VOREEN

VOLUME RENDERING ENGINE

Timo Ropinski, Jörg-Stefan Praßni

V 2.6

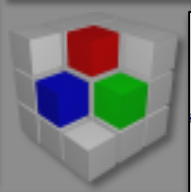
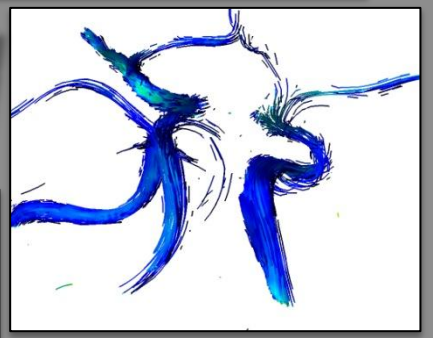
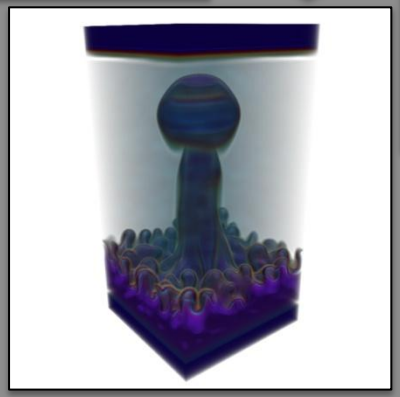
These slides are available at
www.voreen.org/420-Tutorial-Slides.html

voreen
volume rendering engine

Outline

- About Voreen
- Obtaining Voreen
- Conceptual overview
- Project structure
- Extending Voreen
- Visual debugging
- Customizing applications
- Additional features
- Outlook
- Where to go from here



About Voreen

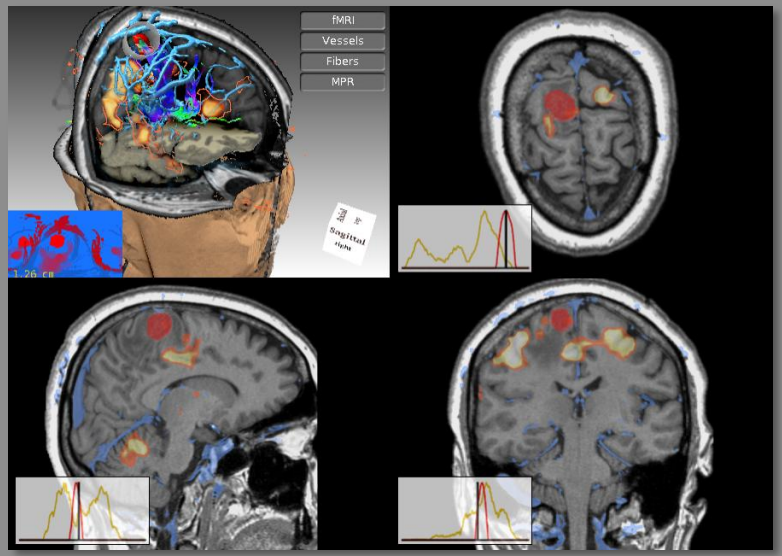
What it is, and what it isn‘t

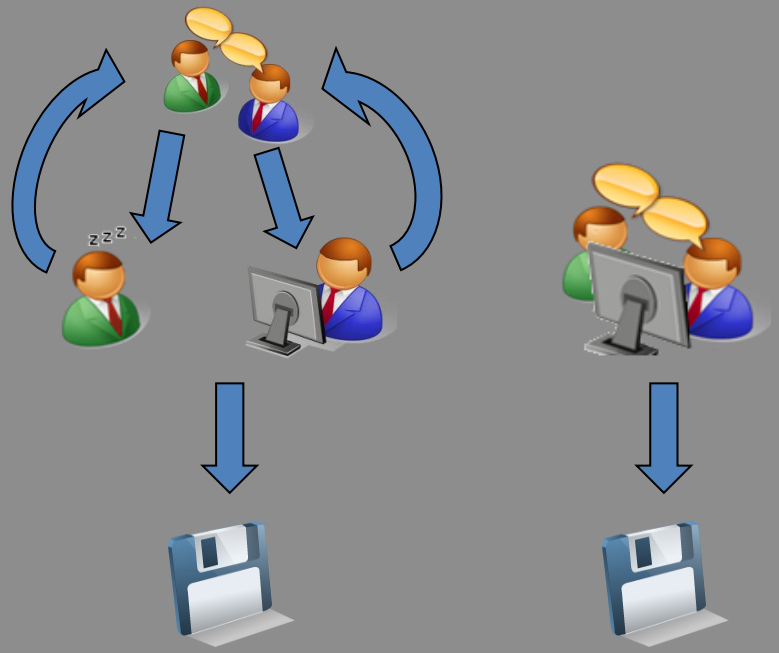
About Voreen

- Volume visualization research platform with a focus on rendering/visualization, some preprocessing capabilities
  - Processors can be reused as functional entities by exploiting the data-flow metaphor
  - Integrates not only volume data (e.g., fibers, geometry, flow data…)
  - Open source
  - Exploits GLSL, OpenCL, CUDA
  - Platform independent (Windows, Linux & Mac OS X)

Collaborating with Domain Experts

- Insightful visualizations can only be generated collaboratively
- Challenging for domain experts and computer scientist

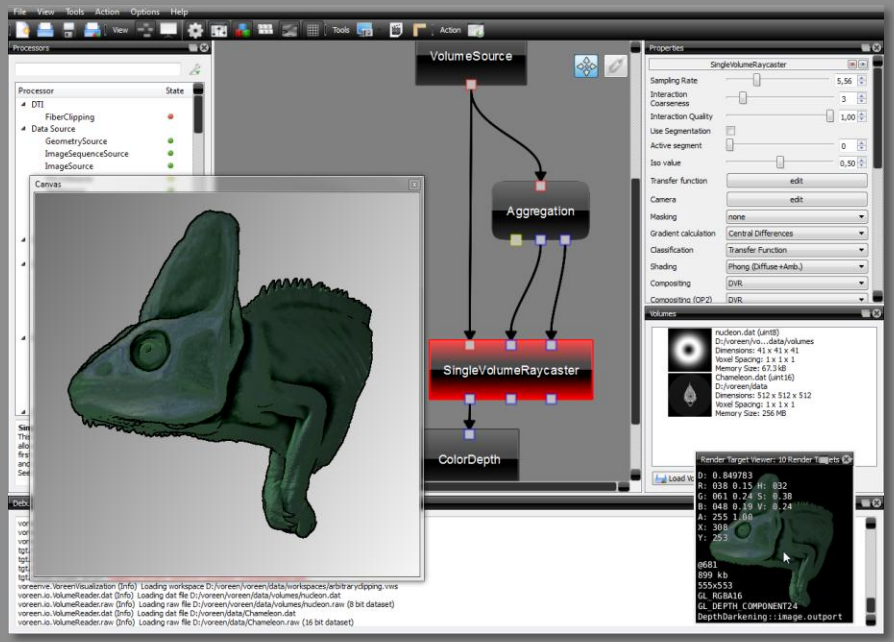




The shown dataset is courtesy of Prof. B. Terwey, Klinikum Mitte, Bremen, Germany.

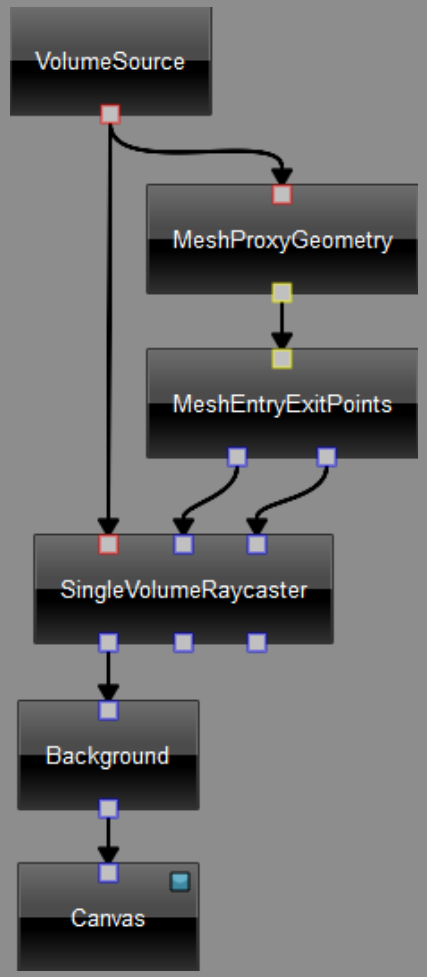
Multi-Level Visual Programming

- Exploit data-flow networks for visual prototyping of interactive visualizations
  - High level through abstraction
  - Reusability
- Grant access to low-level GPU features
  - Support shader programming
  - Allow optimization



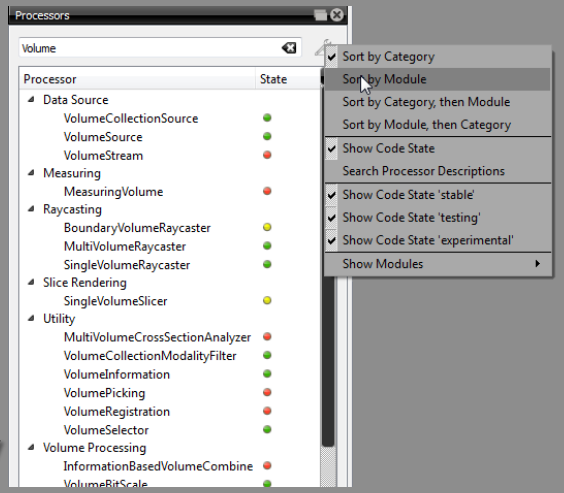
Voreen - Volume Rendering Engine

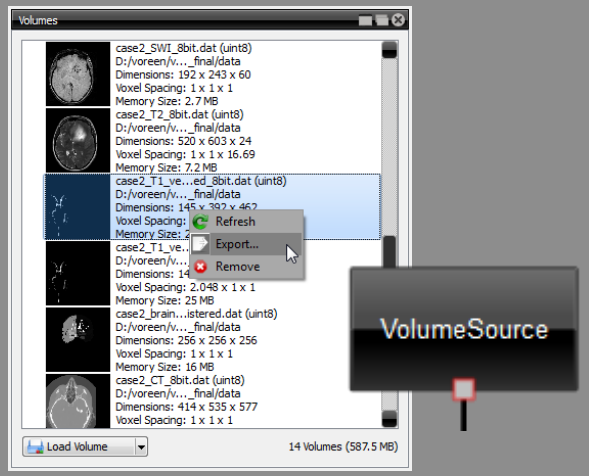
- Data-flow concept
  - Processors
  - Communication via ports
    - Color depicts type
      (e.g., volume, image, geometry, …)
  - Properties specify processor behavior
- Central evaluator
  - Determines evaluation order
  - Manages resources



Using Voreen

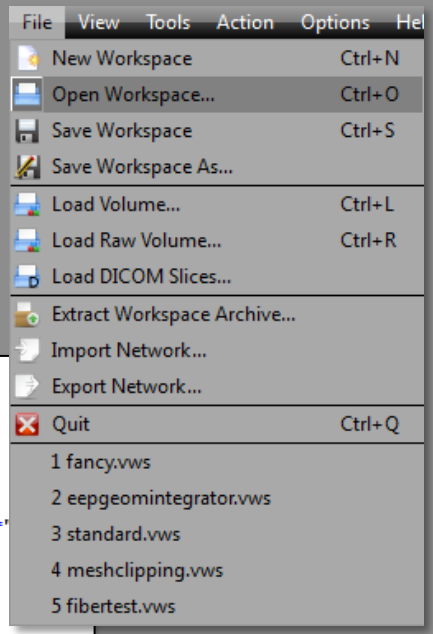
- Reusing processors
  - Processors are organized in processor list
  - List can be searched and sorted by type, module, etc..
  - State flag depicts processors stability (●=broken, ●=testing, ●=stable)
- Managing volume data
  - Volume data is stored in volume container
  - Brought into the network with *VolumeSource* processors





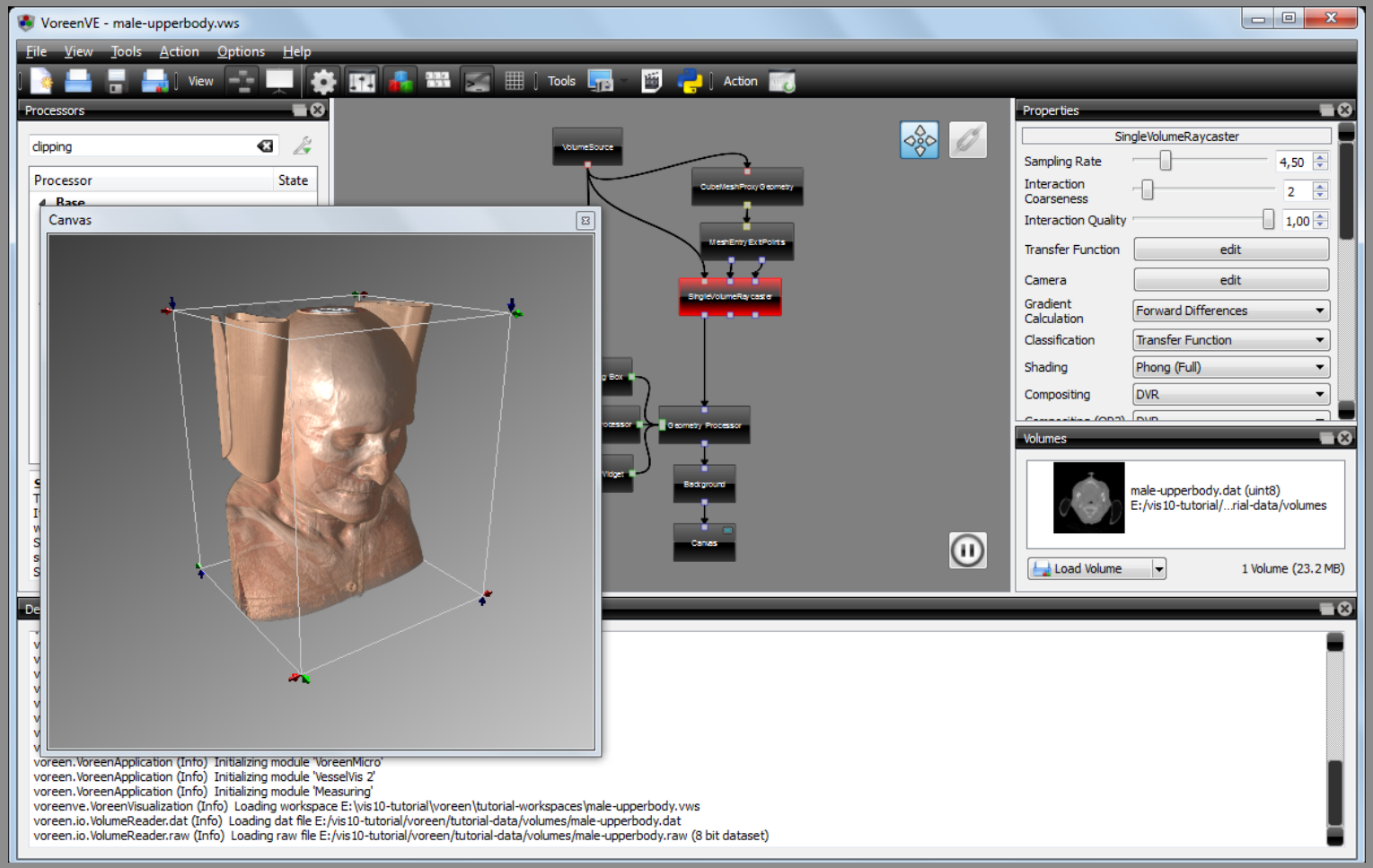
Workspaces

- The current session is serialized within the XML-based Voreen workspace format **`.vws`**
  - Network topology
  - Property states
  - Processor layout
  - Loaded volumes
  - …



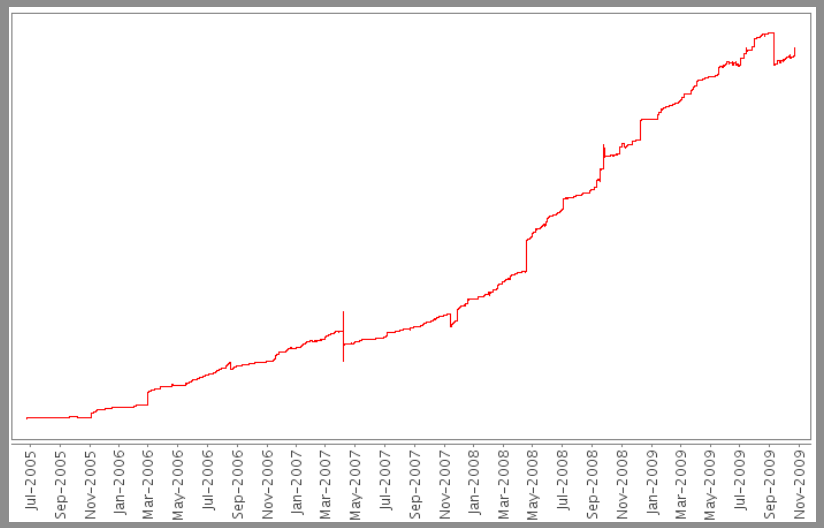

```
<Processor type="MeshEntryExitPoints" name="MeshEntryExitPoints" id="ref8">
    <MetaData>
        <MetaItem name="ProcessorGraphicsItem" type="PositionMetaData" x="-205" y="-174" />
    </MetaData>
    <Properties>
        <Property name="camera" adjustProjectionToViewport="true" id="ref16">
            <MetaData>
                <MetaItem name="EditorWindow" type="WindowStateMetaData" visible="false" x="-955" y="
            </MetaData>
            <position x="-3.16516089" y="1.88449895" z="2.34805989" />
            <focus x="-0.14060999" y="-0.205892" z="0.00218" />
            <upVector x="0.3551189" y="0.87701076" z="-0.3236399" />
        </Property>
        <Property name="filterJitterTexture" value="true" />
        <Property name="jitterEntryPoints" value="false" />
        <Property name="jitterStepLength" value="0.005" />
        <Property name="supportCameraInsideVolume" value="true" />
        <Property name="useFloatRenderTargets" value="false" />
    </Properties>
</Processor>
```

Demonstration



Voreen - Stats

- Developers: ~13 full- and part-time
- Lines of Code: 206.115+



Lines of Code

- Competitive Performance

Table 7: **Comparison between CUDA and fragment-shader ray-casting.** Voreen was adapted as the fragment-shader implementation.

