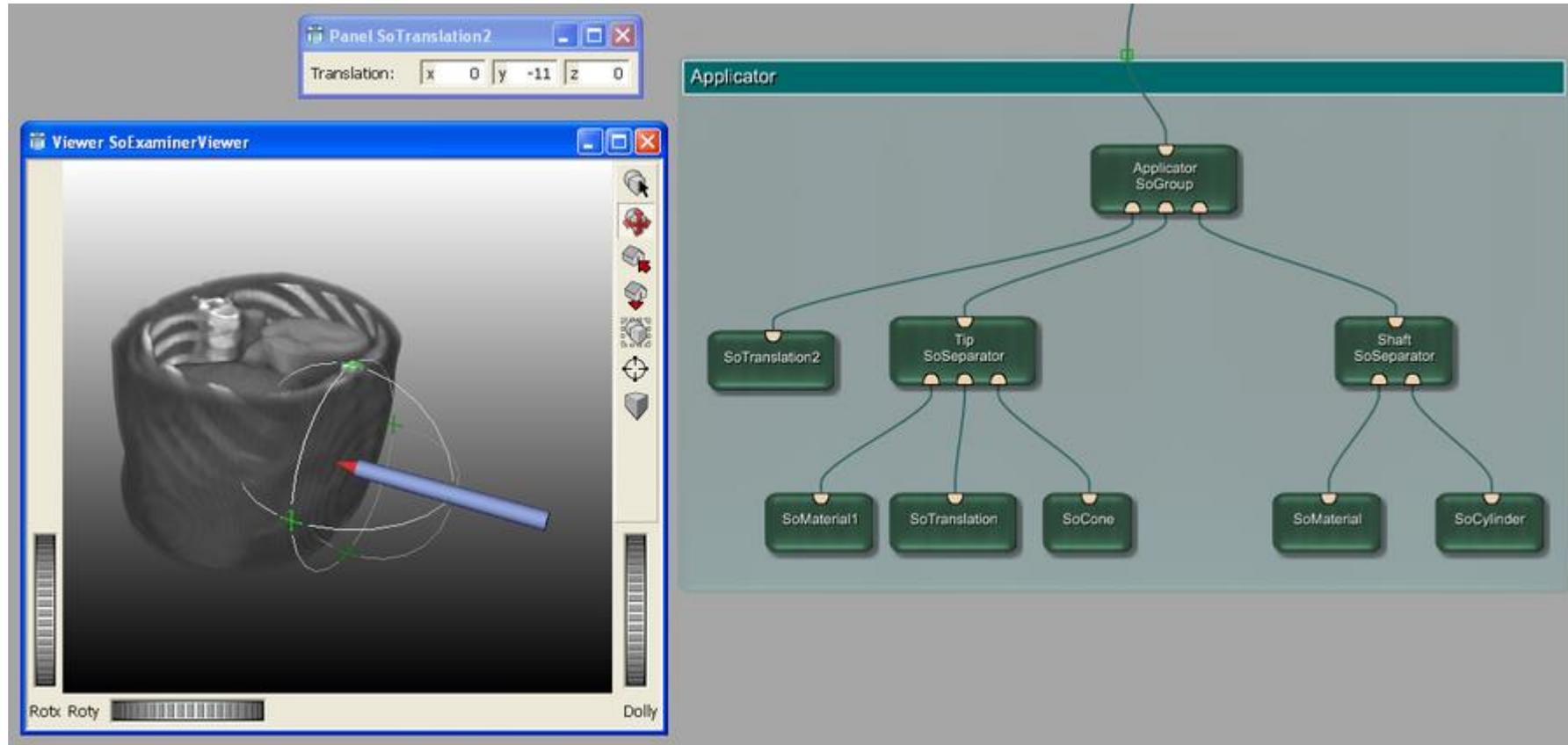
Combinando os Grupos



Adicionando Applicator Scaling



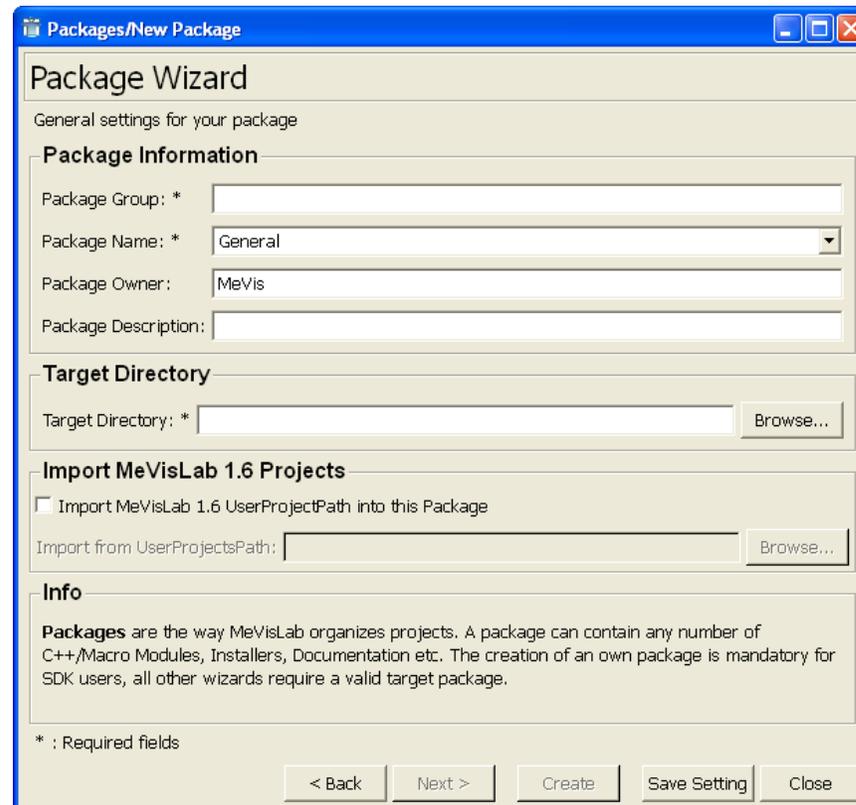
Ajuste



Adicionar um *SoTranslation*

Criação de um pacote

- Execute o Assistente de projeto (**File → Run Project Wizard**)
- Selecione **New Package**. O Assistente de Pacote é aberto.



The screenshot shows the 'Package Wizard' dialog box with the following sections:

- Package Information**:
 - Package Group: *
 - Package Name: * (dropdown menu showing 'General')
 - Package Owner: MeVis
 - Package Description:
- Target Directory**:
 - Target Directory: * (text field with 'Browse...' button)
- Import MeVisLab 1.6 Projects**:
 - Import MeVisLab 1.6 UserProjectPath into this Package
 - Import from UserProjectsPath: (text field with 'Browse...' button)
- Info**:
 - Text: **Packages** are the way MeVisLab organizes projects. A package can contain any number of C++/Macro Modules, Installers, Documentation etc. The creation of an own package is mandatory for SDK users, all other wizards require a valid target package.

* : Required fields

Buttons at the bottom: < Back, Next >, Create, Save Setting, Close

Introdução aos módulos de macro

- Módulos de macros são implementados por meio da MeVisLab Definition Language (MDL).
- Um módulo de macro se comporta como qualquer outro módulo no MeVisLab.
- Como qualquer outro módulo, um módulo de macro deve ser declarado dentro do banco de dados do MeVisLab em um arquivo de definição de módulo (*.def) localizado no *user package path*.

