GPU Volume Ray Casting of two Volumes within VTK

Release 0.00

Karl Krissian, Carlos Falcón-Torres

April 16, 2012

Universidad de Las Palmas de Gran Canaria
departamento de Informática y Sistemas, Spain. [1]

Abstract

We have modified the current VTK volume rendering on GPU to allow simultaneous rendering of two volumes, each of them with its own color and opacity transfer functions. These changes have led to the creation of two new C++ classes and several GLSL shaders. We explain the modifications made to the original classes and shaders and we discuss possible additional improvements. A C++ demo code shows how to use the new classes.

Contents

[1]This work has been supported by the project TIN2009-10770 (Spanish Ministry of Science and Innovation).

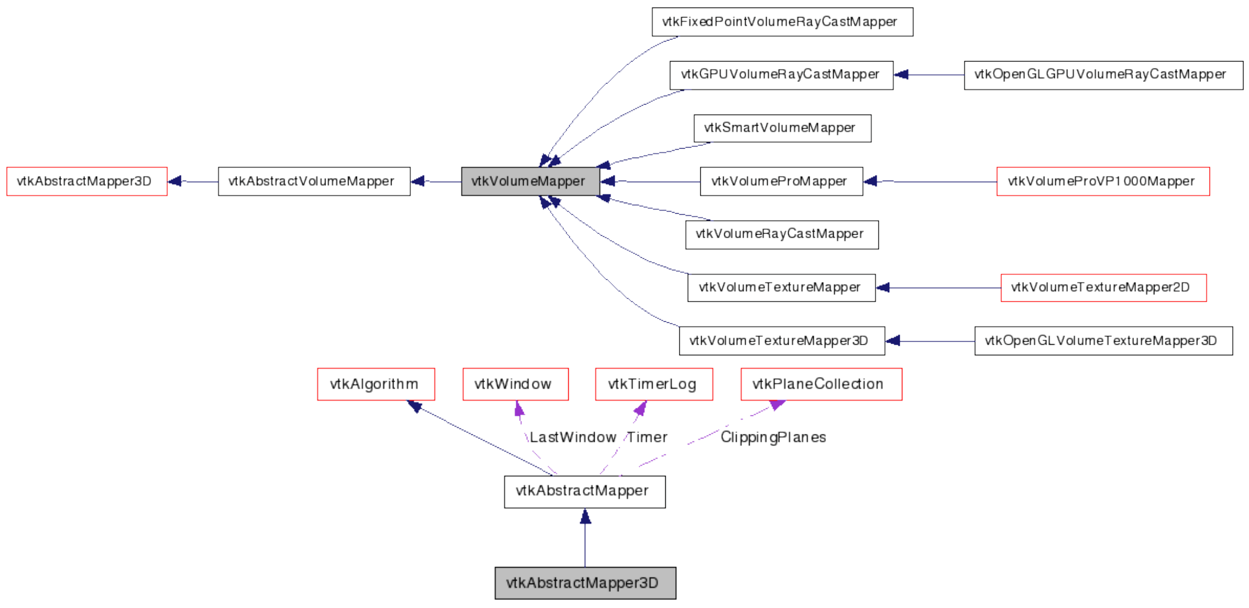

Figure 1: Main C++ classes for VTK volume rendering.

Volume rendering is a technique to project three-dimensional datasets on a plane, allowing a visualization of the whole information of the volumes at once. The volumetric datasets contain one (or several in case of RGB color) scalar component(s) per voxel, and are projected using two transfer functions for color and for opacity mapping, usually defined as functions of the image intensity. The Visualization ToolKit [2] offers several versions of Volume Rendering, within different classes and a common framework, defined by the vtkVolumeMapper abstract class. The two main volume rendering techniques, Ray Casting and Texture Mapping are implemented within different classes:

- Ray casting classes: vtkFixedPointVolumeRayCastMapper, vtkOpenGLGPUVolumeRayCastMapper, vtkVolumeProVP1000Mapper and vtkVolumeRayCastMapper.
- Texture mapping classes: vtkVolumeTextureMapper2D (with MesaGL or OpenGL) and vtkOpenGLVolumeTextureMapper3D.

Additionally, the class vtkSmartVolumeMapper is able to switch between different volume rendering technique based on the hardware configuration and on the required frame rate and user task: for example, it may use texture mapping during mouse events like scene rotation and ray casting when the interaction ends. As a general rule, ray casting can provide better resolution at a higher computational cost as compared to texture mapping techniques.