Dataset	ds1	ds3	ds2	ds4
CUDA (FPS)	19.61	2.75	8.75	1.45
Fragment-shader (FPS)	22.56	3.08	10.76	1.83

We see that the Voreen fragment shader is 1.12x to 1.26x faster than our CUDA implementation, which is comparable to the observations made by Mensmann et al. We believe this is largely due to taking advantage of hardware rasterization to perform empty space skipping [15], which is not an option for our CUDA implementation. As the result, for datasets with many empty voxels (Table 3), such as ds2 and ds4, Voreen achieves speedups of 1.23x and 1.28x, respectively, over our CUDA implementation. For datasets with few empty voxels, such as ds1 and ds3, a significant portion of the execution time is spent in gradient computation. As a result, Voreen achieves smaller speedups of 1.12x and 1.15x, respectively. Overall, we see both CUDA and fragment-shader implementations of ray-casting are comparable in performance.

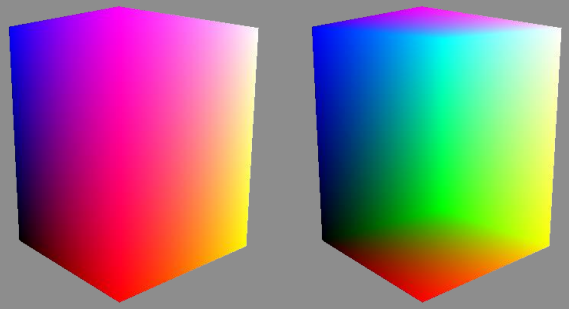
7 Conclusions

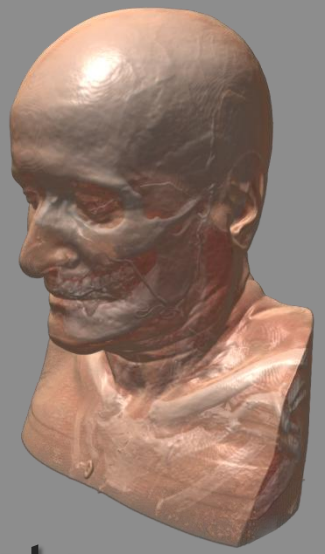
This paper maps and evaluates performance of volume rendering application on three modern parallel architectures: Intel Nehalem, Nvidia GTX280 and Intel Larrabee. Overall our parallel implementation of ray-casting delivers close to 5.8x speed-up on quad-core Nehalem over an optimized scalar baseline version running on a single core Harpertown. This enables us to render a large 750
dataset in 2.5 seconds. In comparison, we ach
on Nvidia GTX280 over the scalar basel
detailed performance simulation.
around 10x speed-up over sin
between 1.5x higher

[Smelyanskiy et al., IEEE Vis 2009]

Technical Aspects

- Main renderer:
  OpenGL/GLSL volume ray-casting





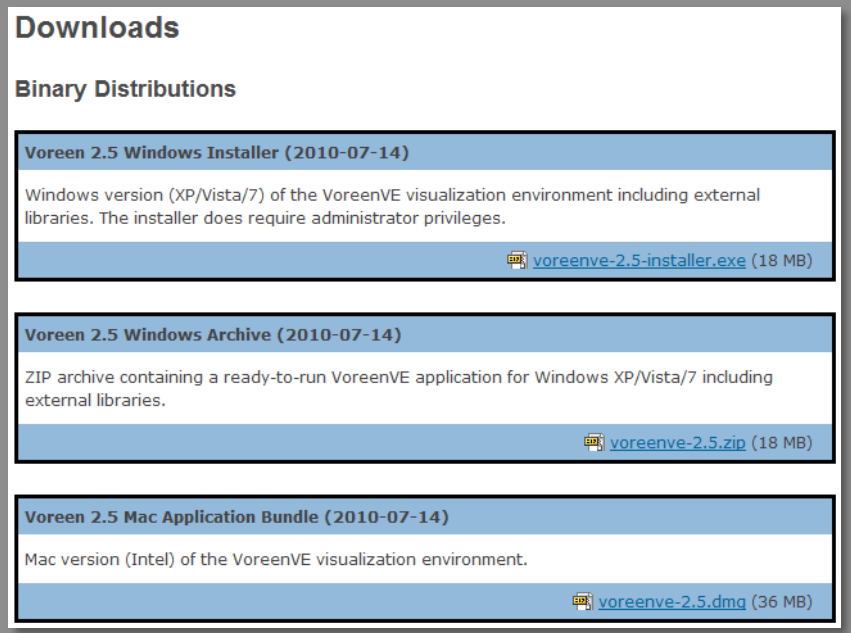
- Support for several volume file formats
  (RAW, DICOM, TIFF-stacks, PVM, Philips 3D US,
  NRRD, vevo, ...)
- Few external dependencies:
  GUI optional (Qt / GLUT / MFC)

Obtaining Voreen

Downloading prebuilds and accessing the SVN

Obtaining Voreen

- Binary distributions
  - Windows (ZIP or installer), Mac OS X (DMG)
- Source distribution
  - Windows, Mac OS X, Linux
  - ZIP source packages
- Public SVN access
  - protocol: *http*
  - server: *svn.voreen.org*
  - path: *public/voreen-snapshot*
  - e.g.:
    ```
    svn co http://svn.voreen.org/public/voreen-snapshot
    ```



[www.voreen.org]

Building Voreen

- Obtain and install Qt
- Using Microsoft Visual Studio
  1. Copy **`config-default.txt`** to **`config.txt`**
  2. Adapt **`qmake-default.bat`**:
     set **`qmake`** path (line 3), set Visual Studio version (line 6/7)
  3. Execute **`qmake-default.bat`**
  4. Add Voreen projects into one solution (File->Add->Existing project)
     `src/core/voreen_core.vcproj`
     `src/qt/voreen_qt.vcproj`
     `apps/voreenve/voreenve.vcproj`
  5. Set **`voreenve`** as startup project and set the dependencies
     (**`voreenve`** depends on **`voreen_core`** and **`voreen_qt`**)
  6. Copy necessary DLLs from **`voreen/ext`** to **`apps/voreenve`**
  7. Compile and run

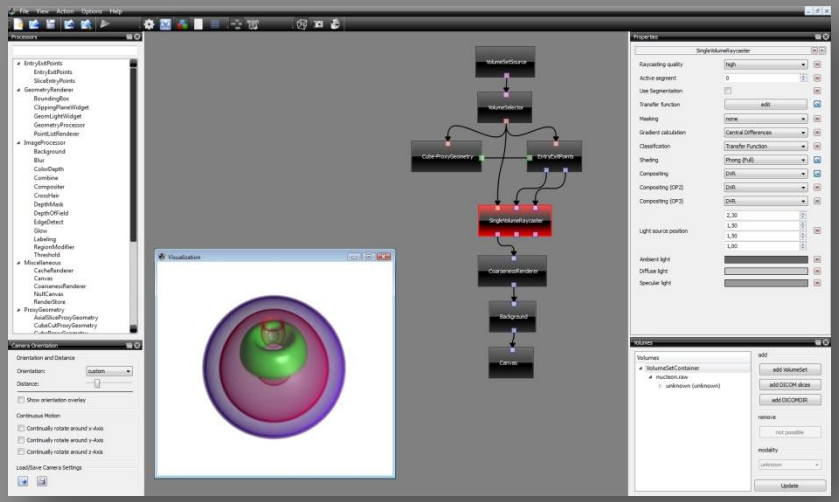
Detailed build instructions available at
www.voreen.org

Conceptual Overview

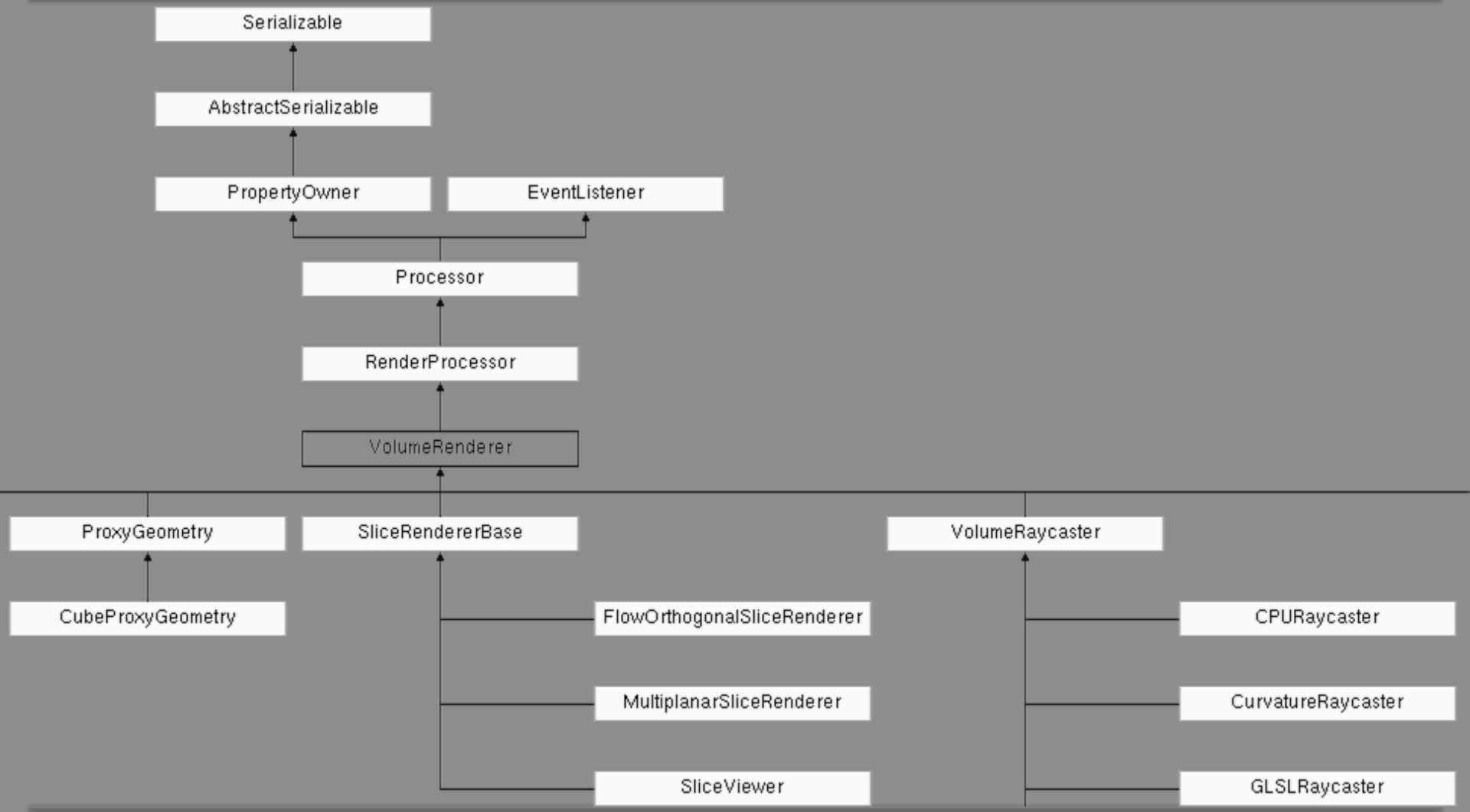
Processors, ports and properties

Processors, Ports and Properties

- Processors
  - Do all the work (`process()` method)
- Ports
  - Are used to transfer data between processors
  - Different types (volume, image, geometry)
  - Inports vs. outports
- Properties
  - Parameterize a processor
  - Can be linked
  - Can be animated

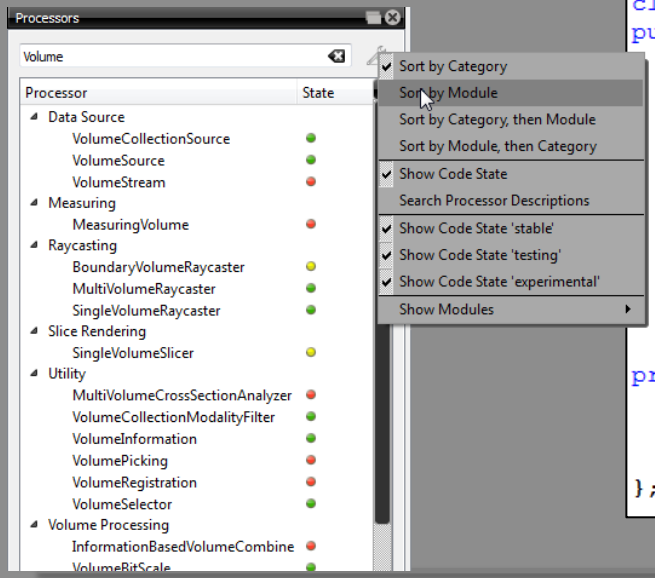


Processors

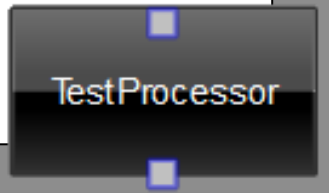

Serializable
AbstractSerializable
PropertyOwner
EventListener
Processor
RenderProcessor
VolumeRenderer
ProxyGeometry
SliceRendererBase
VolumeRaycaster
CubeProxyGeometry
FlowOrthogonalSliceRenderer
MultiplanarSliceRenderer
SliceViewer
CPURaycaster
CurvatureRaycaster
GLSLRaycaster

Processors

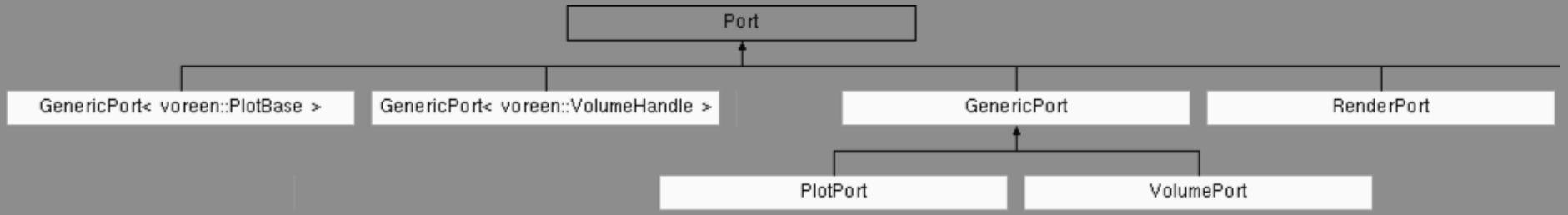
- Processors are described by their category, class name and the module they belong to
- Code state and processor info provide stability remarks and a brief functionality description




```
class TestProcessor : public RenderProcessor {
public:
  TestProcessor();

  virtual std::string getCategory() const { return "Test Processing"; }
  virtual std::string getClassName() const { return "Test Processor"; }
  virtual std::string getModuleName() const { return "test"; }
  virtual Processor::CodeState getCodeState() const { return CODE_STATE_STABLE; }
  virtual std::string getProcessorInfo() const;
  virtual Processor* create() const { return new TestProcessor(); }

  void process();

protected:

  RenderPort inport_;
  RenderPort outport_;
};
```



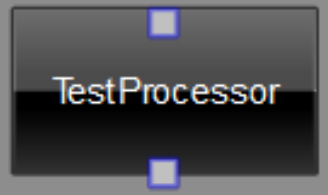
Ports


Port
GenericPort< voreen::PlotBase >
GenericPort< voreen::VolumeHandle >
GenericPort
RenderPort
PlotPort
VolumePort

Ports

Initializing ports

```
TestProcessor::TestProcessor()
    : RenderProcessor(),
      inport_(Port::INPORT, "inport"),
      outport_(Port::OUTPORT, "outport")
{
  addPort(inport_);
  addPort(outport_);
}
```