Introdução aos módulos de macro

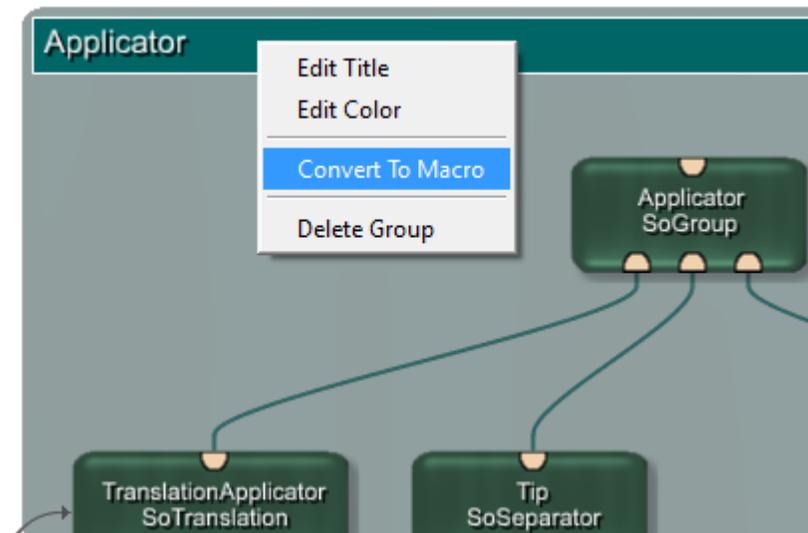
- Para implementação de um módulo MDL, que é a definição de interface (campos ENTRADA, SAÍDA e parâmetros), e também a sua definição GUI, geralmente são escritos num arquivo *.script. O script por ser construído *.py ou *.js. E os arquivos precisam ser incluídos no *.script de definição do módulo.

Módulos de macro

- Encapsulam outros módulos (imagens, processamento, etc.).
- Sua funcionalidade é definida pelo seus campos de entradas, saídas e seus parâmetros. (geralmente utiliza as entradas e saídas dos módulos que foram utilizados para criação da macro).
- Pode ser criado uma macro com módulos ou com outras macros.
- Facilidade para executar uma rotina frequentemente utilizada.
- Torna-se uma interface compacta na construção da rede.

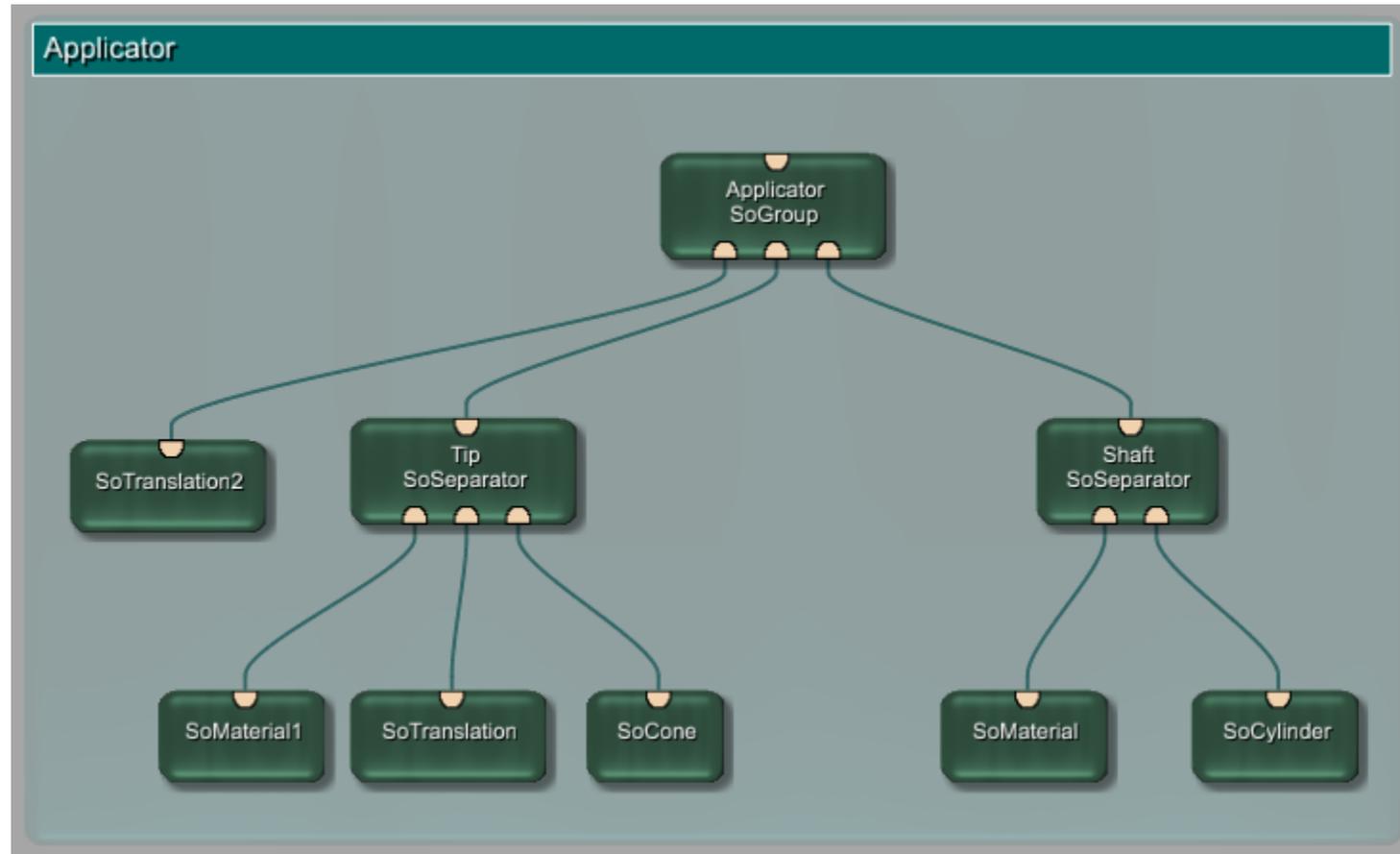
Macros locais

- Módulos de macro também pode ser definida localmente para um determinado caminho de documentos de rede, chamados de “Local Macro Modules”



