The project does a multi-volume ray casting based on the Visualization Toolkit. This document gives some information about the project. The project also includes a demo for testing pourpose.

a) Main Changes in vtkGPUVolumeRayCastMapper

Since the original *vtkGPUVolumeRayCastMapper* is designed for single volume rendering, it uses only one volume dataset and one volume property as its input which are connected to *vtkVolumeMapper* and *vtkVolume* respectively (see fig. 1).

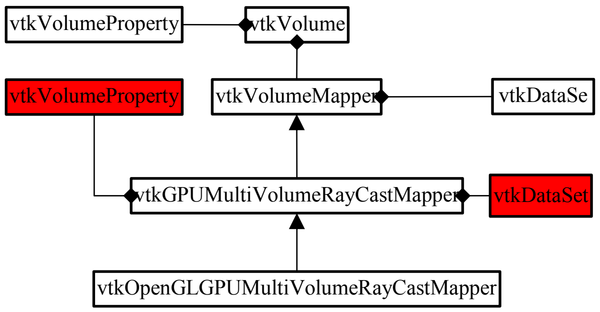

Figure 1 Relations between the new mappers and other classes in the proposed multi-volume ray caster

The first change to have a multi-volume renderer is that the mapper must have direct access to all volume properties and datasets. Hence *vtkVolumeProperty* and *vtkDataSet* are connected to the new *vtkGPUVolumeRayCastMapper* for rendering all input volumes. Figure 1 shows the connections between the new *vtkGPUVolumeRayCastMapper and vtkVolumeProperty* as well as *vtkDataSet*.

New class members have been added to *vtkGPU-MultiVolumeRayCastMapper* for setting and getting the additional datasets and properties. Table 1 shows the function prototype of new methods in the class.

Table 1 New methods for v*tkGPUMultiVolumeRayCastMapper*

//Define the Input for additional datasets.	

Set	void SetInput(int port, vtkDataSet *genericInput);
	void SetInput(int port, vtkImageData *input);

Get	vtkImageData * GetInput (int port=0);
//Define the properties of the additional volumes.	
Set	void SetAdditionalProperty (int volNumber, vtkVolumeProperty *property);
Get	vtkVolumeProperty *GetAdditionalProperty(int volNumber);
//Define user transformation for additional input user transform.	
Set	void SetAdditionalInputUserTransform(int volNumber, vtkTransform *t);
Get	vtkTransform *GetAdditionalInputUserTransform(int volNumber);

b) Main Changes in vtkOpenGLGPUVolumeRayCastMapper

vtkOpenGLGPUVolumeRayCastMapper is the core class of the ray casting method in VTK. This class is responsible for allocating and transferring the volume data sets into the GPU texture memory and afterward calls the GPU shaders with the appropriate parameters for rendering the volumes. Subsequently, the methods which are dealing with volume dataset and volume properties (color and opacity data) are changed to handle more than one volume. These methods are *LoadScalarFields*, *UpdateOpacityTransfer-Function*, *RenderSubVolume*, *BuildProgram* and *Render-ClippedBoundingBox*.

Another significant change in this class is adding a new class member called *TextureCoord_1toN[]* which is an array of *vtkTransform*. This array is used in *RenderClipped-BoundingBox* method and it holds transform matrices which converts the texture coordinates of the first volume to all the other volumes by applying the following equation for each additional volume:

$$\mathrm{TC}_{1toN} = M_N . UT_N^{-1} . M_1^{-1} \quad , N \geq 2$$

As the equation indicates, three components are used to create the final transformation matrix for each additional dataset. M_1 and M_N are the transformation from world coordinates to the first and N^th texture coordinates respectively. UT_N is an user defined transformation matrix and it gives the possibility of freely transforming (scaling, rotating and moving) each additional volume during runtime. UT_N can be updated and retrieved by using *SetAdditionalInputUserTransform()* and *GetAdditionalInputUserTransform()* which are implemented in *vtkGPUVolumeRayCastMapper*. The array *TextureCoord_1toN[]* is latter sent to the shader uniform variable with the name of *P1toPN[]*.

The initializing functions of clipping and cropping are also implemented in this class. The prototypes of these methods are presented in table 2. It is apparent from the signature of the functions that clipping and cropping can be separately applied to different volumes by user request. The user can assign up to six planes for cropping and also allocate a single clipping plane.