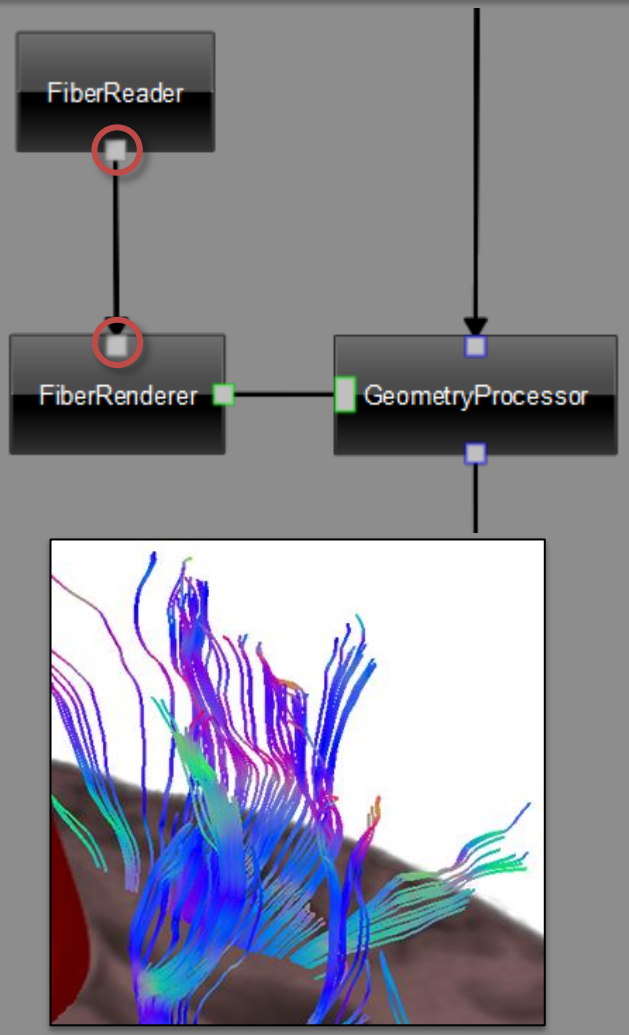


Using ports

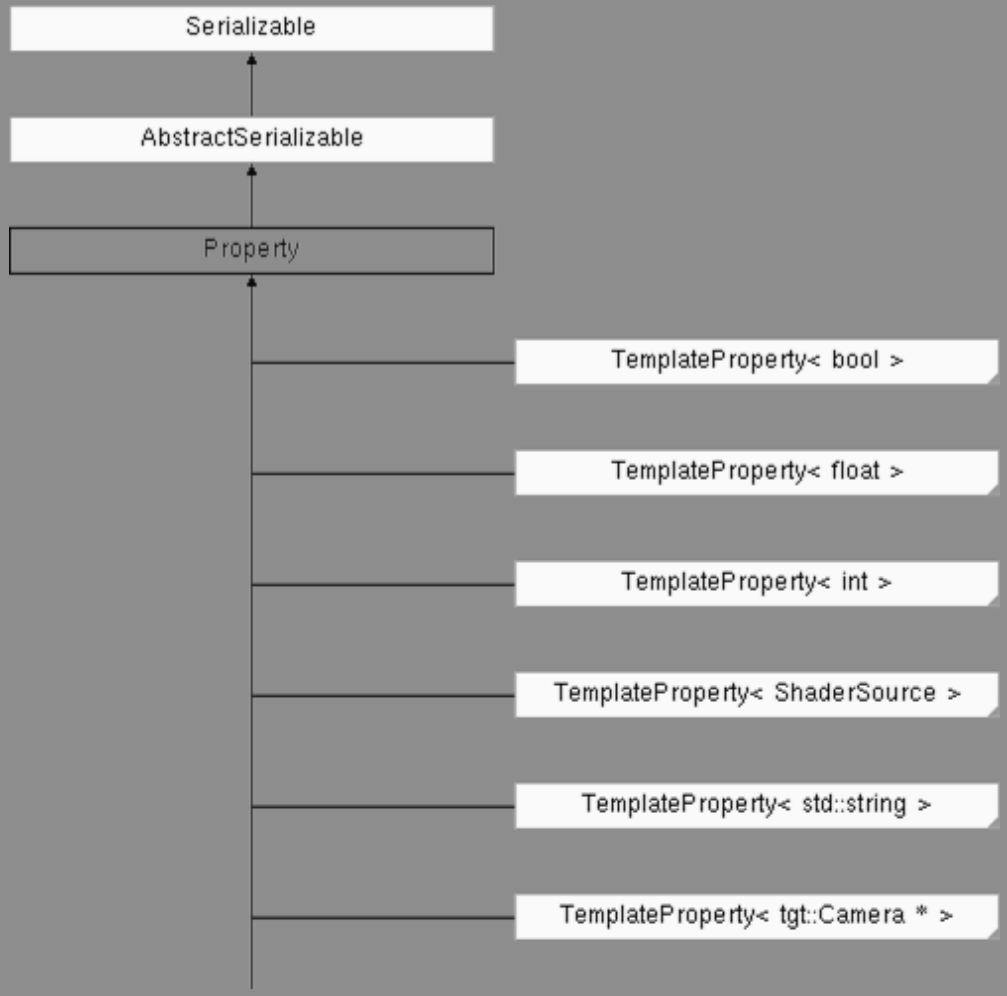
```
void TestProcessor::process() {
    // activate and clear output render target
    outport_.activateTarget();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // bind color texture incoming through inport_
    tgt::TextureUnit texUnit0;
    inport_.bindColorTexture(texUnit0.getEnum());
```

Realizing new Port Types

```
#include "voreen/core/ports/genericport.h"

namespace voreen {

  struct FiberLine {
    //start and end indices:
    size_t start_;
    size_t end_;

    float length_;
    bool visible_;
    int segmentId_;

    FiberLine() : start_(0), end_(0), length_(0.0f), visible_(true) {}
  };

  struct Fibers {
    std::vector<FiberLine> lines_;

    std::vector<tgt::vec3> points_;
    std::vector<float> uncertainties_;

    void updateLength() {
      for(size_t i=0; i<lines_.size(); i++) {
        lines_[i].length_ = 0.0f;
        for(size_t j=lines_[i].start_; j<lines_[i].end_; j++)
          lines_[i].length_ += distance(points_[j], points_[j+1]);
      }
    }
    size_t size() { return lines_.size(); }
  };

  typedef GenericPort<Fibers> FiberPort;

} // namespace voreen
```

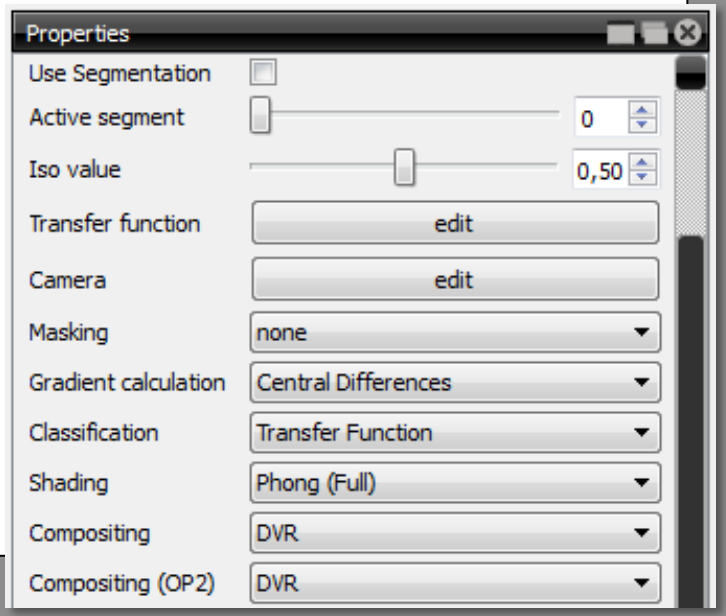


Properties


Serializable
AbstractSerializable
Property
TemplateProperty< bool >
TemplateProperty< float >
TemplateProperty< int >
TemplateProperty< ShaderSource >
TemplateProperty< std::string >
TemplateProperty< tgt::Camera * >

Adding Properties

```
SingleVolumeRaycaster::SingleVolumeRaycaster()
    : VolumeRaycaster()
    , transferFunc_("transferFunction", "Transfer function")
    , camera_("camera", "Camera", new tgt::Camera(vec3(0.f, 0.f, 3.5f), vec3(0.f, 0.f, 0.f), vec3(0.f, 1.f, 0.f)))
    , compositingMode1_("compositing1", "Compositing (OP2)", Processor::INVALID_PROGRAM)
    , compositingMode2_("compositing2", "Compositing (OP3)", Processor::INVALID_PROGRAM)
    , volumeInport_(Port::INPORT, "volumehandle.volumehandle", false, Processor::INVALID_PROGRAM)
    , entryPort_(Port::INPORT, "image.entrypoints")
    , exitPort_(Port::INPORT, "image.exitpoints")
    , outport_(Port::OUTPORT, "image.output", true, Processor::INVALID_PROGRAM)
    , outport1_(Port::OUTPORT, "image.output1", true, Processor::INVALID_PROGRAM)
    , outport2_(Port::OUTPORT, "image.output2", true, Processor::INVALID_PROGRAM)
{
    addProperty(useSegmentation_);
    addProperty(segment_);
    addProperty(isoValue_);

    addProperty(transferFunc_);
    addProperty(camera_);

    addProperty(maskingMode_);
    addProperty(gradientMode_);
    addProperty(classificationMode_);
    addProperty(shadeMode_);
    addProperty(compositingMode_);

    compositingMode1_.addOption("dvr", "DVR");
    compositingMode1_.addOption("mip", "MIP");
    compositingMode1_.addOption("iso", "ISO");
    compositingMode1_.addOption("fhp", "FHP");
    compositingMode1_.addOption("fhn", "FHN");
    addProperty(compositingMode1_);
```

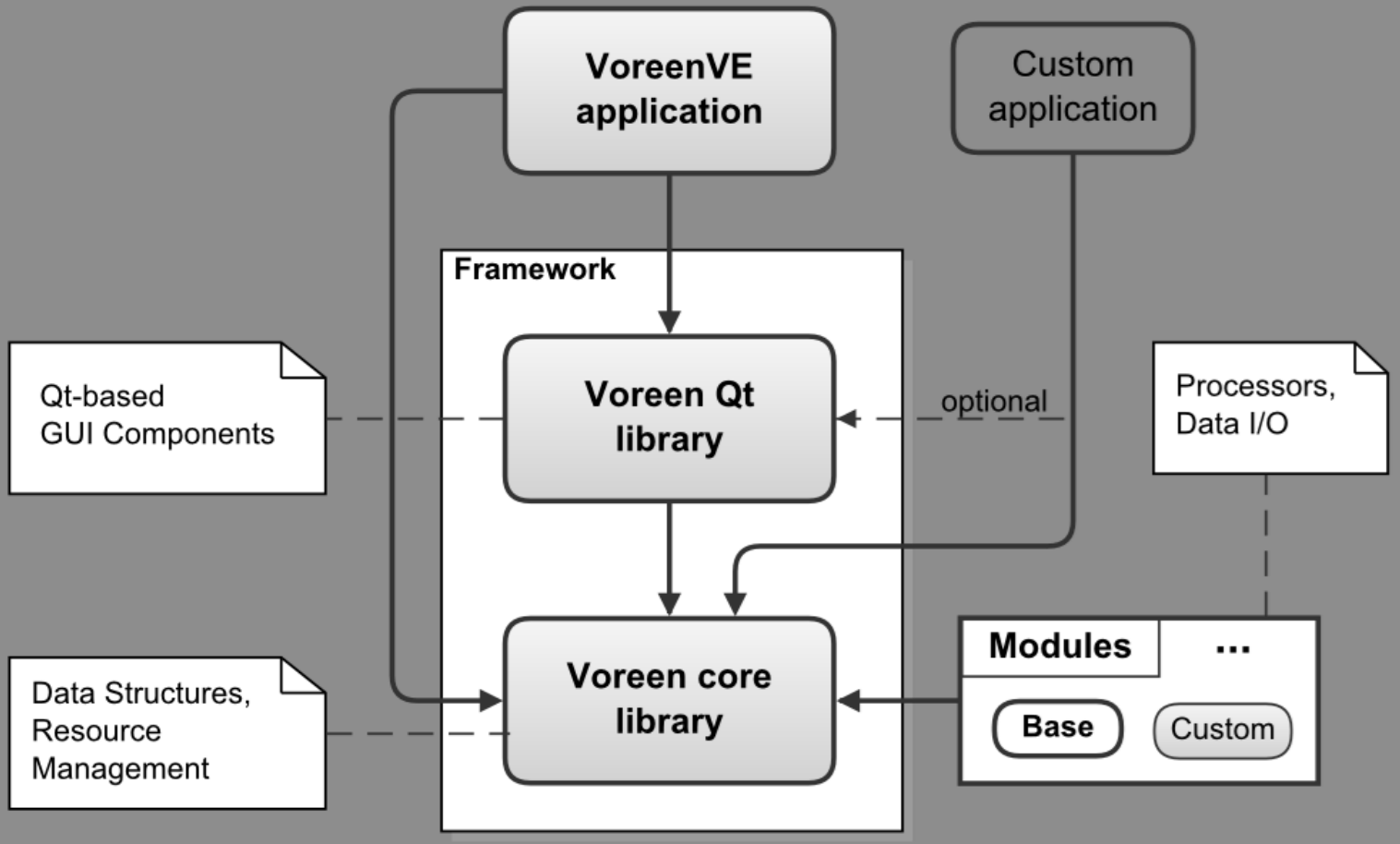



```
raycastPrg_->setUniform("isoValue_", isoValue_.get());
```

Project Structure

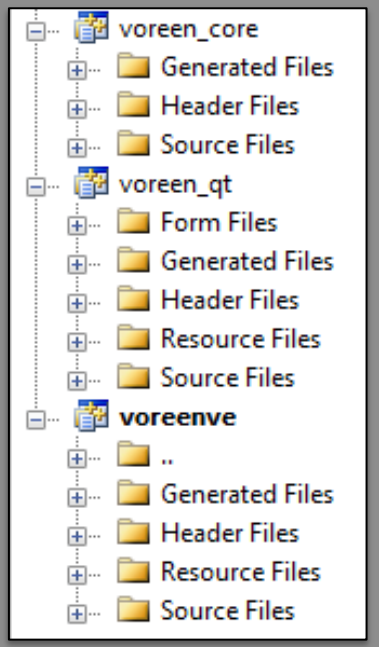
Framework and modules

Voreen Architecture

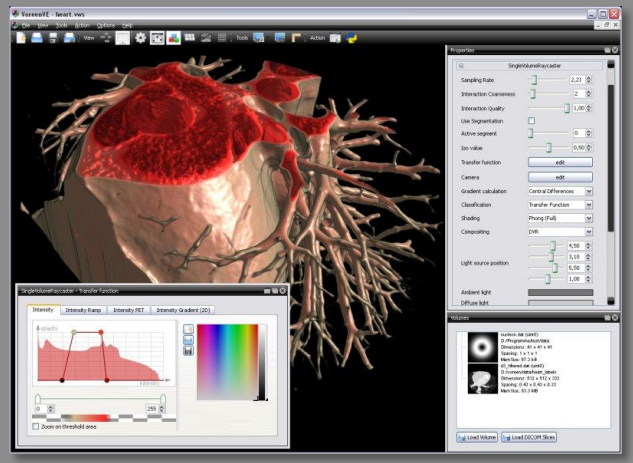


Framework

- Voreen core library
  - Ports, Properties
  - Processor base classes
    - `Processor, VolumeRenderer, ImageProcessor, …`
  - Data structures
    - Data-flow network, volumes, images, geometry
  - Network handling
  - Minimal dependencies:
    OpenGL, GLEW, TinyXML
- Voreen Qt library
  - Property widgets
  - Processor widgets



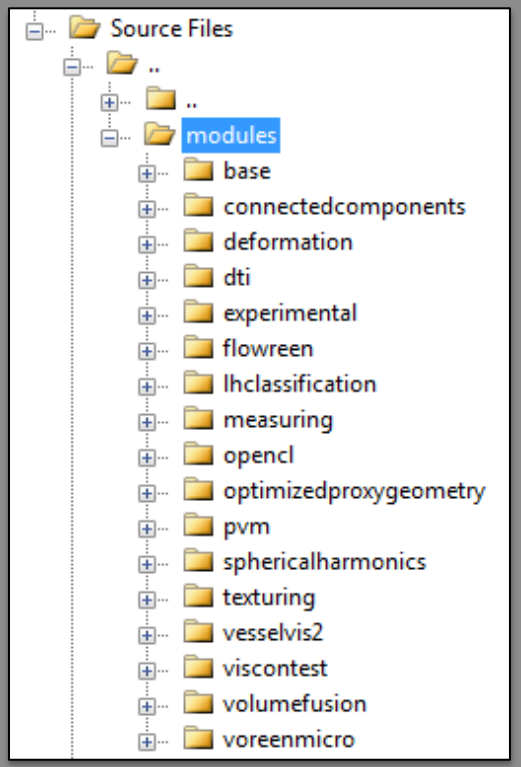
VoreenVE Application



- Rapid prototyping environment
  - Graphical network editor
  - Auto-generated property widgets (Voreen Qt library)
  - Visual debugging
    - Inspection of intermediate rendering results
  - Runtime shader editing
- Application mode for domain experts
  - Hides the underlying network
  - Visibility of single properties can be configured