Criando uma Macro

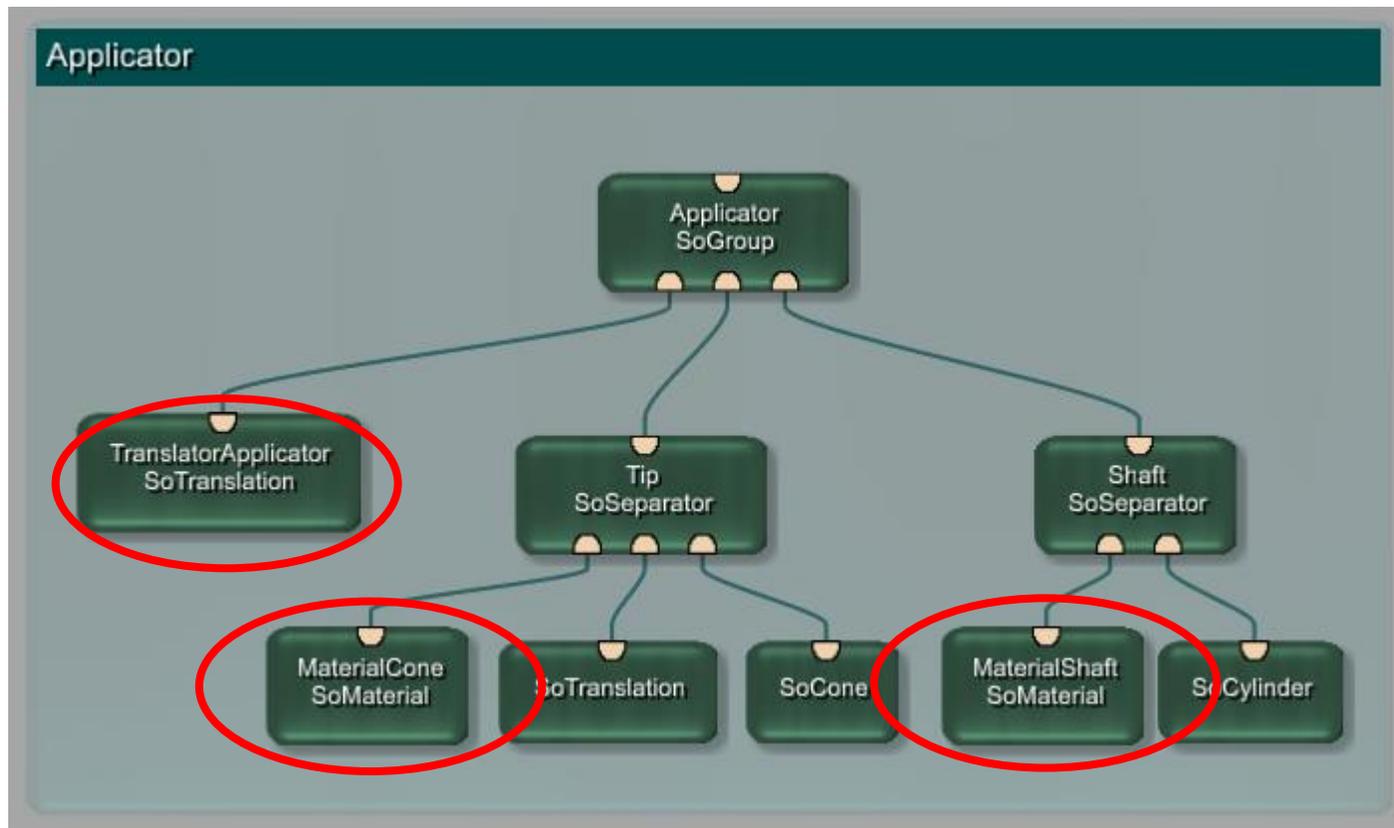
- Utilizaremos o aplicador para criar a macro



Criando uma Macro

- Para a criação de uma macro local, a rede precisa ser salva.

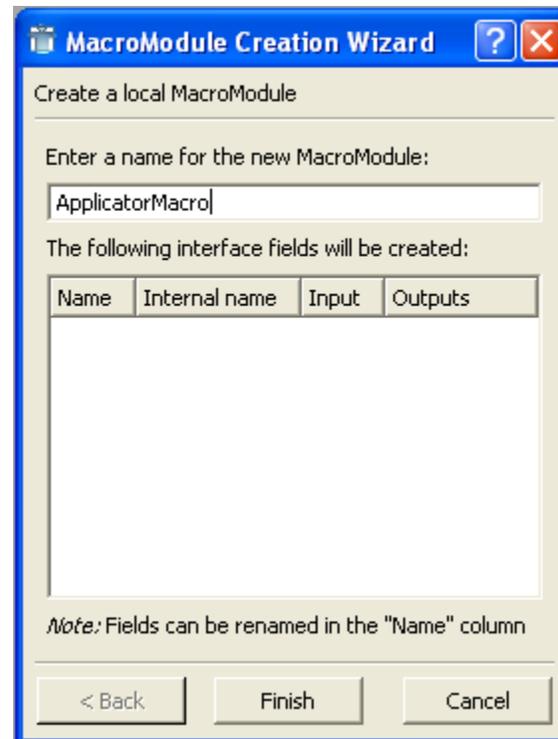
Editar nomes dos módulos



botão direito do mouse no módulo e selecione **Edit Instance Name**

Criando uma Macro

- Selecione todos os módulos com um duplo clique na barra de título do grupo **Applicator** e selecione **File** → **Create Local Macro**.



Um diálogo para macro local é aberto.

Propriedades do módulo

- **Name:** O nome do módulo tem que ser exclusivo no banco de dados do MeVisLab.
- **Author:** A entrada autor é obrigatória e será utilizado em pesquisas módulo.
- **Comment:** Insira uma breve descrição para o módulo. A entrada de comentário é obrigatória.
- **Keywords:** Termos para facilitar a busca.
- **See also:** Módulos relacionados que possam ser de interesse.
- **Genre:** Define o lugar do módulo no menu *Module Browser*.
- **Add reference to example network:** Cada módulo deve ter uma rede de exemplo, para explicar a sua função e uso em uma aplicação. Pode ser adicionado uma rede no arquivo ExampleModuleName.mlab.
- **Project:** Onde o módulo será agrupado.
- **Alvo do pacote:** Selecione um pacote de destino a partir da lista, para este exemplo "Exemplo / General".

Propriedades do módulo

Modules (Scripting)/Macro Module

Module Properties

Enter the general properties of the module.

General Module Properties

Name: * Author: *

Comment:

Keywords:

See Also:

Genre: Add reference to example network

Select Target Package

Package: *

Project Properties

Project: * Prefix:

Include project files

* : Required fields

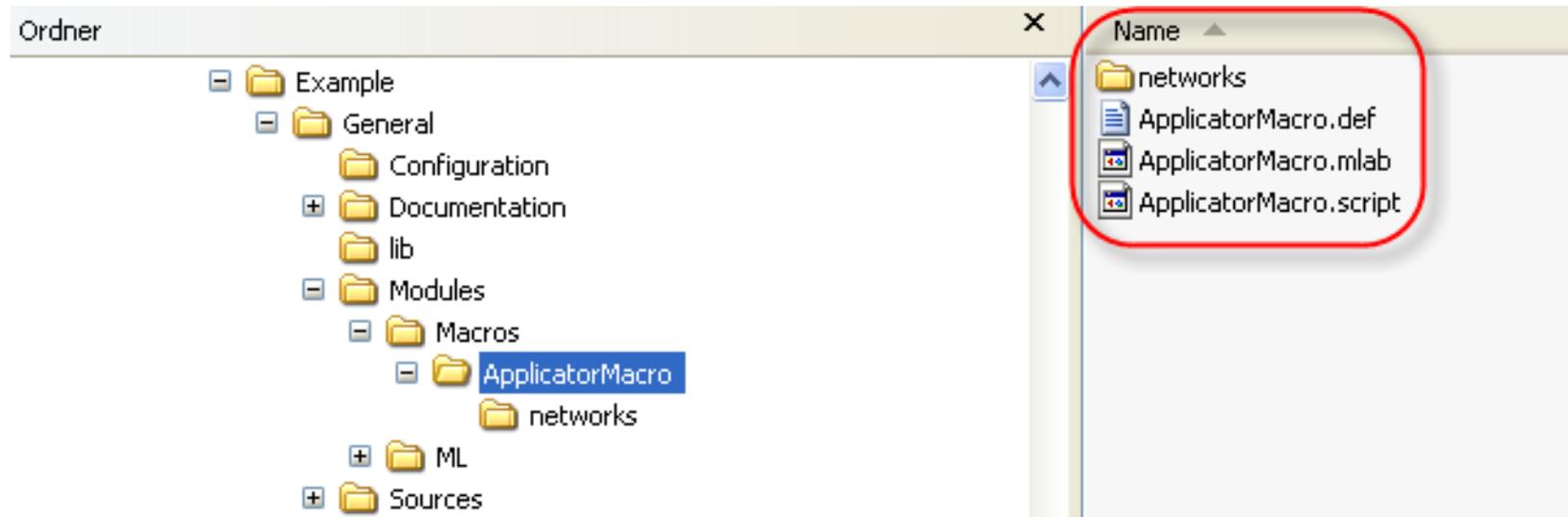
< Back Next > Create Save Setting Close

Macro Module Wizard

Really remove original local macro files?

novos arquivos da Macro

- .def - arquivo de definição de módulo, para registrar o módulo (s) ao banco de dados do MeVisLab.
- .mlab - de arquivos de rede que inclui os módulos e suas definições.
- .script - arquivo de script de MDL e de scripts (Python ou JavaScript).