One of the most attractive of these classes is vtkOpenGLGPUVolumeRayCastMapper since it combines speed and accuracy by taking advantage of the power of the Graphics Processing Unit (GPU) of the graphic card. The difference between, for example, the use of vtkFixedPointVolumeRayCastMapper and vtkOpenGLGPUVolumeRayCastMapper is that the latter enables high resolution of the rendering during interactive tasks like the rotation of the scene or camera.

Algorithm 1 Main VTK volume rendering scheme.

```
1: Create renderer                                  ▷ as a vtkRenderer
2: Create volume                                    ▷ as a vtkVolume
3: Create mapper                                    ▷ as a vtkOpenGLGPUVolumeMapper
4: Create colorFun                                  ▷ as a vtkColorTransferFunction
5: Create opacityFun                                ▷ as a vtkPiecewiseFunction
6: Create property                                  ▷ as a vtkVolumeProperty
7: property.SetColor( colorFun )
8: property.SetScalarOpacity( opacityFun )
9: volume.SetProperty( property )
10: mapper.SetInputData(input)
11: volume.SetMapper( mapper )
12: renderer.AddVolume( volume )
```

It can be convenient for different purposes like registration of datasets (multi-modal or mono-modal) to include and visualize several volumes in the same scene. Unfortunately, the current classes of VTK don't allow the correct combination of various volumes in the same scene with GPU volume rendering. We propose to modify the current VTK code to enable this feature for two volumes.

1 Design of the C++ classes

The current framework to display a volume with volume rendering in VTK is summarized in algorithm 1.The volume object contains on one side the properties, including the color and opacity transfer functions, and on the other side the volume mapper, which contains the input dataset. We renamed vtkGPUVolumeRayCastMapper and vtkOpenGPUVolumeRayCastMapper classes to vtkGPUMultiVolumeRayCastMapper and vtkOpenGPUMultiVolumeRayCastMapper and proposed the modifications showed in figure 2. Since the combination of the different volumes of the scene has to be processed at the GPU level, they need to be accessible from the same vtkVolumeMapper. Concerning the transfer functions, each volume needs its own vtkVolumeProperty instance. The vtkVolumeProperty of the first volume is passed to the mapper by passing the instance of the vtkVolume as a parameter. We decided to include directly in the new mapper all the necessary parameters to render the second volume: the vtkDataSet instance and the vtkVolumeProperty instance. Another option would have been to follow the scheme of figure 3, however it requires two additional classes: vtkMultiVolume that would inherit from vtkVolume and vtkMultipleVolumeMapper that would inherit from vtkVolumeMapper. This option may be implemented in future releases. The usage of the new class vtkOpenGLGPUMultiVolumeMapper is summarized in algorithm 2.

2 Main Changes in the code

2.1 C++ classes

The two new classes that we created are vtkGPUMultiVolumeRayCastMapper and vtkOpenGLGPUMultiVolumeMapper, based on the original VTK classes vtkGPUVolumeRayCastMapper and vtkOpenGLGPUVolumeMapper. In this section, we review the main changes that have been applied to these classes.

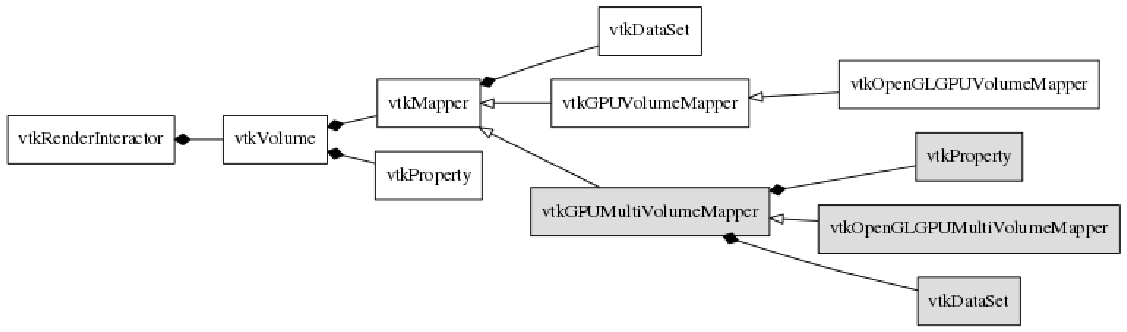

Figure 2: Proposed new classes.