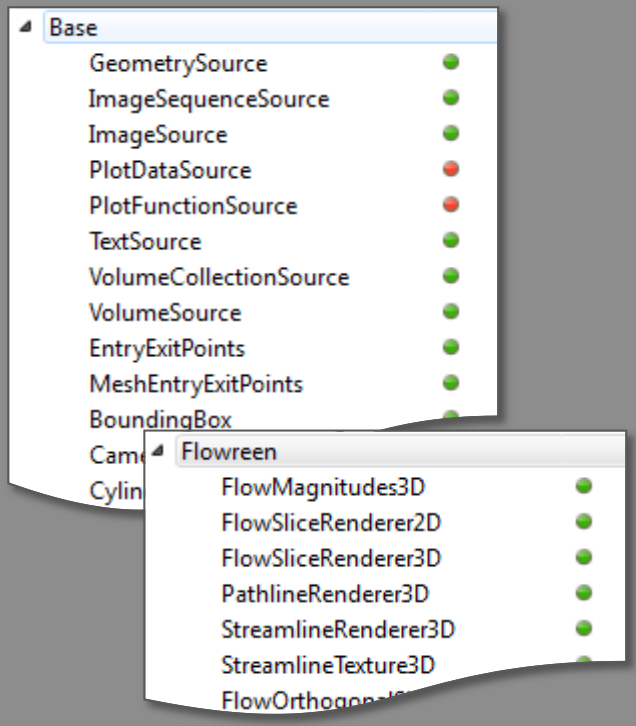
Modules

- Recommended way to extend Voreen
- Encapsulate rendering and data processing functionality
  - Processors
  - Data readers and writers
- Are included/excluded from the build process by a single line in `config.txt`
- May contain external libraries



Important Modules

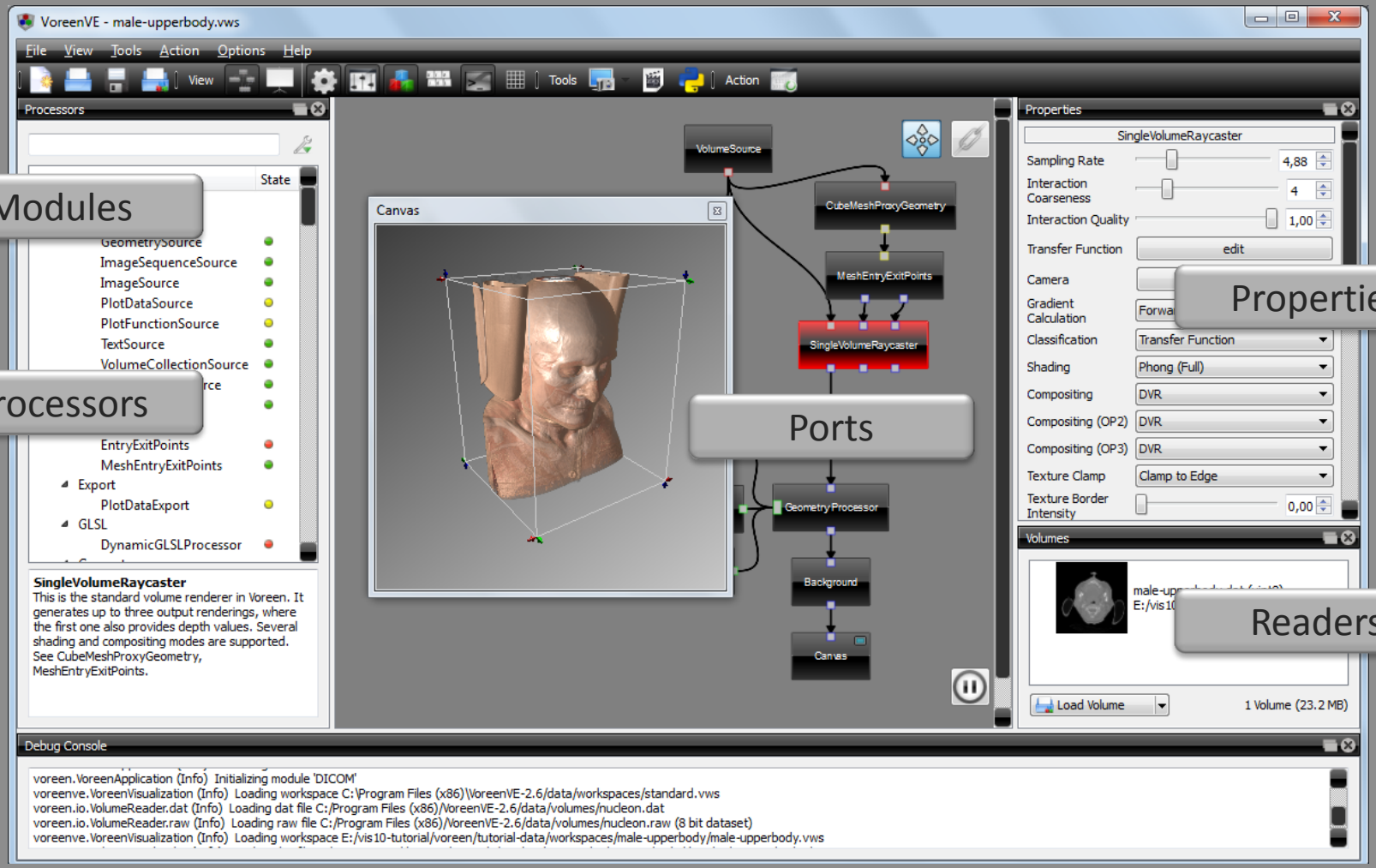
- *Base*
  - Basic rendering and data processing functionality
  - Volume and geometry renderers
  - Image and volume processors
  - Volume I/O
- *DICOM*
  - DICOM volume reader and writer
  - Requires DCMTK
- *Flowreen*
  - Flow visualization
- *Python*
  - Scripting



Extending Voreen

Adding modules and processors

What Can Be Customized



Module Structure

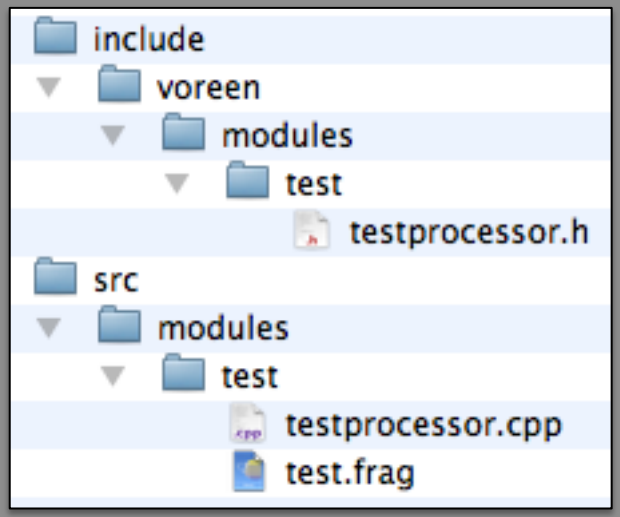
- Location
  - Source files: `src/modules/<modulename>/`
  - Headers: `include/voreen/modules/<modulename>/`
- Project files
  - Integrate code files and libraries into build process
  - Located in module‘s source directory
- Module class
  - Registers resources at runtime
  - Adds shader search paths
  - Derived from `VoreenModule`

Module Recipe

1. Create module directories
   - Place code, header and shader files there
2. Write module class
   - Instantiate and register module processors
   - Add shader search path
3. Create module project files
   - Specify class name and filenames of module class
   - Reference code, header and shader files
4. Activate module
   - Add `VRN_MODULES += <module>` to `config.txt`

Example: *TestModule*

- We add a module named *TestModule*
  - Processor `TestProcessor`
  - Shader file `test.frag`
- 1. Create module directories



The *TestModule* archive
can be downloaded at
www.voreen.org/files/testmodule.zip

2. Write Module Class

include/voreen/modules/test/testmodule.h

```
#ifndef VRN_TESTMODULE_H
#define VRN_TESTMODULE_H

#include "voreen/core/voreenmodule.h"

namespace voreen {

class TestModule : public VoreenModule {

public:
  TestModule();

  std::string getDescription() const {
    return "My first test module.";
  }
};

} // namespace

#endif // VRN_TESTMODULE_H
```

src/modules/test/testmodule.cpp

```
#include "voreen/modules/test/testmodule.h"
#include "voreen/modules/test/testprocessor.h"

namespace voreen {

TestModule::TestModule()
  : VoreenModule()
{
  // module name
  setName("Test");

  // each module processor has to be registered
  addProcessor(new TestProcessor());

  // adds <VOREEN_ROOT>/src/modules/test
  // to the shader search path
  addShaderPath(getModulesPath("test"));
}

} // namespace
```

3. Create Module Project Files

include/voreen/modules/test/test_common.pri

```
# module availability macro
DEFINES += VRN_MODULE_TEST

# name of the module class
VRN_MODULE_CLASSES += TestModule

# module class header and source file. Paths are relative to
# module base directories 'include/voreen/modules' and 'src/modules', resp.
VRN_MODULE_CLASS_HEADERS += test/testmodule.h
VRN_MODULE_CLASS_SOURCES += test/testmodule.cpp
```

include/voreen/modules/test/test_core.pri

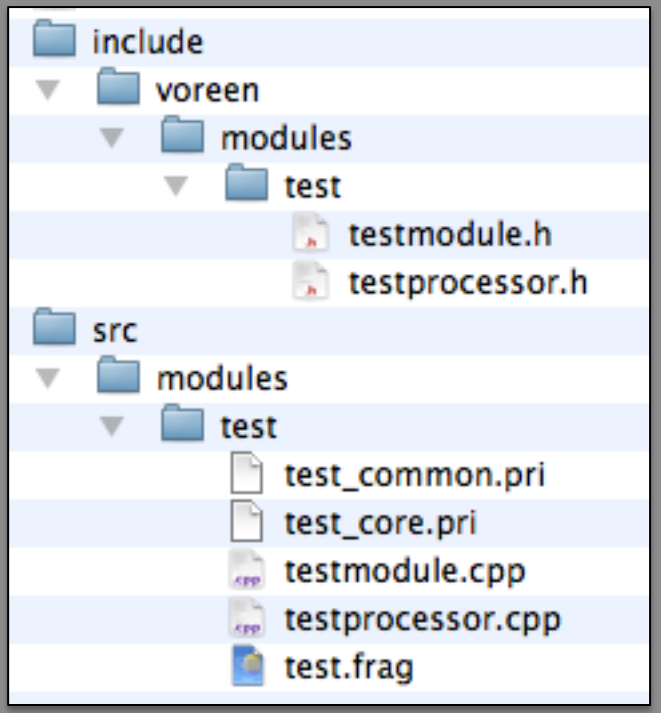
```
# processor headers
HEADERS += $${VRN_MODULE_INC_DIR}/test/testprocessor.h

# processor sources
SOURCES += $${VRN_MODULE_SRC_DIR}/test/testprocessor.cpp

# shaders
SHADER_SOURCES += $${VRN_MODULE_SRC_DIR}/test/test.frag
```

4. Activate Module

config.txt

```
...
VRN_MODULES += test
...
```

➔ Rebuild Project

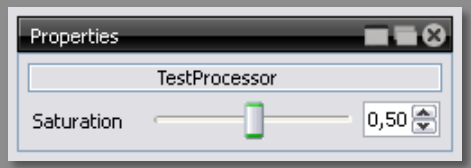


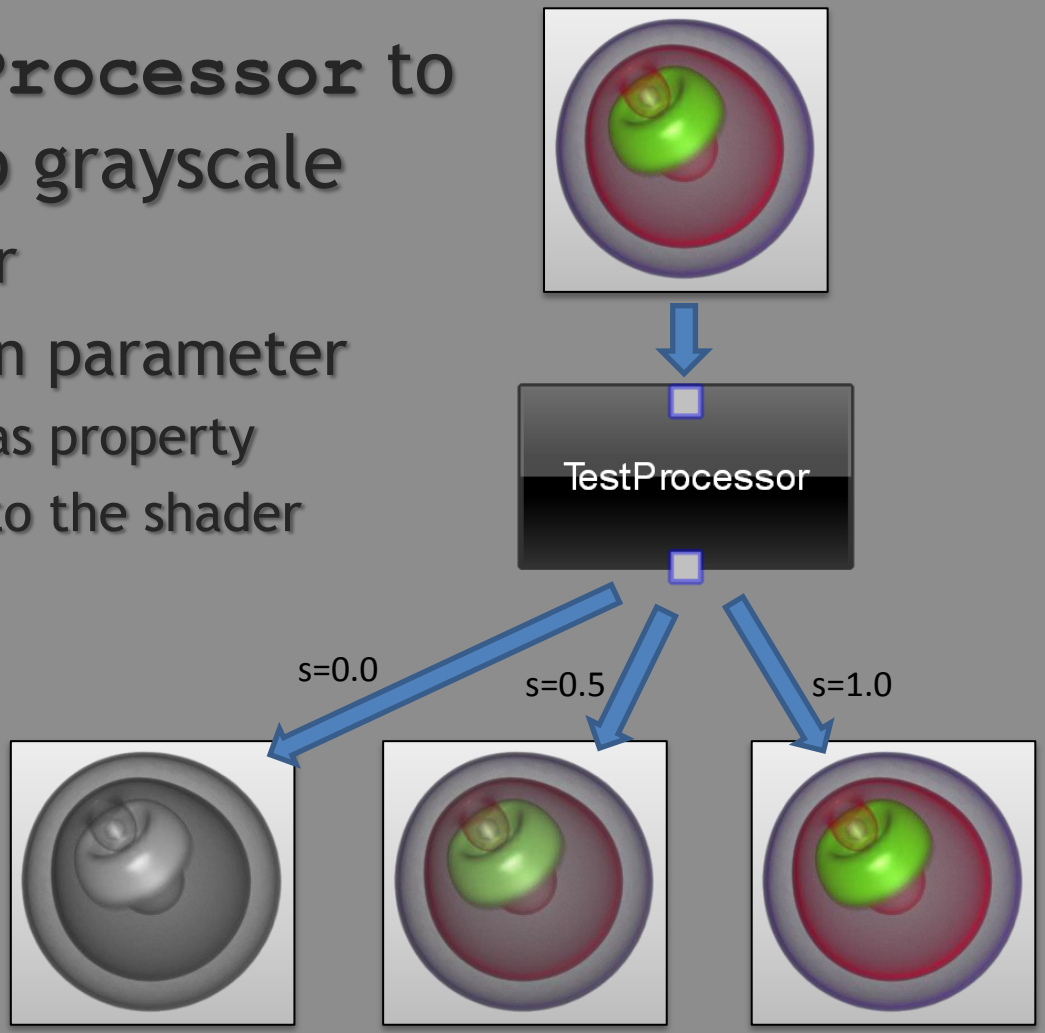
Shader Handling in Voreen

- **`tgt::Shader`**
  - Represents an OpenGL shader program
  - Contains vertex, fragment and/or geometry shaders
  - Methods for passing uniforms and attributes
  - C-like **`#include`** mechanism
  - Headers for adding generated code at runtime, e.g., preprocessor defines
- **`tgt::ShaderManager`**
  - Convenient loading of shaders from file
  - Shader search path
  - Singleton: accessible via define **`ShdrMgr`**

Example: Image Processor

- We want our **`TestProcessor`** to convert an image to grayscale
  - Use fragment shader
  - Additional saturation parameter
    - To be implemented as property
    - Needs to be passed to the shader





Header and Constructor

Header

```
#include "voreen/core/processors/renderprocessor.h"
#include "voreen/core/properties/floatproperty.h"
#include "tgt/shadermanager.h"

class TestProcessor : public RenderProcessor {
public:
  ...
protected:
  virtual void process();

  virtual void initialize()
      throw (VoreenException);

  virtual void deinitialize()
      throw (VoreenException);

private:
  RenderPort inport_;
  RenderPort outport_;
  FloatProperty saturation_;
  tgt::Shader* shader_;
};
```

Processors that use **RenderPorts** have to be derived from **RenderProcessor**

Loads the shader

Deletes the shader

Constructor

```
TestProcessor::TestProcessor()
  : RenderProcessor()
  , inport_(Port::INPORT, "inport")
  , outport_(Port::OUTPORT, "outport")
  , saturation_("saturation", "Saturation")
{
  addPort(inport_);
  addPort(outport_);
  addProperty(saturation_);
}
```

Loading the Shader

```
void TestProcessor::initialize()
  throw (VoreenException) {

  RenderProcessor::initialize();

  shader_ = ShdrMgr.loadSeparate(
      "passthrough.vert", //< dummy
      "test.frag",
      generateHeader()
  );

  if (!shader_)
    throw VoreenException(
        "failed to load shader");
}
```

OpenGL resources should be allocated in **`initialize()`** instead of in the constructor.