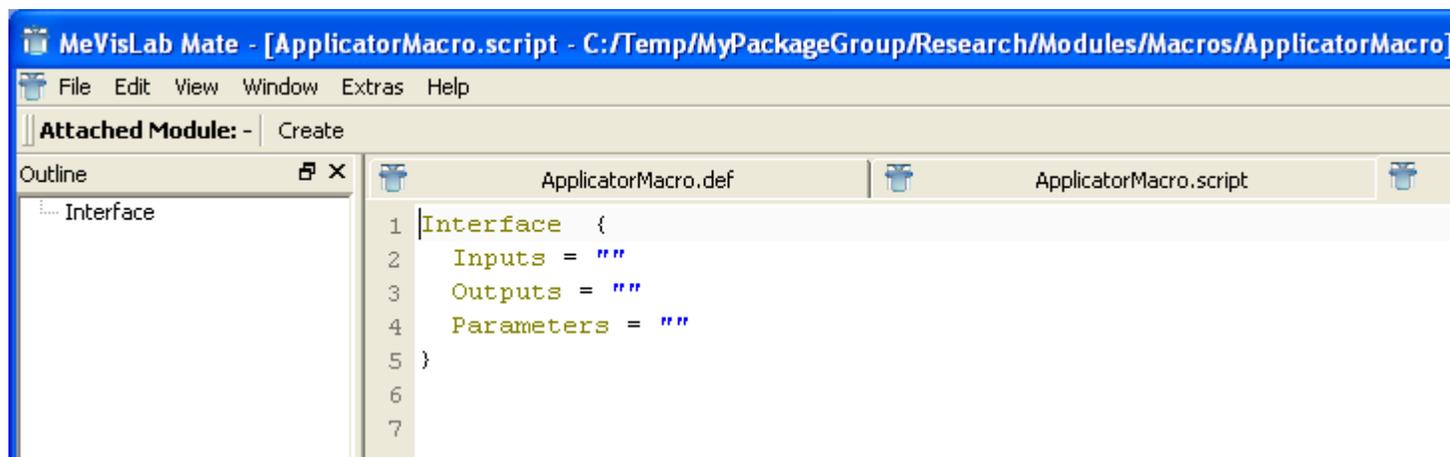


ApplicatorMacro como Módulo de Macro



Adicionando os parâmetros

- Até o momento, o módulo macro não tem pontos de interação. Portanto, a entrada/saída, os parâmetros/campos e o script precisa ser adicionado.



The screenshot shows the MeVisLab Mate software interface. The title bar reads "MeVisLab Mate - [ApplicatorMacro.script - C:/Temp/MyPackageGroup/Research/Modules/Macros/ApplicatorMacro]". The menu bar includes "File", "Edit", "View", "Window", "Extras", and "Help". Below the menu bar is a toolbar with "Attached Module: -" and a "Create" button. The main workspace is divided into two panes. The left pane, titled "Outline", shows a tree view with "Interface" expanded. The right pane, titled "ApplicatorMacro.def", shows the following code:

```
1 Interface {
2   Inputs = ""
3   Outputs = ""
4   Parameters = ""
5 }
6
7
```

botão direito do mouse em ApplicatorMacro → Related Files → ApplicatorMacro.script

Adicionando os parâmetros macro e Painel

- Interface: define as entradas e saídas de conexões de dados para a macro. No nosso caso, a macro não tem entradas de outros módulos, mas uma saída do Inventor.
- Commands: define o arquivo de script a ser executado sobre a atividade dos campos definidos.
- Window: Cria um painel para definir os parâmetros. No nosso caso, o comprimento e diâmetro. Esta é uma entrada opcional. Se não definido, apenas o painel automático está disponível.

Adicionando os parâmetros da macro

- Para a saída, podemos utilizar a saída do módulo *SoGroup* no módulo denominado *Applicator*. As linhas a seguir irão resultar em um campo de saída.

```
Interface {
  Inputs = ""
  Outputs {
    Field Scene {
      internalName = "Applicator.self"
    }
  }
  Parameters = ""
}
```

Aplicando alterações

- Clicando com o botão direito do mouse no módulo e selecionando **Reload Definition.**
- O módulo agora mostra um conector de saída.



Adicionando os parâmetros da macro

- Como próximo passo, vamos definir os parâmetros para a nossa interface. Neste exemplo, queremos ter dois parâmetros:
- Length: este será o comprimento total do aplicador.
- Diameter: este será o diâmetro do aplicador.

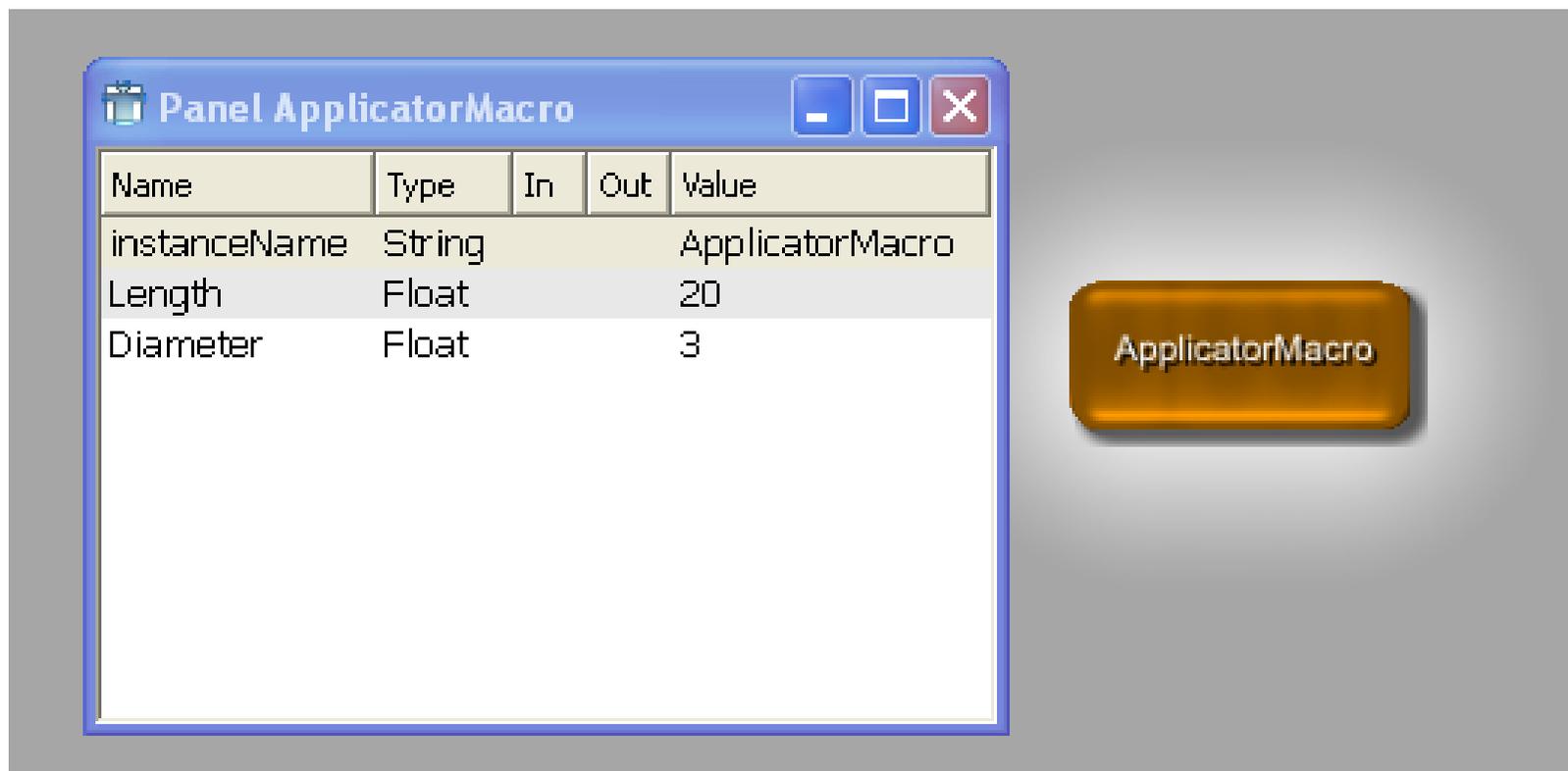
Adicionando os parâmetros macro e Painel

```
Interface {  
  Inputs = ""  
  Outputs {  
    Field Scene { internalName = "Applicator.self" }  
  }  
}
```

```
Parameters {  
  Field length {  
    type = float  
    value = 20  
    min = 1  
    max = 50  
  }  
  Field diameter {  
    type = float  
    value = 3  
    min = 0.1  
    max = 10  
  }  
}  
}
```

Mais uma vez, salvar o script e recarregar.