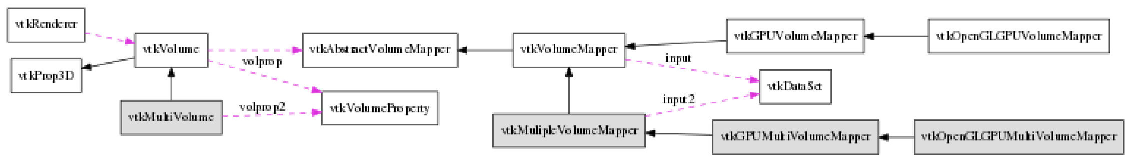

Figure 3: Other possible design of the classes.

vtkGPUMultiVolumeRayCastMapper

We added functions members to set and get several input datasets, and to set and get the volume property of the second dataset.

```
// Define the Input for both datasets
void SetInput( int port, vtkImageData *input );
void SetInput( int port, vtkDataSet *genericInput );
vtkImageData * GetInput( int port=0);

// set/get the properties of the second volume
void SetProperty2(vtkVolumeProperty *property);
vtkVolumeProperty *GetProperty2();
```

Additionally, a member TransformedInput2 is created in the same way as TransformedInput, which, according to the code comments, allows to deal with the limitation of GPUVolumeRayCastMapper in handling extents starting from non zero values. The corresponding changes are included in `vtkGPUMultiVolumeRayCastMapper::ValidateRender(...)`.

vtkOpenGLGPUMultiVolumeRayCastMapper

In the proposed implementation, this class will deal with two required input datasets, and the feature of vtkOpenGLGPUVolumeRayCastMapper to allow a mask dataset has been disabled. Each dataset will be sent as a 3D texture to the graphic card, with its own texture coordinates. We have added a class member TextureCoord_1to2, of type vtkTransform, that computes the transform matrix needed to convert texture coordinates of the first volume to texture coordinates of the second volume. This matrix and its inverse are passed to the shaders. The structure of vtkOpenGLGPUMultiVolumeRayCastMapper is the same as the previous vtkOpenGLGPUVolumeRayCastMapper class with main implementation

Algorithm 2 VTK volume rendering with two volumes.

```
 1: Create renderer                                   ▷ as a vtkRenderer
 2: Create volume                                     ▷ as a vtkVolume
 3: Create mapper                                     ▷ as a vtkOpenGLGPUMultiVolumeMapper
 4: for $n = 0$ to 1 do
 5:     Create colorFun[n]                            ▷ as a vtkColorTransferFunction
 6:     Create opacityFun[n]                          ▷ as a vtkPiecewiseFunction
 7:     Create property[n]                            ▷ as a vtkVolumeProperty
 8:     property[n].SetColor( colorFun[n] )
 9:     property[n].SetScalarOpacity( opacityFun[n] )
10: end for
11: mapper.SetInput( 0, input1 )                      ▷ set the first dataset
12: mapper.SetInput( 1, input2 )                      ▷ set the second dataset
13: mapper.SetProperty2( property[1] )                ▷ set the properties of the second dataset
14: volume.SetProperty( property[0] )
15: volume.SetMapper( mapper )
16: renderer.AddVolume( volume )
```

changes in the following methods: `LoadScalarFields(...)`, `UpdateOpacityTransferFunction(...)`, `RenderClippedBoundingBox(...)`, `RenderSubVolume(...)` and `BuildProgram(...)`.

The OpenGL texture indexes relative to the datasets and their transfer functions are:

TEXTURE	0	1	2	···	7	8	9
	1st dataset	1st Color	1st Opacity	···	2nd dataset	2nd Color	2nd Opacity

2.2 Shaders

Based on the original shaders, we create the following ones: vtkGPUMultiVolumeRayCastMapper_{Shade,Composite,FourComponents,NoShade,OneComponent}FS.glsl

vtkGPUMultiVolumeRayCastMapper_ShadeFS.glsl

In `InitShade()`, the increments in directions x,y,z are computed for the second volume using the P1toP2 matrix, that transform texture coordinates of the first volume to texture coordinates of the second volume.