Call superclass function first

Necessary for shaders that access image or volume data

Exception indicates initialization failure

```
void TestProcessor::deinitialize()
  throw (VoreenException) {

  ShdrMgr.dispose(shader_);
  shader_ = 0;

  RenderProcessor::deinitialize();
}
```

Free the shader

Call superclass function last

Accessing Images in the Shader

- **`textureLookup2Dscreen()`**
  - Standard lookup function for input images
  - Expects fragment coordinates
  - Suitable, if the processor's input and output images have the same dimensions
  - Provided by shader module **`mod_sampler2D.frag`**
- Alternatives
  - **`textureLookup2Dnormalized()`**
    - expects normalized texture coordinates
  - **`textureLookup2D()`**
    - pixel coordinates

Accessing Images in the Shader (con't)

TestProcessor::process()

```
...
// 0. activate shader
shader_->activate();
setGlobalShaderParameters(shader_);

// 1. bind input image to texture units
tgt::TextureUnit colorUnit, depthUnit;
inport_.bindTextures(colorUnit, depthUnit);

// 2. pass texture units to shader
shader_->setUniform(
    "colorTex_",
    colorUnit.getUnitNumber());

shader_->setUniform(
    "depthTex_",
    depthUnit.getUnitNumber());

// 3. pass texture parameters
inport_.setTextureParameters(
    shader_,
    "texParams_");
...
```

test.frag

```
#include "modules/mod_sampler2d.frag"

uniform SAMPLER2D_TYPE colorTex_;
uniform SAMPLER2D_TYPE depthTex_;
uniform TEXTURE_PARAMETERS texParams_;

uniform float saturation_;

void main() {

  // lookup input color
  vec4 color = textureLookup2Dscreen(
      colorTex_,
      texParams_,
      gl_FragCoord.xy);

  // lookup depth value
  float depth = textureLookup2Dscreen(
      depthTex_,
      texParams_,
      gl_FragCoord.xy).z;

  ...
```

Complete Example: Grayscale

```
void TestProcessor::process() {
  // activate and clear output render target
  outport_.activateTarget();
  outport_.clearTarget();

  // activate shader
  shader_->activate();
  setGlobalShaderParameters(shader_);

  // bind input image to texture units
  tgt::TextureUnit colorUnit, depthUnit;
  inport_.bindTextures(colorUnit, depthUnit);

  // pass texture units and parameters to shader
  shader_->setUniform("colorTex_", colorUnit.getUnitNumber());
  shader_->setUniform("depthTex_", depthUnit.getUnitNumber());
  inport_.setTextureParameters(shader_, "texParams_");

  // pass property value to shader
  shader_->setUniform("saturation_", saturation_.get());

  // render screen aligned quad
  renderQuad();

  // cleanup
  shader_->deactivate();
  outport_.deactivateTarget();
  tgt::TextureUnit::setZeroUnit();

  // check for OpenGL errors
  LGL_ERROR;
}
```

```
#include "modules/mod_sampler2d.frag"

uniform SAMPLER2D_TYPE colorTex_;
uniform SAMPLER2D_TYPE depthTex_;
uniform TEXTURE_PARAMETERS texParams_;

uniform float saturation_;

void main() {

  // lookup input color
  vec4 color = textureLookup2Dscreen(
    colorTex_,
    texParams_,
    gl_FragCoord.xy);

  // lookup depth value
  float depth = textureLookup2Dscreen(
    depthTex_,
    texParams_,
    gl_FragCoord.xy).z;

  // compute gray value
  float brightness =
    (0.30*color.r)+(0.59*color.g)+(0.11*color.b);
  vec4 graycol = vec4(vec3(brightness), color.a);
  FragData0 = mix(graycol, color, saturation_);

  // pass through depth value
  gl_FragDepth = depth;

}
```

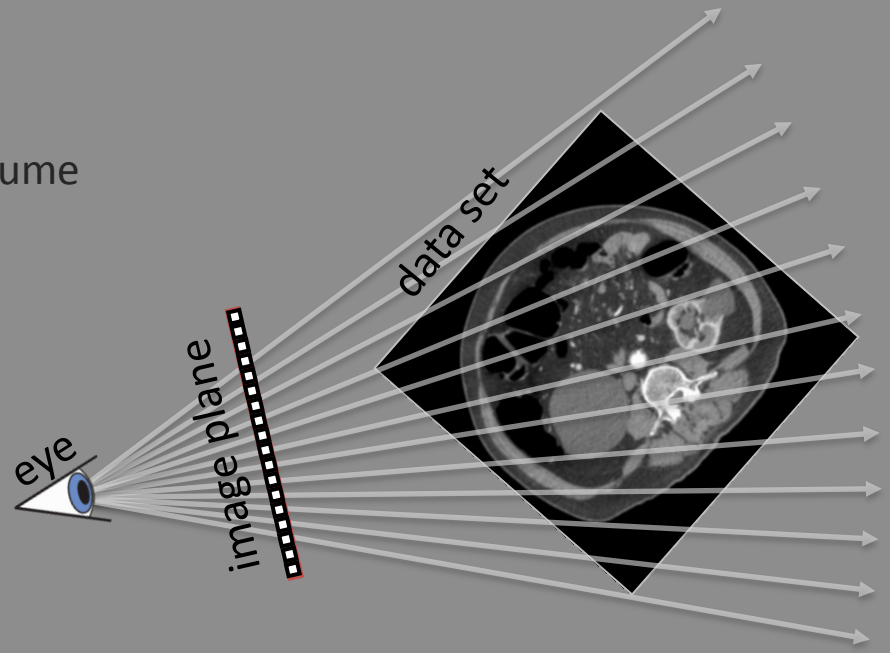
Accessing Volume Data in Shaders

- **`getVoxel()`**
  - Shader lookup function for volume data
  - Expects (normalized) texture coordinates
  - Provided by shader module **`mod_sampler3d.frag`**
- **`VolumeRenderer::bindVolumes()`**
  - Passes volume textures to the shader
  - Sets additional meta data for each volume, such as the data set‘s dimensions and bit depth
    ➔ see struct **`VOLUME_PARAMETERS`** in **`mod_sampler3d.frag`**

Volume Ray-Casting

- Cast rays through each pixel and the volume

for each pixel on the image plane
    compute entry- and exit-points
    while current position inside volume
        read intensity
        apply transfer function
        (compute shading)
        apply compositing
        compute new position
    end while
    set pixel color
end if



Example: Volume Raycaster

```
void SampleRaycaster::process() {
  // ... activate outport and shader
  // ... pass entry/exit point textures

  // pass volume texture to shader
  std::vector<VolumeStruct> volumeTextures;
  TextureUnit volUnit;
  volumeTextures.push_back(VolumeStruct(
      volumePort_.getData()->getVolumeGL(),
      &volUnit,
      "volume_",
      "volumeParams_")
  );
  bindVolumes(raycastPrg_, volumeTextures);

  // pass transfer function to shader
  TextureUnit transferUnit;
  transferUnit.activate();
  transferFunc_.get()->bind();
  raycastPrg_->setUniform("transferFunc_",
      transferUnit.getUnitNumber());

  renderQuad();

  // ... clean up
}
```

```
// includes all shader modules
#include "modules/vrn_shaderincludes.frag"
...
vec4 result = vec4(0.0); float t = 0.0;
while ((t < t_end) && (result.a < 1.0)) {
  // determine and update sampling position
  vec3 sample = frontPos.rgb + t*direction;
  t += samplingStepSize_;

  // fetch intensity at sampling point
  // (see mod_sampler3d.frag)
  float voxel = getVoxel(
      volume_,
      volumeParams_,
      sample).a;

  // apply transfer function
  // (see mod_transfunc.frag)
  vec4 col = applyTF(transferFunc_, voxel);

  // perform compositing
  col.a *= samplingStepSizeComposite_;
  result.rgb += (1.0-result.a)*col.a*col.rgb;
  result.a += (1.0-result.a)*col.a;
}
...
```

How to Extend Voreen - Summary

- Copy&Paste the *Test* module
  - *www.voreen.org/files/testmodule.zip*
- Use existing processors as templates
  - Image processing
    - **`TestProcessor`** (test module)
  - Volume raycasting
    - **`SimpleRaycaster`** (base module)
  - Volume processing
    - **`VolumeInversion`** (base module)
  - Shader-based rapid prototyping
    - **`DynamicGLSLProcessor`** (base module)

Serialization

- **`Workspace::save()`** serializes network topology and processor properties to XML
- Custom data can be serialized by overriding **`Processor::serialize()`** and **`deserialize()`**
  - Serializer supports primitive types and STL containers

```
std::vector< std::pair<float, tgt::vec3> > myData_;

void TestProcessor::serialize(XmlSerializer& s) const {
    s.serialize("MyData", myData_);
}
void TestProcessor::deserialize(XmlDeserializer& d) {
    d.deserialize("MyData", myData_);
}
```

- Custom classes can be serialized by implementing interface **`Serializable`**

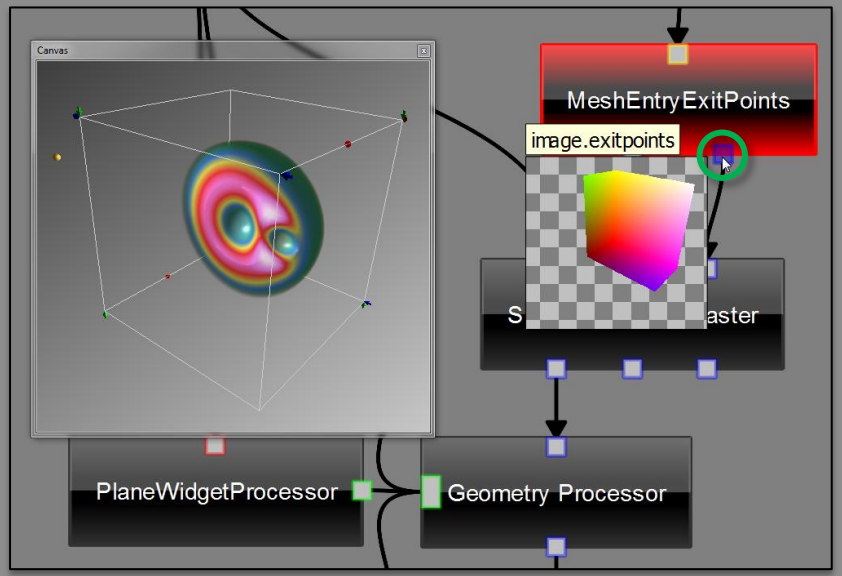
Visual Debugging

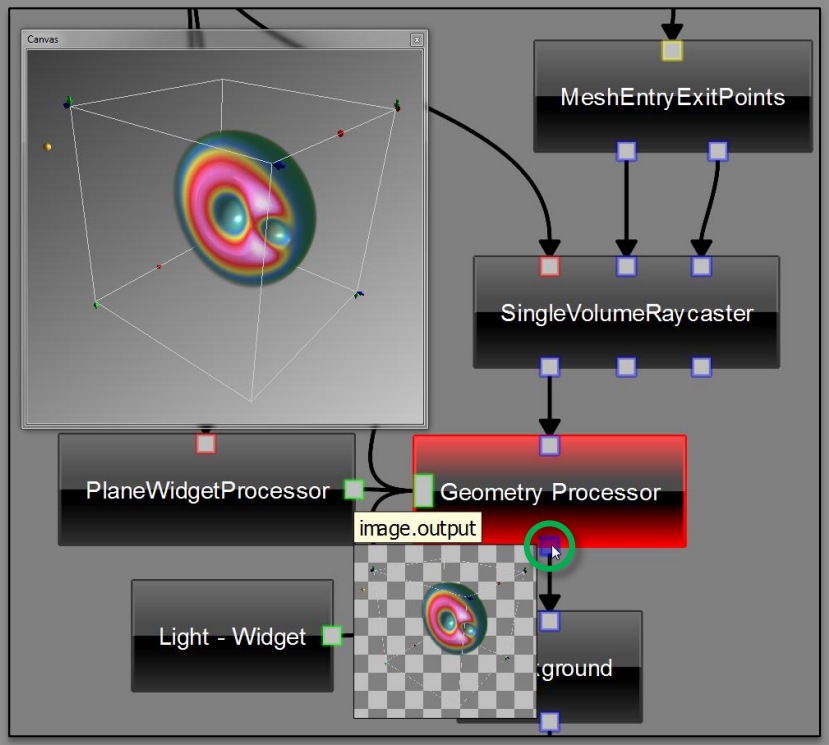
Run-time shader editing and framebuffer inspection

Demo

Render Target Inspection

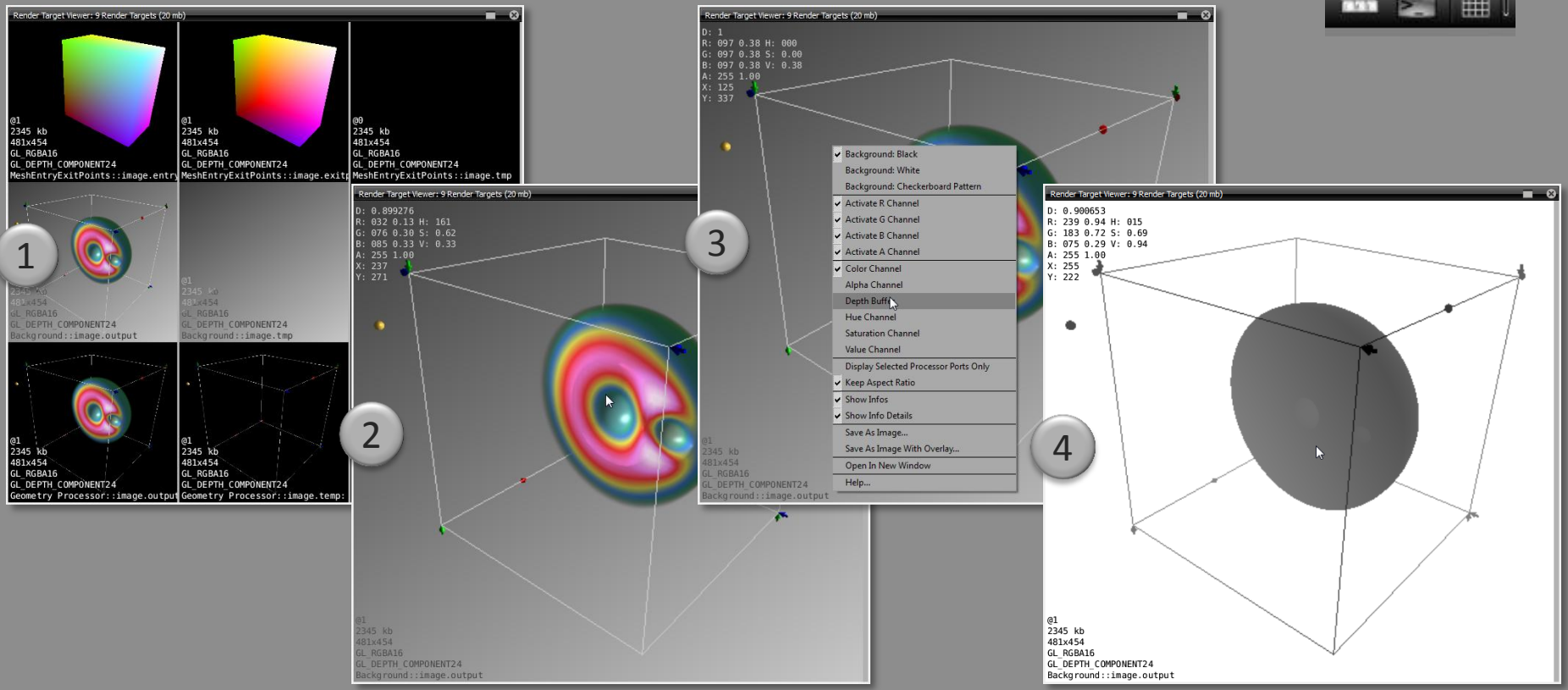
- By hovering over render ports, their content can be inspected