Automatic Panel do módulo



Painel do Módulo ApplicatorMacro

- Em princípio, isso seria o suficiente para introduzir os valores. No entanto, geralmente um painel mais amigável deve ser oferecido.
- Utilizar apenas parâmetros mais usados.
- Para criar um painel para os dois parâmetros, a nova seção *Window* é acrescentado no fim do script.

```
Interface {  
  Inputs = ""  
  Outputs {  
    Field Scene { internalName = "Applicator.self" }  
  }  
}
```

```
Parameters {  
  Field length {  
    type = float  
    value = 20  
    min = 1  
    max = 50  
  }  
  Field diameter {  
    type = float  
    value = 3  
    min = 0.1  
    max = 10  
  }  
}
```

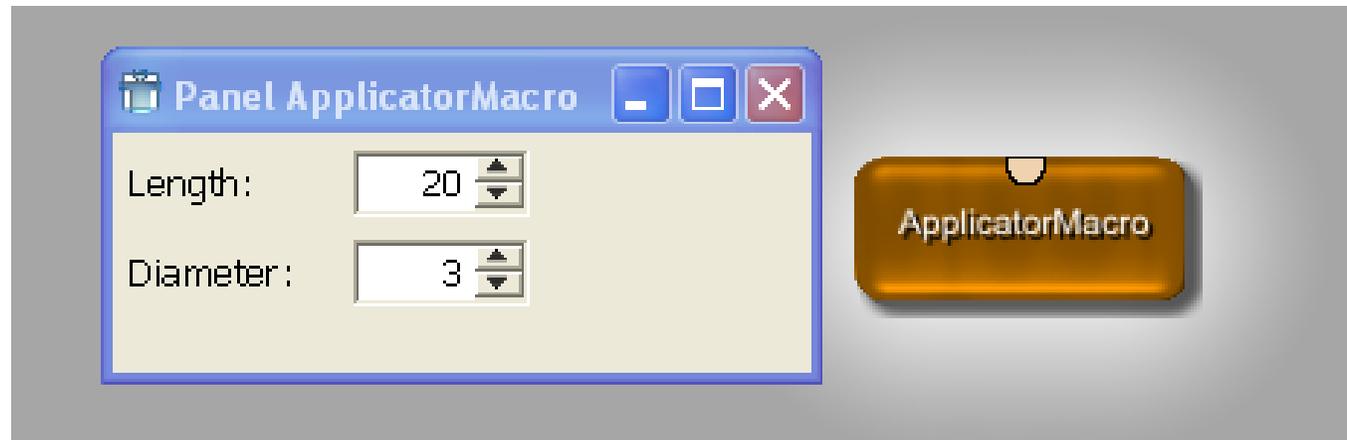
```
}
```

```
Commands {
```

```
}
```

```
Window {  
  Category {  
    Field length { step = 1 }  
    Field diameter { step = 0.1 }  
  }  
}
```

Painel do Módulo ApplicatorMacro



Interação dos parâmetros com a macro

- Todos os parâmetros são definidos e o painel está pronto para inserir valores no entanto, ainda não tem qualquer interação.
- Deveremos implementar a função *Command* (adicionar um script *Python*)
- A fonte será um arquivo local que iremos adicionar manualmente, com o nome **ApplicatorMacro.py** por convenção.
- Para fazer a interação com o script precisamos do comando **FieldListener**, um para o comprimento e outro para o diâmetro.

Commands

```
Commands {  
  source = $(LOCAL)/ApplicatorMacro.py  
  
  FieldListener length { command = AdjustLength }  
  FieldListener diameter { command = AdjustDiameter }  
}
```

mensagens de erros aparecerão

Salve o script e recarregar.

Programação do Python Script

- Se ainda não existente, crie o arquivo Python.
- **File → New** na barra de menu do MATE, salvar como **ApplicatorMacro.py** na mesma pasta que os outros arquivos do módulo.
- O que precisamos, é uma linha para importar os módulos MeVis Python.

```
# MeVis module import  
from mevis import *
```

Programação do Python Script

- Em seguida, precisamos adicionar duas funções, uma para cada comando de script.

```
def AdjustLength():  
    return  
def AdjustDiameter():  
    return
```

Programação do Python Script

- O diâmetro é ajustado pelo campo *diameter* da seguinte forma:

```
def AdjustDiameter():  
    diameter = ctx.field ("diameter").value  
    return
```

Parâmetros para Diameter Setting

- Para se ter um efeito tanto no eixo quanto na ponta da agulha, o diâmetro de ambos deve ser definido com o valor do *diameter*. Nas propriedades de *SoCone* e *SoCylinder* mostra que ambos os módulos oferecem um parâmetro de raio.

The image shows a scene graph and two parameter panels. The scene graph has a root node 'Tip SoSeparator' and 'Shaft SoSeparator'. 'Tip SoSeparator' is connected to 'SoTranslation' and 'SoCone'. 'Shaft SoSeparator' is connected to 'MaterialShaft SoMaterial' and 'SoCylinder'. Below the scene graph are two panels: 'Panel SoCone' and 'Panel SoCylinder'. Both panels have a 'Parameters' tab and an 'Outputs' tab. The 'Parameters' tab in 'Panel SoCone' shows a table with columns 'Name', 'Type', 'In', 'Out', and 'Value'. The 'Parameters' tab in 'Panel SoCylinder' shows a table with columns 'Name', 'Type', 'In', 'Out', and 'Value'.

Name	Type	In	Out	Value
instanceName	String		SoCone	
parts	String		ALL	
bottomRadius	Float		1	
height	Float		3	

Name	Type	In	Out	Value
instanceName	String		SoCylinder	
parts	String		ALL	
radius	Float		1	
height	Float		20	