A new function `vec4 Shade2(vec4 value)` is created similar to `Shade()`, that computes the gradient of the second volume intensity at the current 3D position of the ray, in the space coordinates of the first volume (it uses {x,y,z}vec2 computed in `InitShade()`). The same lighting parameters are used for both volumes.

vtkGPUMultiVolumeRayCastMapper_CompositeFS.glsl

In this shader, we use the two input 3D textures from the two input datasets. The position of the ray in the texture coordinates of the first dataset: *pos* is transformed to the coordinates of the second dataset: *pos2*. If *pos2* falls within the limits of the second dataset given by the variables *lowBounds2* and *highBounds2*, then voxel of the second volume is added to the ray using its computed color and opacity, as if it was just behind the corresponding voxel of the first volume, as described by the following code:

```
    if (all(greaterThanEqual(pos2,lowBounds2))
         && all(lessThanEqual(pos2,highBounds2)))
      {
        vec4 value2=texture3D(dataSetTexture2,pos2);
        float scalar2=scalarFromValue(value2);
        // opacity is the sampled texture value in the 1D opacity texture at
        // scalarValue
        opacity2=texture1D(opacityTexture2,scalar2);
        if(opacity2.a>0.0)
        {
          text2=true;
          color2=shade2(value2);
          color2=color2*opacity2.a;
          destColor=destColor+color2*remainOpacity;
          remainOpacity=remainOpacity*(1.0-opacity2.a);
        }
      }
```

Additional minor changes have been applied to the *FourComponents*, *NoShade* and *OneComponent* shaders.

2.3 Applying a transformation matrix

We have included the possibility to apply a user supplied transformation matrix to the second volume. This transformation can be set and get using the two following methods of the *vtkGPUMultiVolumeRayCastMapper* class: `void SetSecondInputUserTransform(vtkTransform *t);` and `vtkTransform *GetSecondInputUserTransform();`.

The given transformation, denoted T_2 is then included in the matrix *TextureCoord_1to2*, denoted TC_{12}, that converts texture coordinates from the first dataset to the second dataset, using the following equation:

$$TC_{12} = M_2.T_2^{-1}.M_1^{-1}, \tag{1}$$

where M_1 denotes the transformation from the world coordinates to the first texture coordinates, and M_2 denotes the transformation from the world coordinates to the second texture coordinates. This matrix is computed in the *RenderClippedBoundingBox* method and is passed to the shaders with the name ”P1toP2”, its inverse is passed to the shaders as ”P2toP1”.

3 Example

We include a modified version of the *vtkGPUVolumeRenderDemo.cxx*, named *vtkGPUMultiVolumeRenderDemo.cxx* where we show how to use the proposed class with two volumes. A *vtkBoxWidget* is used to control in real-time the transformation of the second volume. The code is also included as a module of AMILab [1] as show in figure 4. A video is also available at http://youtu.be/izaabm1Qq2o.

4 Conclusion and Discussion

We have proposed a modified version of the current GPU volume rendering available in VTK to allow the simultaneous rendering of two input datasets. Changes in the code include changes in the VTK classes and