In Detail Inspection

- The render target viewer allows to inspect the color, alpha and depth layer

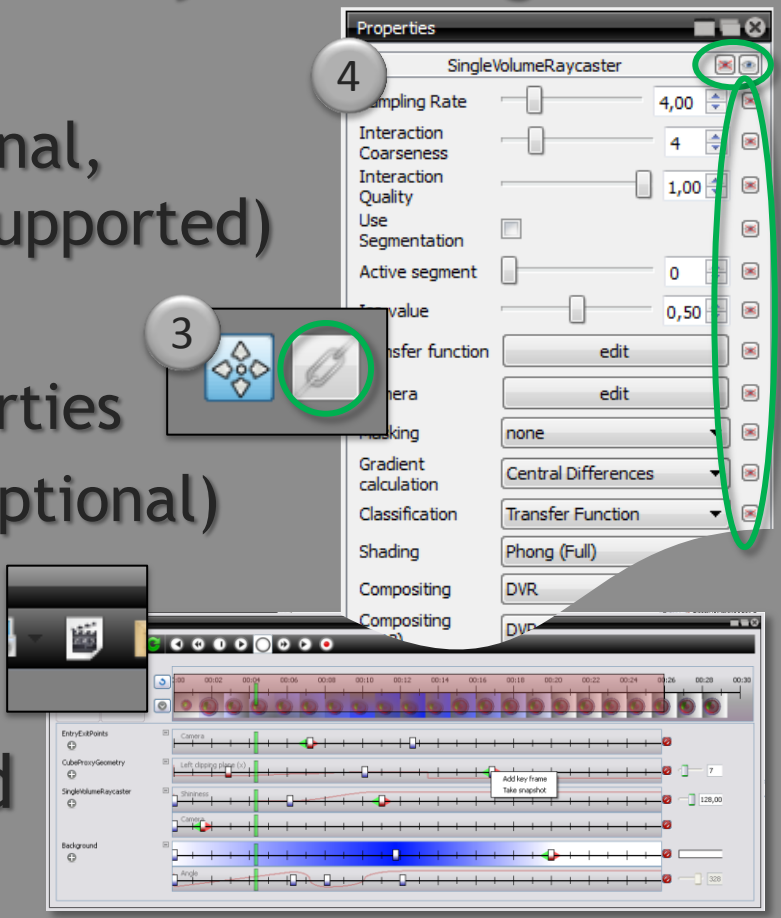


Customizing Applications

Generating end-user applications

Application Development Recipe

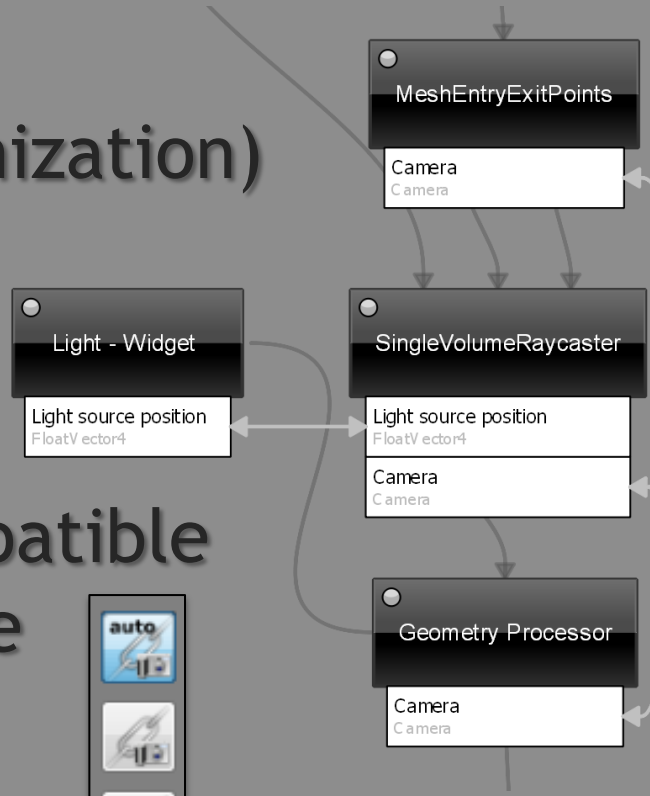
- Applications can be developed by following a few easy steps
  1. Write data importers (optional, many formats are already supported)
  2. Develop missing processors
  3. Define links between properties
  4. Define property visibility (optional)



- To showcase visualizations animations can be recorded

Property Linking

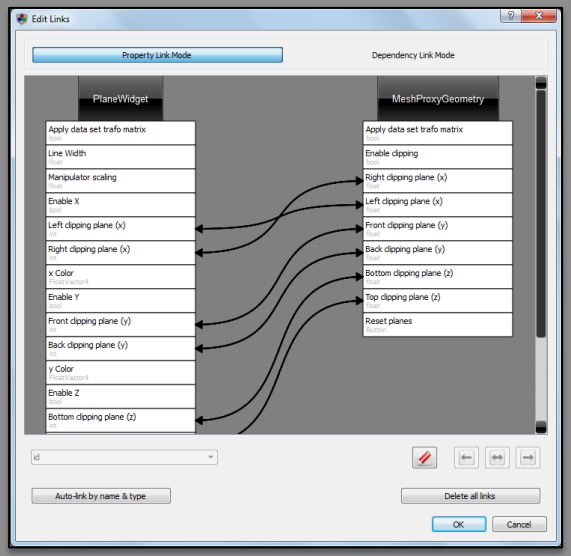
- Properties of the same type can be linked (value synchronization)
  - Within or across processors
  - Uni- or bidirectional
  - Cycle prevention
- Linking of differing, but compatible property types is also possible
  - Float ⇔ Integer ⇔ Boolean
- Transfer function linking
- Optional auto-linking of camera properties

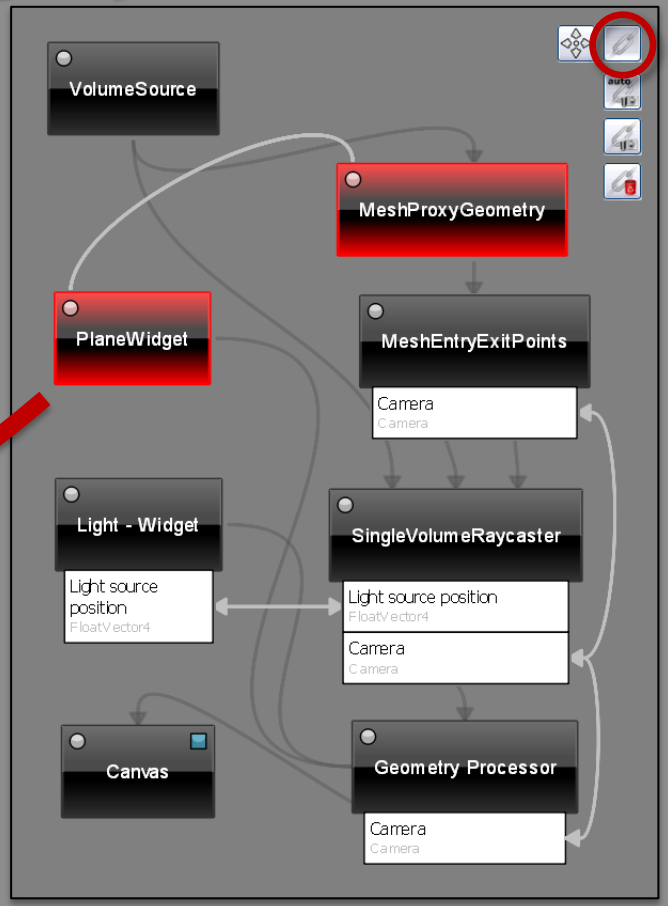


Demo

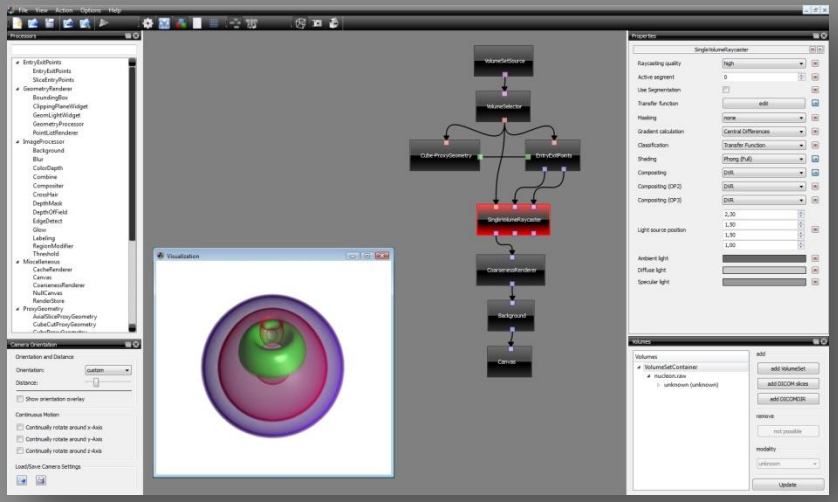
Managing Links in VoreenVE

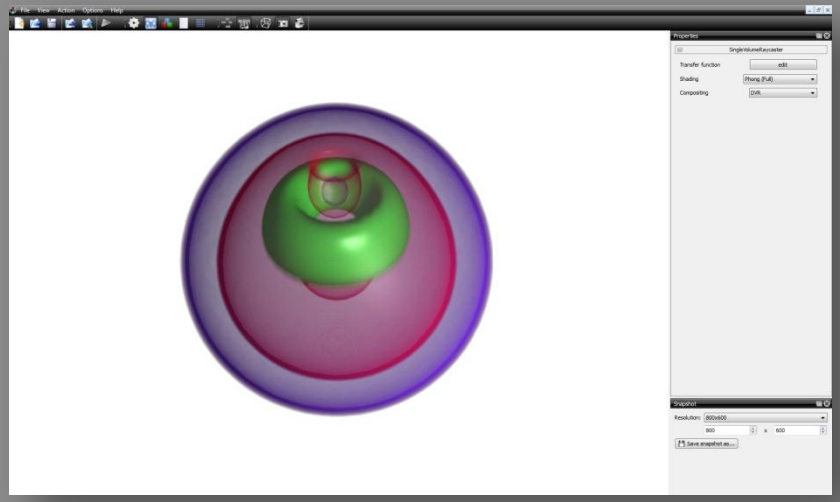
- Network editor provides *linking layer*
  - Links are represented by arrows
  - Port connections are fade out
  - Dragging a line between processors opens *linking dialog*





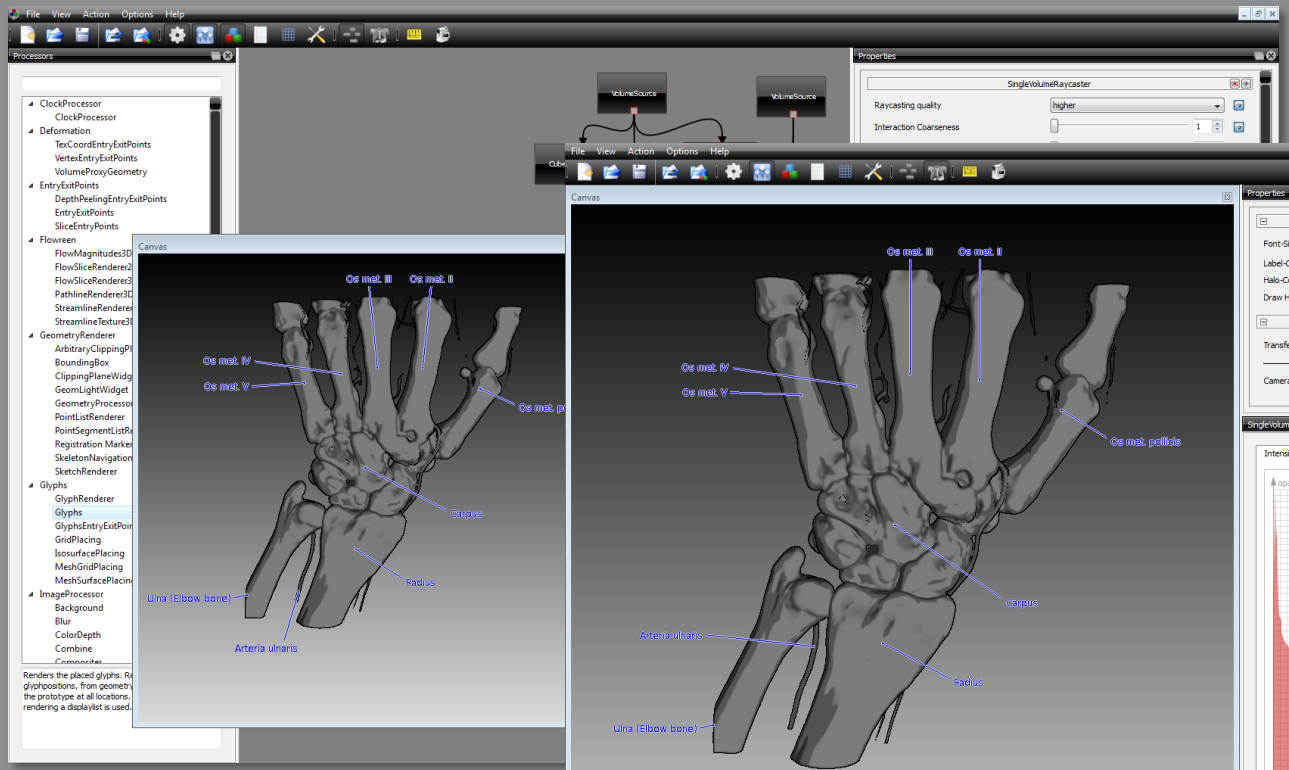
Application Prototyping





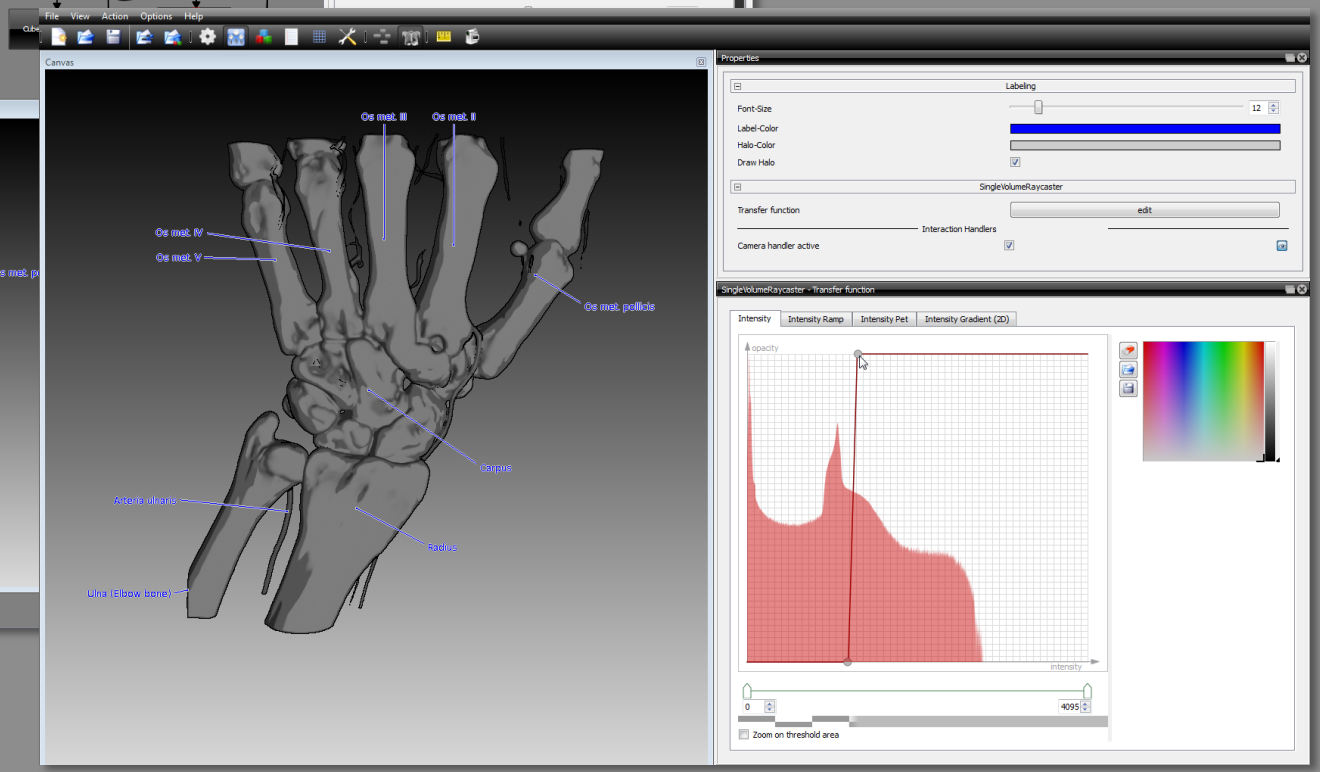
- Development Mode
  - Edit data-flow network
  - Specify properties changeable in Application Mode
- Application Mode
  - Explore visualization by altering provided properties