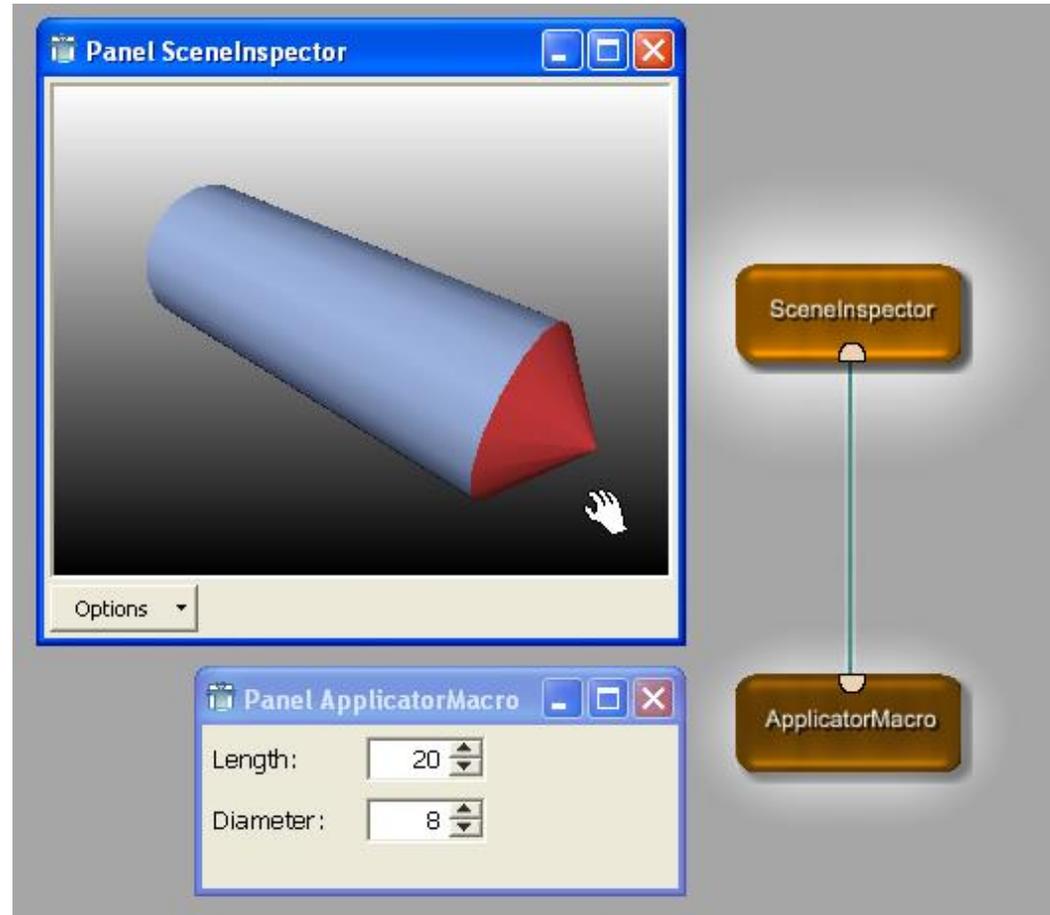
Parâmetros para Diameter Setting

- Estes parâmetros de raio precisam ser ajustados para *diameter*.
- O raio é a metade do diâmetro, então um fator de correção de 0,5 tem de ser adicionado para a equação de diâmetro.

```
def AdjustDiameter():  
    diameter = ctx.field("diameter").value * 0.5  
  
    ctx.field("SoCone.bottomRadius").value = diameter  
    ctx.field("SoCylinder.radius") .value = diameter  
  
    return
```

Testando o módulo

- Para testar se o diâmetro ajustando está funcionando, vamos adicionar um **SceneInspector** e conectar a sua entrada na saída do nosso módulo.



Length Setting

- Ajustar o comprimento é um pouco mais complicado. A variação de comprimento devem ter os seguintes efeitos:
 - Somente o eixo deve ser alargado, não a ponta.
 - O ajuste deve ser feito para o lado oposto da ponta, para que assim ele não se sobreponha sobre a ponta.

Length Setting

- Podemos definir um comprimento total, comprimento da ponta e comprimento do eixo. Eles podem ser calculados como se segue:

```
def AdjustLength():  
    overallLength = ctx.field("length").value  
    tipLength = ctx.field("SoCone.height").value  
  
    shaftLength = overallLength - tipLength  
    return
```

Length Setting

- O fator de tradução original para a ponta foi dada pela metade do comprimento do eixo ("10"), mais a metade do comprimento da ponta ("1.5"). Isso pode ser escrito de uma forma geral.

```
tipTranslation = shaftLength*0.5 + tipLength*0.5
```

- O shaftLength define a altura do cone *SoCylinder*

```
ctx.field ("SoCylinder.height").valor = shaftLength
```

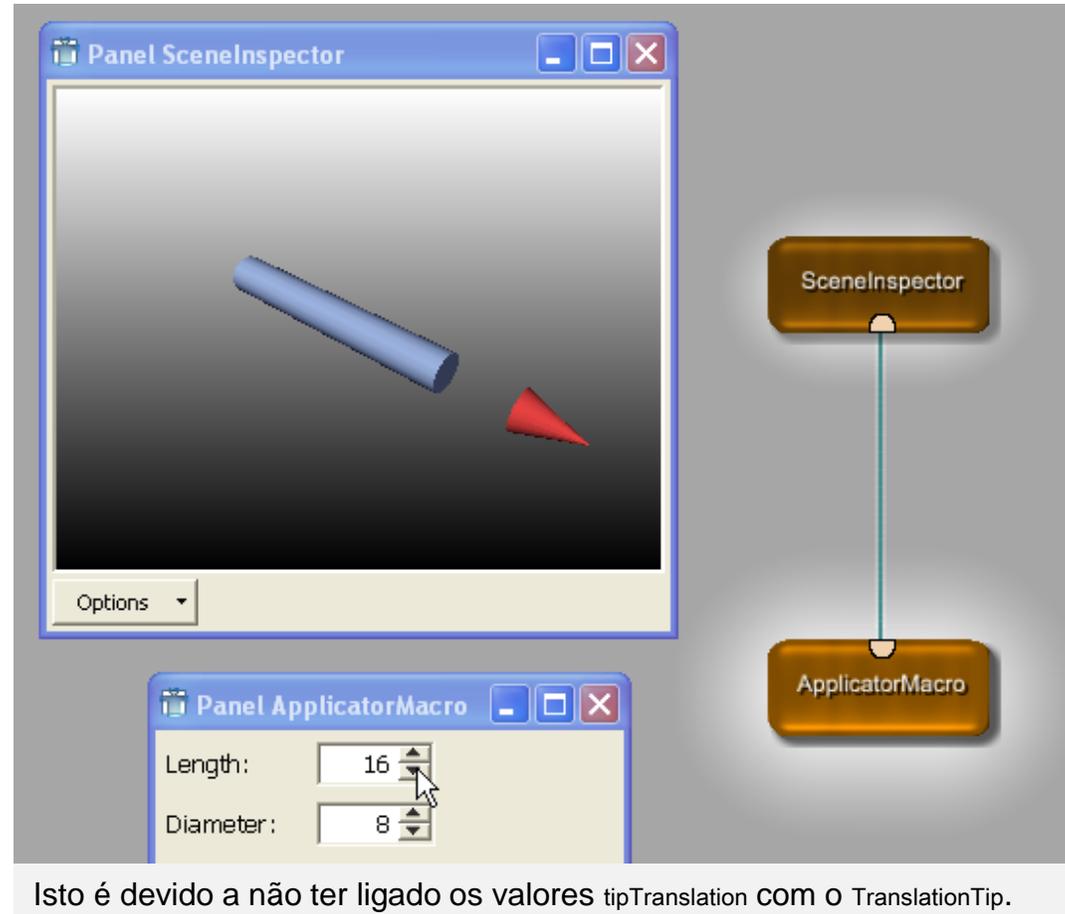
Length Setting

- As linhas de código que ajustam o comprimento:

```
def AdjustLength():  
  
    overallLength = ctx.field("length").value  
    tipLength = ctx.field("SoCone.height").value  
  
    shaftLength = overallLength - tipLength  
    tipTranslation = shaftLength*0.5 + tipLength*0.5  
  
    ctx.field ("SoCylinder.height").value = shaftLength  
  
    return
```

Comportamento do ApplicatorMacro

- Adicione este código ao script Python, salvar e recarregar.