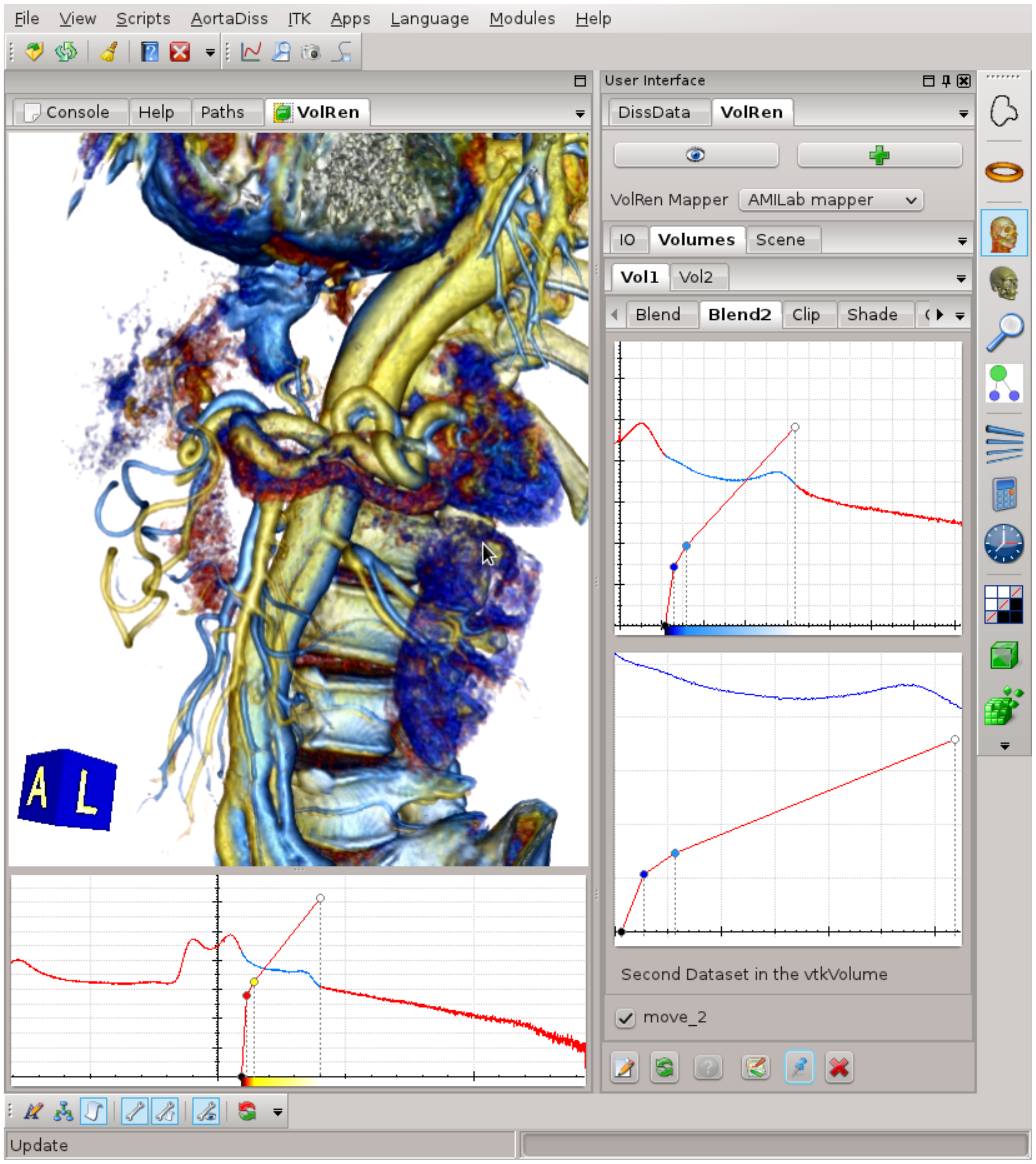
File View Scripts AortaDiss ITK Apps Language Modules Help
Console Help Paths VolRen
User Interface
DissData VolRen
VolRen Mapper AMILab mapper
IO Volumes Scene
Vol1 Vol2
Blend Blend2 Clip Shade
Second Dataset in the vtkVolume
move_2
Update

Figure 4: Example of Rendering from two Computed Tomography datasets using the proposed VTK class within AMILab.

in some of the GLSL shaders. The current VTK class design is not well suited to include several volumes with the same mapper since the vtkVolume class, which includes both the mapper and the properties, is designed to contain one dataset. Thus, we decided to duplicate the current GPU volume mapper and to allow it to have a second dataset and a second vtkProperty instance, minimizing the changes that we apply to the initial VTK classes and design. However, this choice is still open to improvements and another class design is suggested in figure 3. The proposed implementation is limited to two input datasets, using more inputs would require implementing additional shaders that can deal with the given number of inputs, and the generalization to any number of input datasets would require sending several arrays of 3D and 2D textures to the graphic card for the input datasets and their associated transfer functions, and further work is needed to check if the shader language allows it and if it is a interesting extension to the proposed implementation. Another limitation is that we currently limit the 3D scene to the extents of the first volume. This limitation could be overcome by projecting the bounding boxes of each volume and with additional changes in the shaders.

References

[1] K. Krissian, F. Santana, D. Santana-Cedrés, C. Falcón-Torres, S. Arencibia, S. Illera, A. Trujillo, C. Chalopin, and L. Alvarez León. Amilab software: Medical image analysis, processing and visualization. In *MMVR NextMed Medicine Meets Virtual Reality*, volume 173 of *Studies in Health Technology and Informatics*, pages 233–237, February 2012. 3

[2] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. *The Visualization Toolkit*. Kitware, Inc., fourth edition, 2006. (document)