Application Prototyping



Development Mode

Application Mode



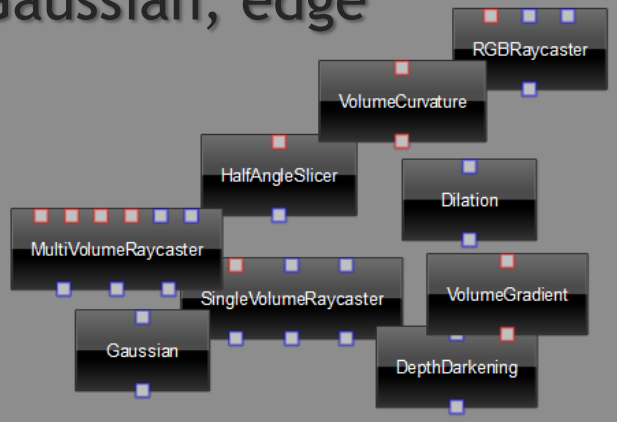
Demo

Additional Features

Helpful processors and other functionality

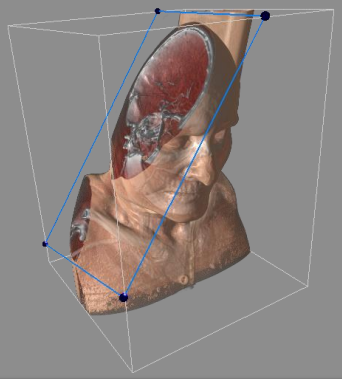
Processor Potpourri

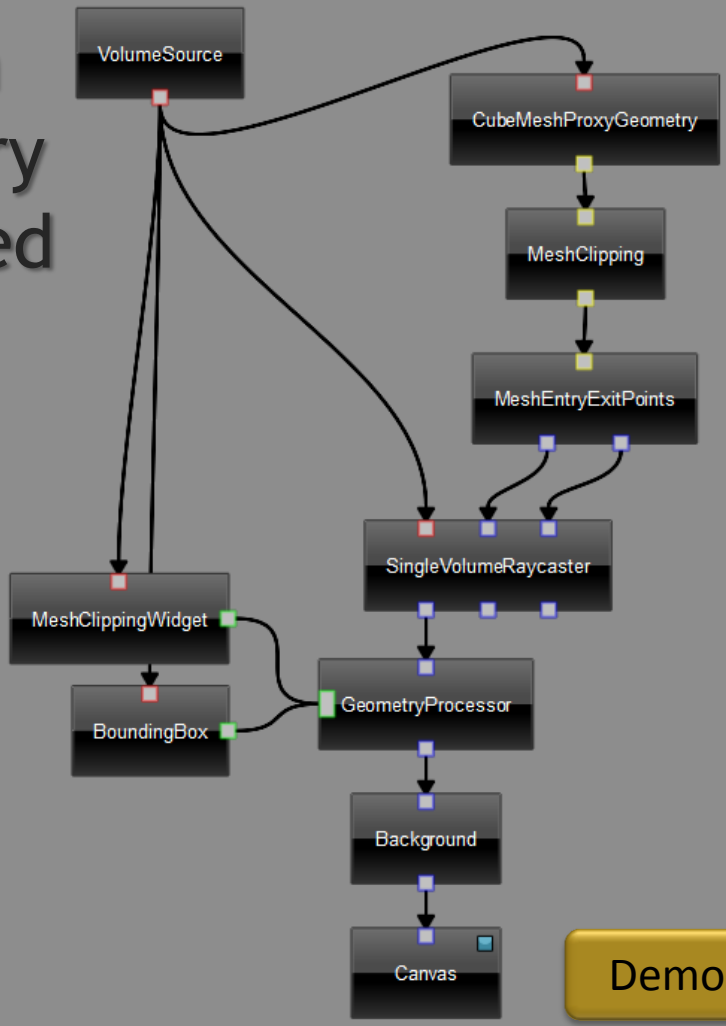
- Volume ray-casting
  - Single volume, multiple volume, advanced shading, RGB data …
- Volume slicing
  - Regular, half-angle slicing, directional occlusion shading…
- Image processing
  - Depth darkening, dilation, erosion, Gaussian, edge detection …
- Volume processing
  - Median, gradients, curvature …
- …



Additional Features - Clipping

- Axis-aligned clipping and an arbitrary number of arbitrary clipping planes are supported
  - Axis-aligned: **`CubeMeshProxyGeometry`**
  - Arbitrary: **`MeshClipping`** (sequential)

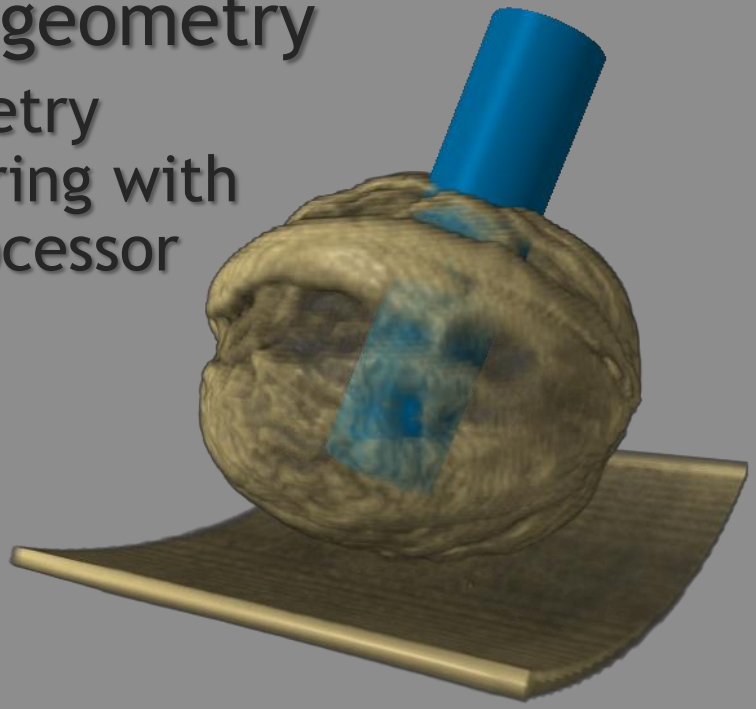


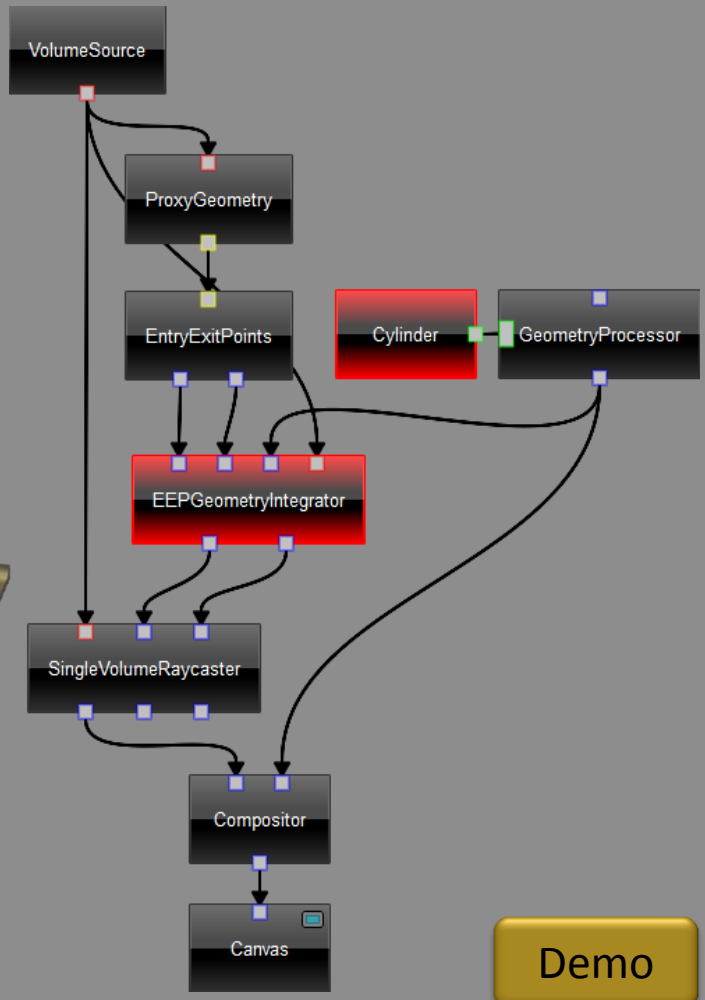


Demo

Additional Features - Geometry

- Combination of DVR and opaque geometry
  - Geometry rendering with co-processor ports



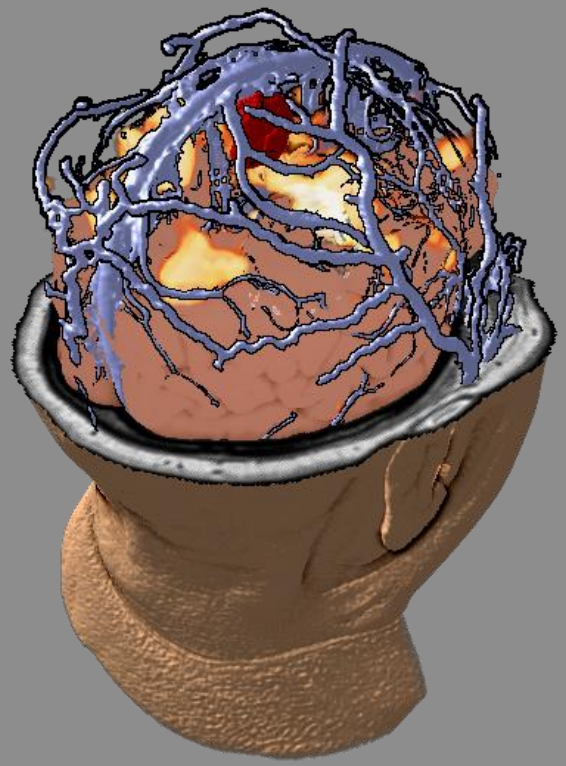


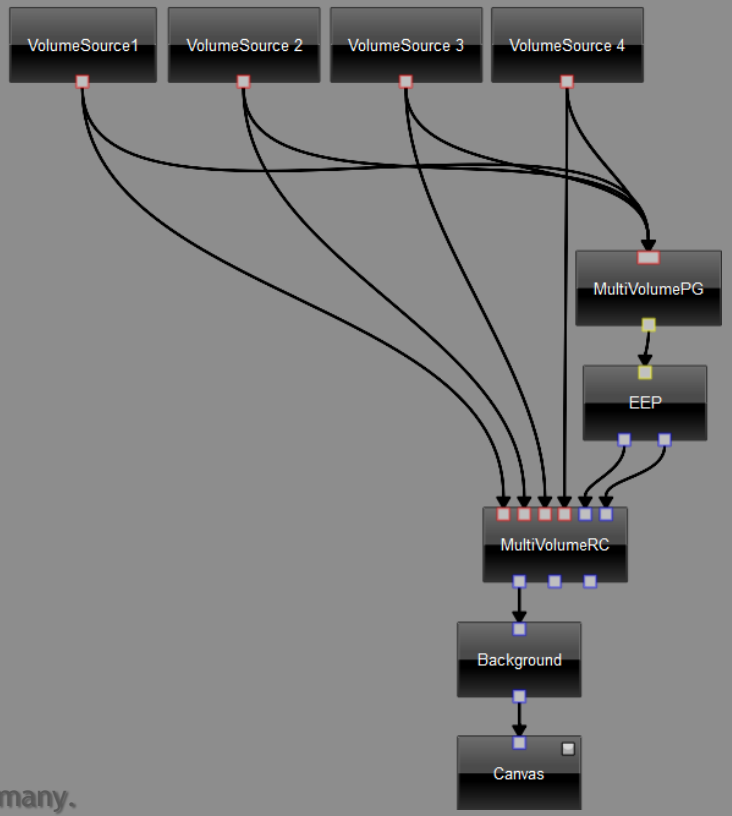
- Semi-transparent geometry through depth peeling

Demo

Additional Features - Multi Volume

- Multi-Volume ray-casting is done with the `MultiVolumeRaycasting` processor

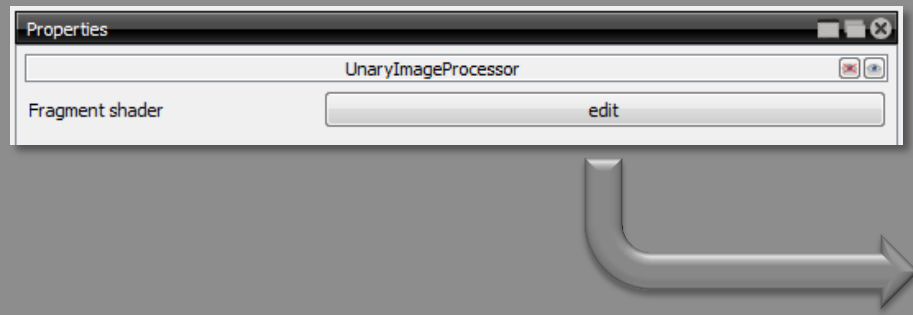


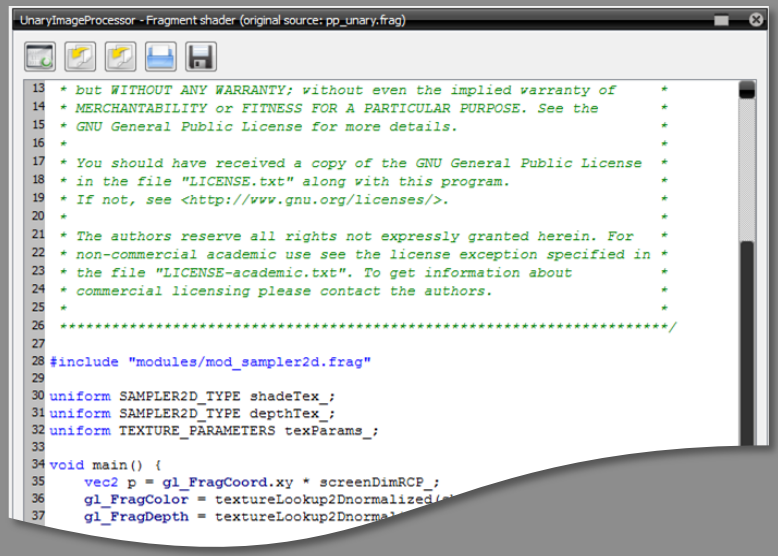


The shown dataset is courtesy of Prof. B. Terwey, Klinikum Mitte, Bremen, Germany.

GLSL Image Processing

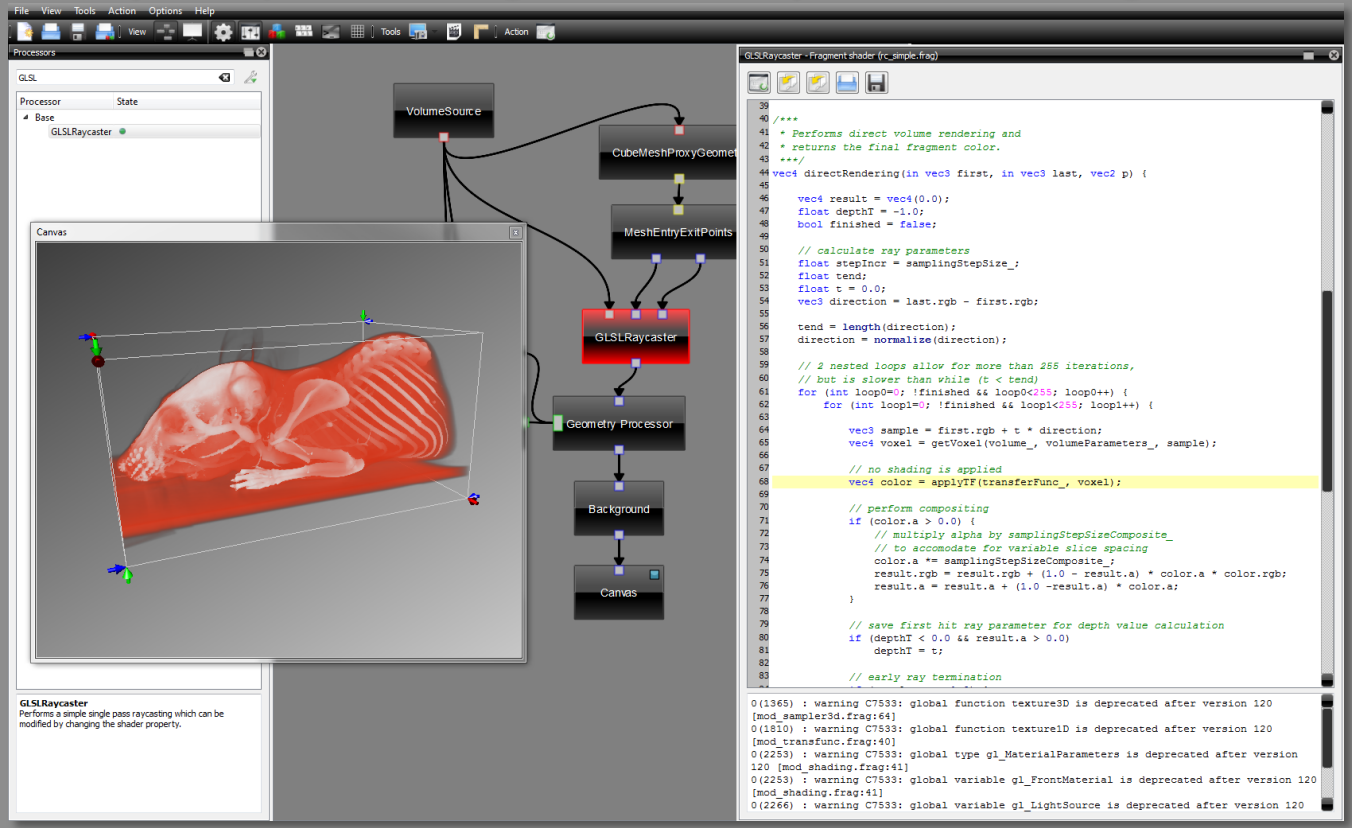
- Shader properties allow runtime shader editing
- **`UnaryImageProcessor`** and **`BinaryImageProcessor`** allow to implement image processing and compositing effects