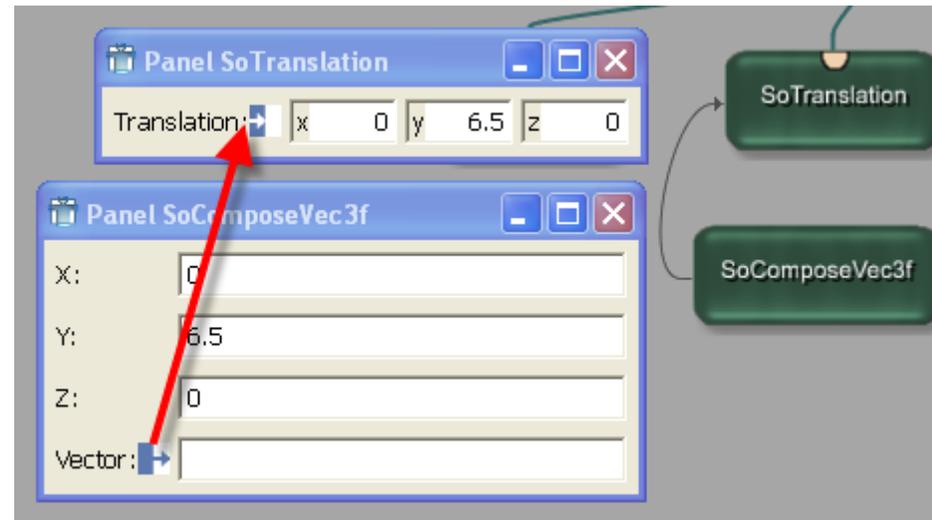
Resolvendo

- Para resolver este problema, adicione o módulo **SoComposeVec3f** à rede interna da macro e atribua a sua tradução no parâmetro `y` para o valor `tipTranslation` calculado.

```
ctx.field ("SoComposeVec3f.y"). valor = tipTranslation
```

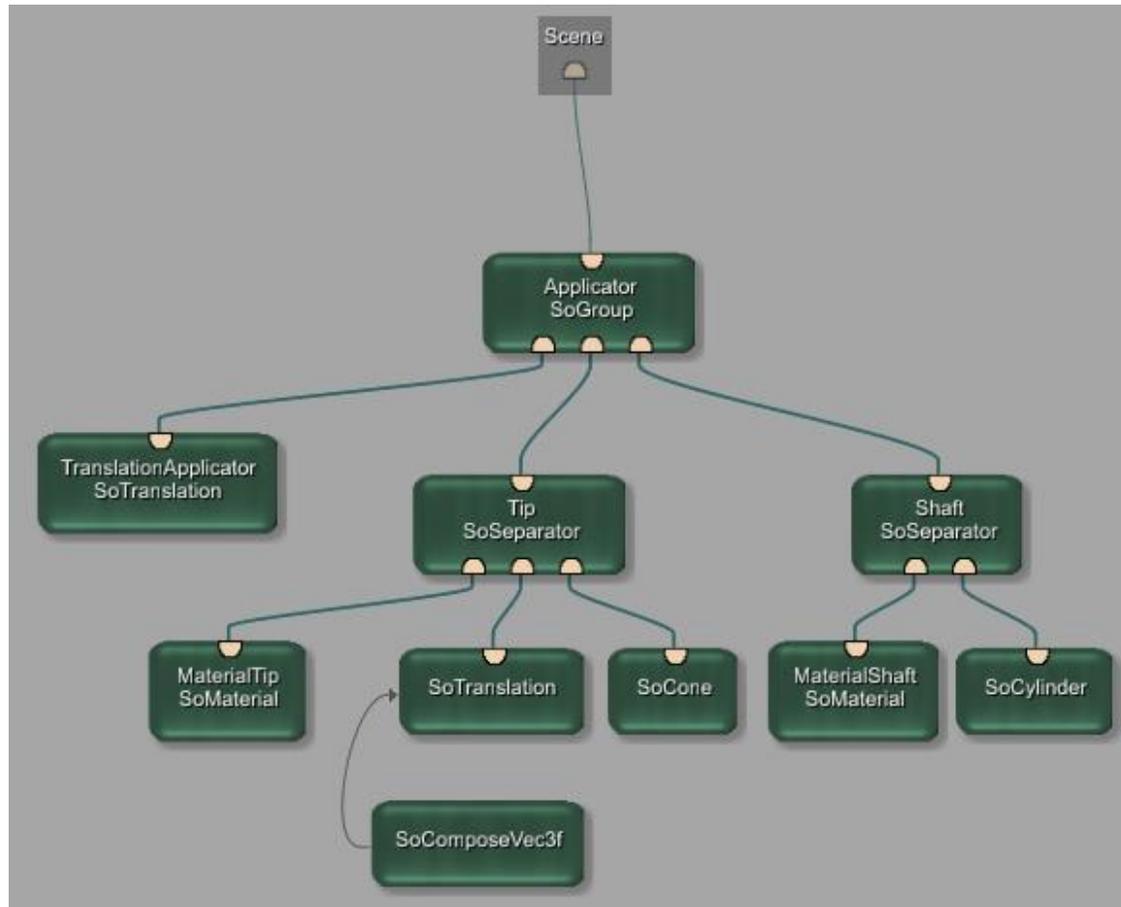
Resolvendo

- Num último passo, este *translation* precisa ser ligado à ponta do módulo **SoTranslation** através de uma ligação de parâmetro.



ApplicatorMacro completa

- Aqui a rede e script Python completa do exemplo ApplicatorMacro:



```
# -----  
## This file implements scripting functions for the LocalFileName module  
#  
# \file  ApplicatorMacro.py  
# \author Luiz  
# \date  08/2014  
#  
# -----
```

```
# MeVis module import  
from mevis import *
```

```
def AdjustDiameter():  
    diameter = ctx.field("diameter").value * 0.5  
  
    ctx.field("SoCone.bottomRadius").value = diameter  
    ctx.field("SoCylinder.radius") .value = diameter  
    return
```

```
def AdjustLength():  
    overallLength = ctx.field("length").value  
    tipLength     = ctx.field("SoCone.height").value  
  
    shaftLength  = overallLength - tipLength  
    tipTranslation = shaftLength*0.5 + tipLength*0.5  
  
    ctx.field("SoCylinder.height").value = shaftLength  
    ctx.field("SoComposeVec3f.y") .value = tipTranslation  
    return
```