Table 2 Function prototype for implemented clipping and cropping methods in v*tkGPUVolumeRayCastMapper*

//Signature of clipping methods	

Clipping	void AddClippingPlane(int vol, vtkPlane *plane);
	void RemoveClippingPlane(int vol);
	vtkPlane* GetClippingPlane(int vol);

//Signature of cropping methods	

Cropping	void SetCroppingRegionPlanes(int vol, double xMin, double xMax, double yMin, double yMax, double zMin, double zMax);
	
	void SetCroppingRegionPlanes (int vol, double *args);
	void SetCropping (int vol,int arg);

Similar to other parts of the code, clipping and cropping functions perform initialization of the actual clipping and cropping; the core part of the clipping and cropping is implemented in shader code to be run on the GPU. For instance, *AddClippingPlane()* allocates and initialize a plane (consist of an origin point in space and a normal vector) for the selected volume and then just before the rendering of the volume, the initialized planes are converted from the world coordinate system to texture coordinate system. The converted planes are passed to the appropriate shaders by using the uniform variables. A similar procedure applies to *SetCroppingRegionPlanes()* for cropping.

c) Main changes in GLSL shaders

Nowadays, programmable GPUs are used for general algorithms that can be run in parallel. There are different ways to use GPUs for general purpose programming.. Using shaders is one of the very first methods that facilitate GPU programing. Shaders are basically small programs that can be used as part of the standard GPU pipeline. This enables users to use GPUs for more general purposes and design a shader according to their needs.

The VTK ray casting method also uses shaders for running the GPU codes. The benefits of using shaders compared to newer technologies like OpenCL are: 1) shaders are faster, and 2) it is supported by more devices (i.e. GPUs). Therefore, the proposed multi-volume ray caster is implemented using shaders. Table 3 illustrates the names of GLSL shaders which are changed to realize a multi volume ray caster; the first three shaders have minor changes and the last two have major changes (*vtkGPUVolumeRay-CastMapper_ShadeFS* and *vtkGPUVolumeRayCastMapper_-CompositeFS.)*

Table 3 List of changed GLSL shaders

	Shader Name	**Rate of Changes**
1	vtkGPUVolumeRayCastMapper_FourComponentsFS	Low
2	vtkGPUVolumeRayCastMapper_OneComponentFS	Low
3	vtkGPUVolumeRayCastMapper_NoShadeFS	Low
4	vtkGPUVolumeRayCastMapper_ShadeFS	Medium
5	vtkGPUVolumeRayCastMapper_CompositeFS	Medium

If shading of volumes is desired for the ray casting, *vtkGPUVolumeRayCastMapper_ShadeFS* performs the shading of the volumes. In the modified version of this shader, the shading method remains similar to the original VTK shader. The main change in this shader is applied in the *InitShade()* function; by using *P1toPN* (texture coordinate from the first volume to the other volumes) the increments of x, y and z are computed according to the corrected position of each volume.

vtkGPUVolumeRayCastMapper_CompositeFS is the shader which creates the final color for each pixel on the screen. The ray traversal, collecting samples and α-blending all take place in this shader. Moreover, since this shader has the authority for deciding which samples should be included in the final pixel color, clipping and cropping are implemented on this shader.

When all the needed variable and parameters are calculated by the rest of the program, the *trace()* function is called as the last function of ray casting method to calculate the final pixel color and pixel opacity (i.e. *gl_FragColor* and *gl_FragColor.a*) for all pixels in the viewport. Inside the function there is a while loop which does the ray traversal through all the volumes. The initial position of the ray for the first volume is already calculated and is available in the *pos* variable. The first change in order to achieve multi-volume rendering is to calculate the initial position of the ray for other volumes. This is achieved by multiplication of *P1toPN[]* (see the previous section) and *pos*; the results are kept in an array called *posX[]* for each extra volume. As the ray traverse through the volumes, *pos* and *posX[]* is updated according to the ray direction (*rayDir*). After the position of the ray for each volume is obtained, the position of the ray is tested against the volume bounding box to check whether the ray is still inside the volume or not. *lowBouands[]* and *highBounds[]* are keeping the bounding box informations of each volume (cropping of each volume is also attained by modifying those variables). Afterward, the clipping flags are tested, if the ray position is not inside the clipped part of the volume, the ray position of the volume is sampled from the first to the last volume and according to the front to back α-blending formula:

$$I(D)_{f-b} = \sum_{i=0}^{n} \alpha_i c_i \prod_{j=0}^{i-1} (1 - \alpha_j)$$