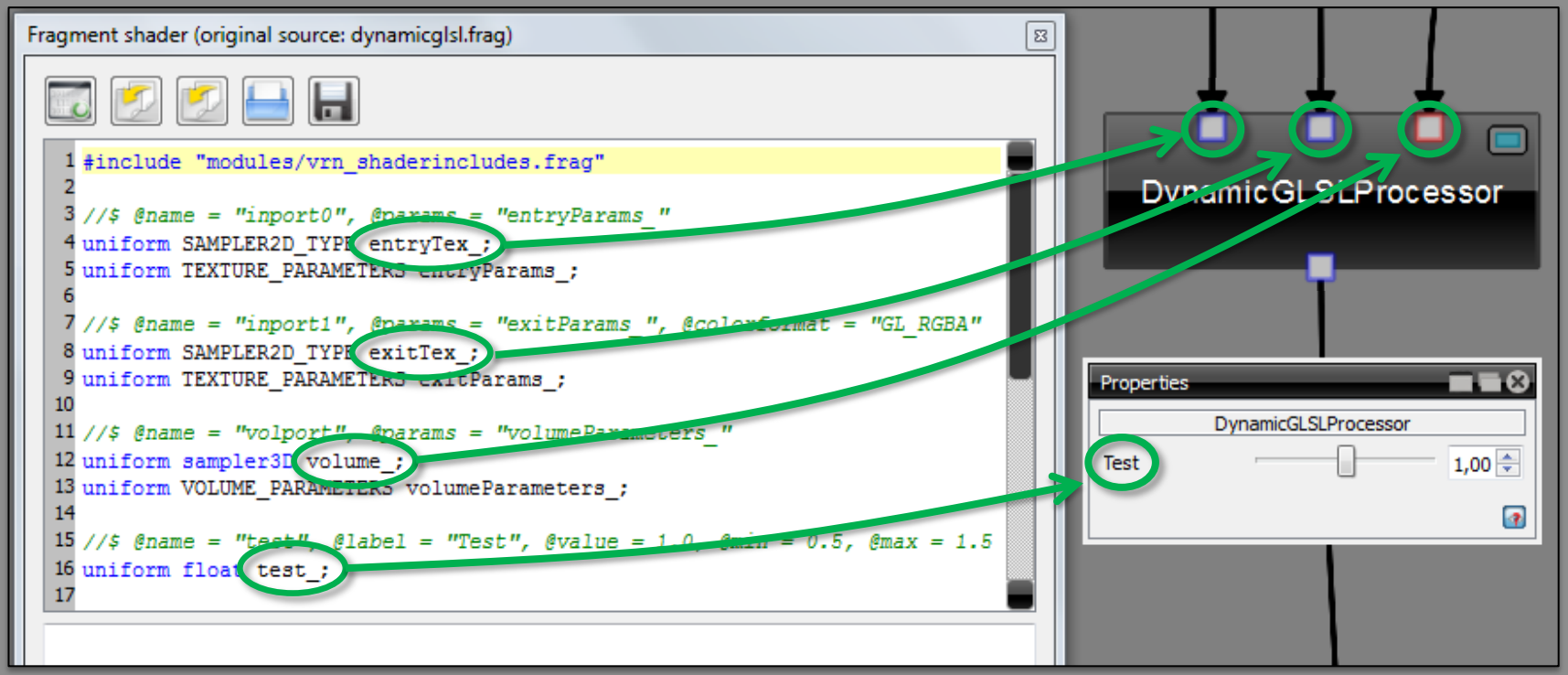
GLSL Volume Raycaster

- The **`GLSLRaycaster`** processor can be used to develop raycasting techniques during runtime



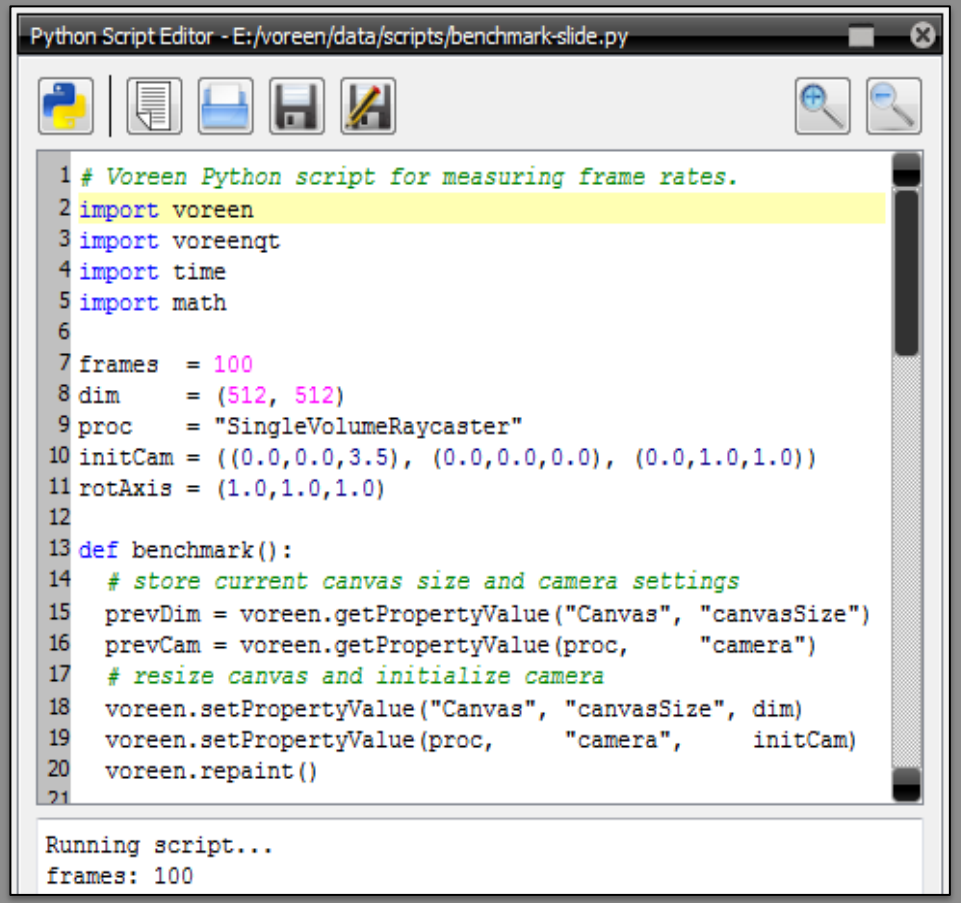
Dynamic GLSL Processor

- Parses its shader and dynamically adds ports and properties representing declared uniforms



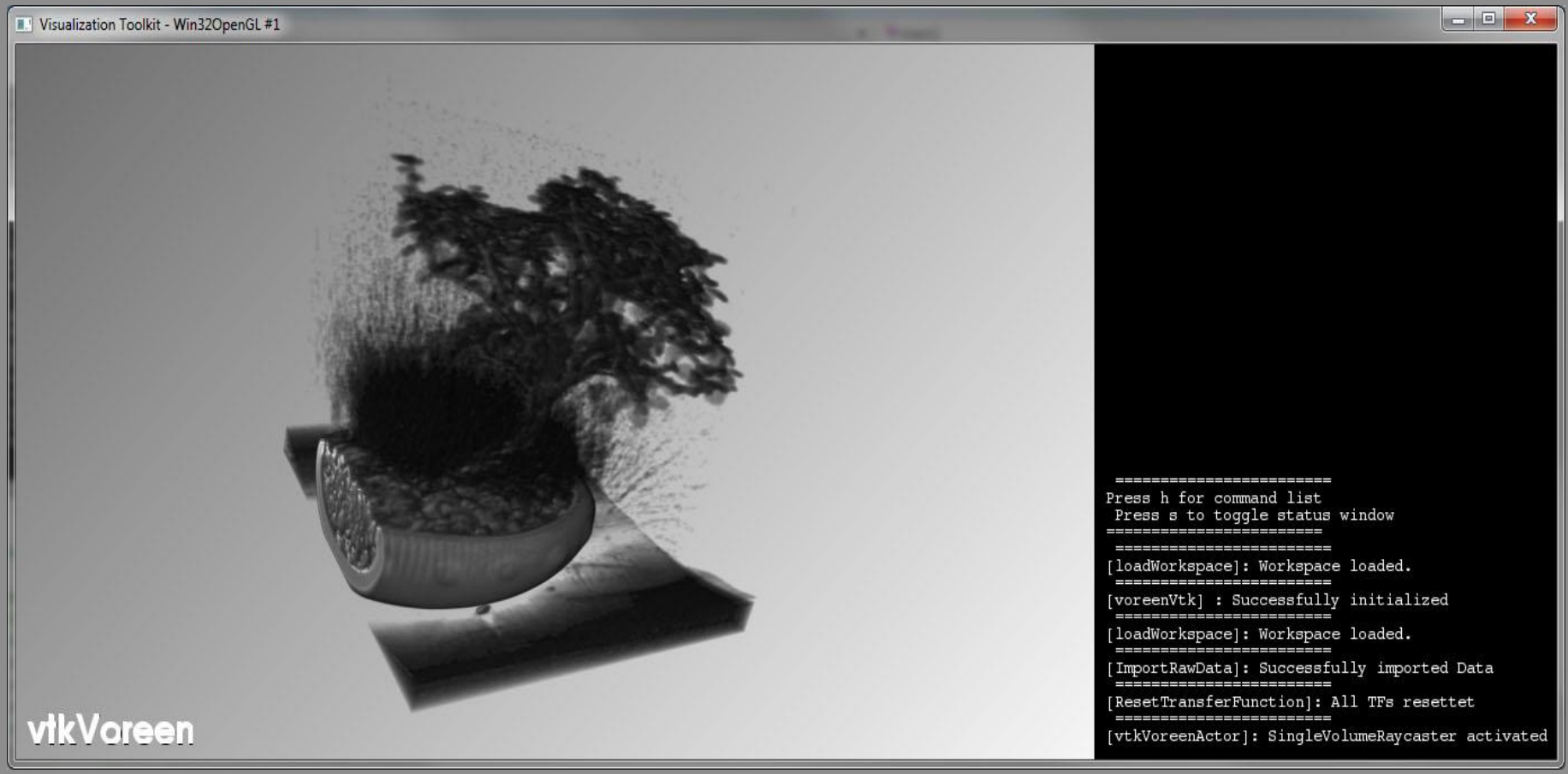
Python Scripting

- Generic read/write access to almost all types of properties, including cameras
- Volume and transfer function loading
- Canvas snapshots
- Integrated Python editor



Demo

vtkVoreen

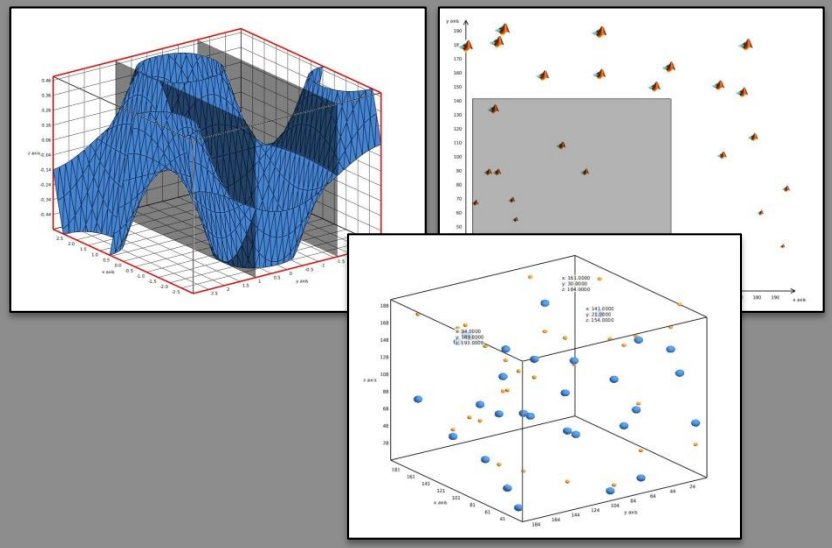

Visualization Toolkit - Win32OpenGL #1
vtkVoreen
Press h for command list
Press s to toggle status window
[loadWorkspace]: Workspace loaded.
[voreenVtk] : Successfully initialized
[loadWorkspace]: Workspace loaded.
[ImportRawData]: Successfully imported Data
[ResetTransferFunction]: All TFs resettet
[vtkVoreenActor]: SingleVolumeRaycaster activated

Outlook

The future of Voreen

Outlook

- ~~Plotting~~ (released in version 2.6)
- Kiosk solution
- Rapid-prototyping
  - ~~GLSL parsing~~
  - OpenCL kernel editing
- Volume processing
  - More processors
  - Volume caching
- Documentation, documentation, documentation…



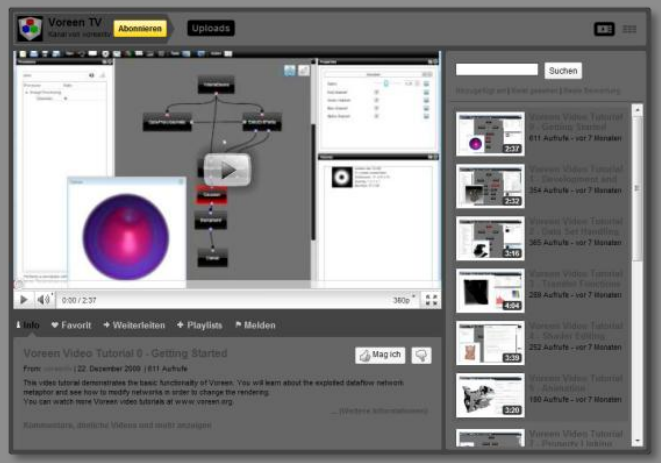
Where to go from here

Getting some more information

Where to go from here

- Everybody
  - www.voreen.org (Voreen, data sets, workspaces)
  - Contact the Voreen mailing list
- As a developer
  - API Documentation
  - Programming tutorials
- As a user
  - Youtube channel (*voreentv*)

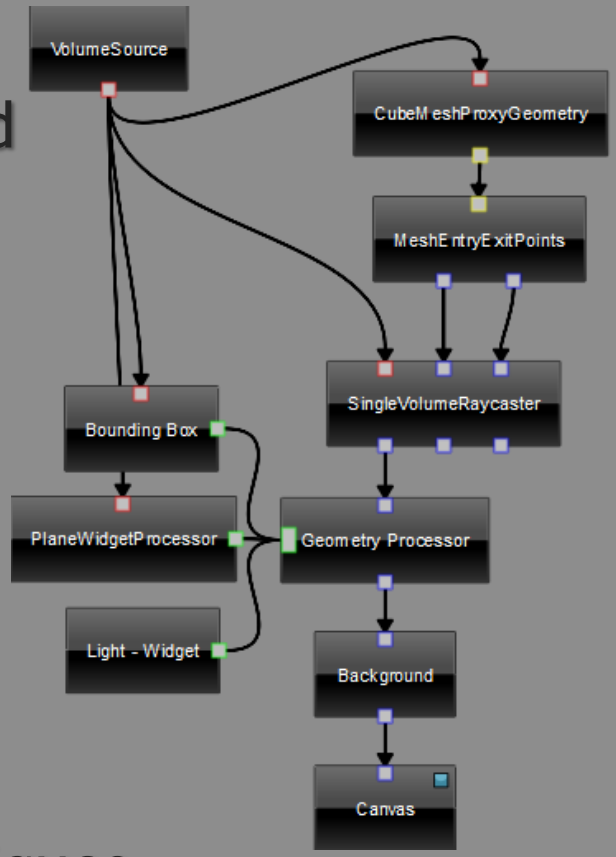
[www.voreen.org]



[www.youtube.com/voreentv]

Conclusions

- Key concepts
  - Data-flow realization of GPU-based volume ray-casting
  - Properties can be combined with linking
  - Development vs. Application Mode



- Specific functionality
  - Enable rapid prototyping of volume visualizations
  - Allow reusability of existing techniques
  - Support application deployment

Acknowledgments

- The Voreen team (alphabetically)
  - Alexander Bock, Benjamin Bolte, Stefan Diepenbrock, Christian Döring, Jan Esser, André Exeler, Dirk Feldmann, Alejandro Figueroa Meana, Timo Griese, Dieter Janzen, Jens Kasten, Daniel Kirsch, Rico Lehmann, Roland Leißa, Florian Lindemann, Markus Madeja, Jörg Mensmann, Jennis Meyer-Spradow, Borislav Petkov, Jörg-Stefan Praßni, Stephan Rademacher, Rainer Reich, Mona Riemenschneider, Timo Ropinski, Christoph Rosemann, Jan Roters, Sönke Schmid, Michael Specht, Fabian Spiegel, David Terbeek, Christian Vorholt, Carolin Walter, Michael Weinkath, Frank Wisniewski

Thanks for your attention!

www.voreen.org

These slides are available at
www.voreen.org/420-Tutorial-Slides.html