Addition: Shifting the Whole Tip

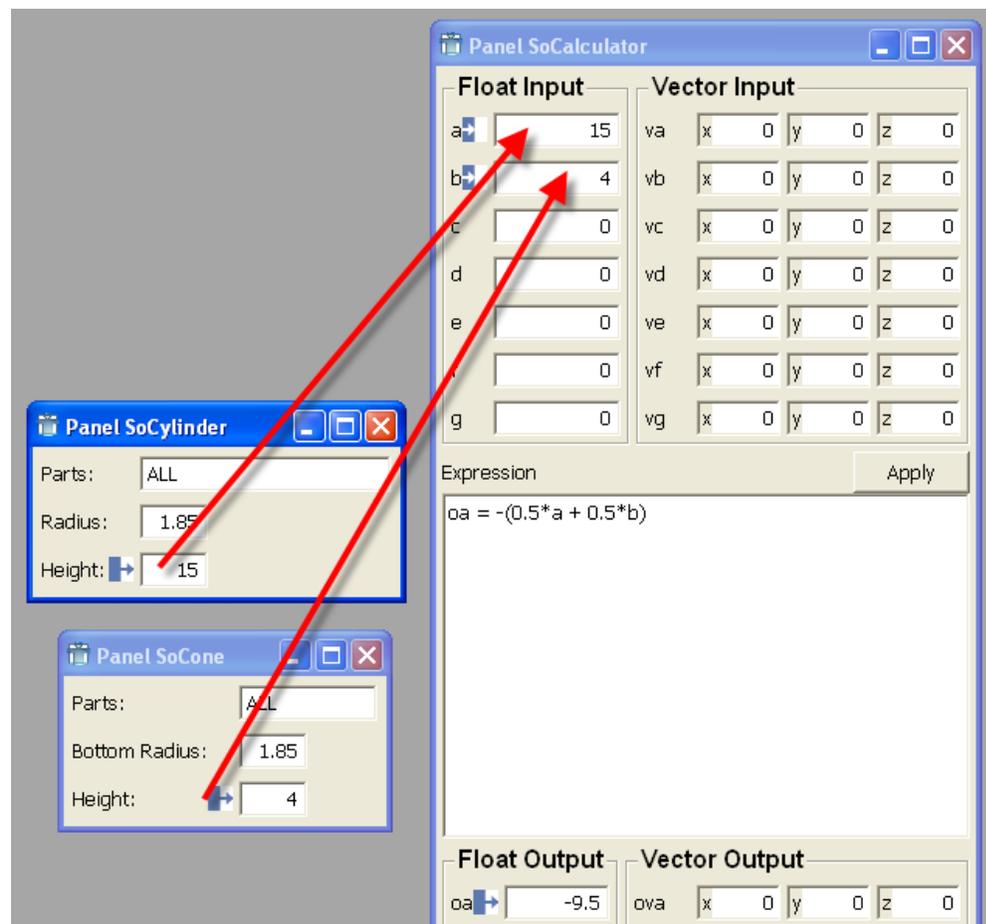
- No exemplo acima, a alteração no comprimento irá ser traduzido para uma mudança global com o centro de rotação, como centro global. No entanto, talvez seja preferível manter a ponta no lugar e mudar o comprimento do eixo em outra direção.
- Basicamente, este é o mesmo problema que do cálculo do comprimento que fizemos no script em Python. No entanto, em vez de calculá-lo no script, também podemos usar um módulo para o cálculo.

Addition: Shifting the Whole Tip

- Para isso, os seguintes módulos devem ser adicionados:
- **SoCalculator**: Para o cálculo do comprimento do eixo.
- **SoComposeVec3f**: Para aplicar e tradução do valor float para o vetor do **TranslationApplicator**.

Alimentando o Módulo SoCalculator

- O módulo **SoCalculator** oferece uma entrada e saída de valores float e vetores.



Alimentando o Módulo SoCalculator

- Para o cálculo usamos os valores de altura do cone e comprimento do eixo, usando o módulo *SoCalculator*, configurando as conexões de parâmetros.
 - Conecte SoCylinder.height para SoCalculator.a
 - Conecte SoCone.height para SoCalculator.b
 - Introduzir o cálculo: $oa = - (0.5*a+0.5*b)$

Alimentando o Módulo SoCalculator

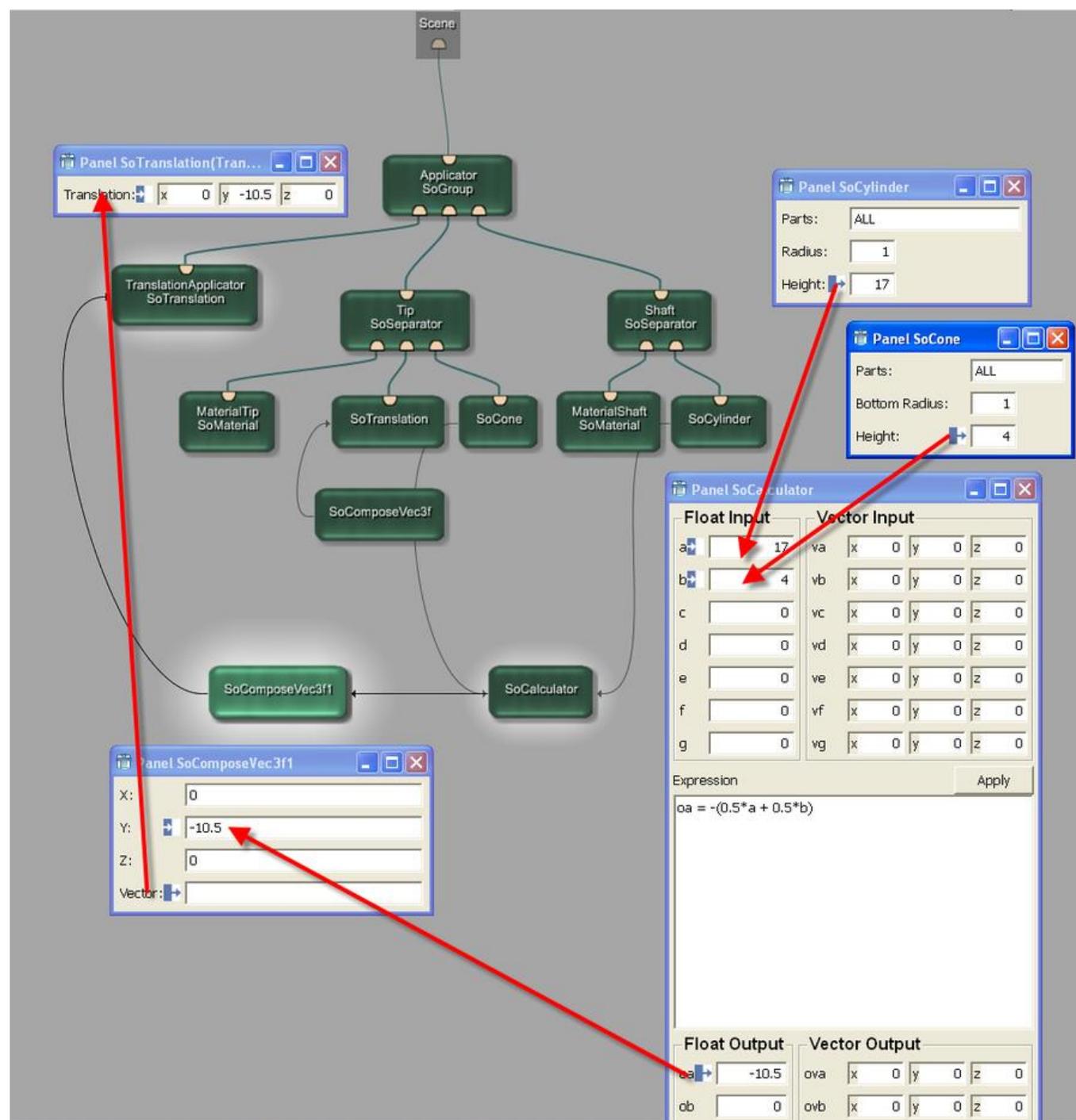
The image shows a software interface with three panels:

- Panel SoCylinder:** Parts: ALL, Radius: 1.85, Height: 15.
- Panel SoCone:** Parts: ALL, Bottom Radius: 1.85, Height: 4.
- Panel SoCalculator:**
 - Float Input:** a: 15, b: 4, c: 0, d: 0, e: 0, f: 0, g: 0.
 - Vector Input:** va, vb, vc, vd, ve, vf, vg (each with x, y, z components set to 0).
 - Expression:** $oa = -(0.5*a + 0.5*b)$ (circled in red).
 - Float Output:** oa: -9.5.
 - Vector Output:** ova (x, y, z components set to 0).

Red arrows point from the 'Height' field in SoCylinder and the 'Height' field in SoCone to the 'a' and 'b' input fields in SoCalculator, respectively.

Alimentando o Módulo SoCalculator

- Para aplicar a nova tradução, precisamos de outro módulo SoComposeVec3f. Ele permite a conversão do valor float de y em um vetor na direção y . Para isso, tem de receber a saída do SoCalculator e proporcionar a entrada para o SoTranslation.
 - Conecte SoCalculator.oa para SoComposeVec3f1.y
 - Conecte SoComposeVec3f1.vector para SoTranslation.translation



Este é o fim