Adding Refined Isosurface Rendering and Shadow Mapping to vtkGPUVolumeRayCastMapper

Release 1.00

Imanol Herrera, Carlos Buchart and Diego Borro

October 4, 2012

CEIT and Tecnun (University of Navarra)

Abstract

In the medical simulation world the use of isosurfaces is a common action, as the information from some sources, i.e. CTs, is very well defined, and the isosurface can be easily set. Additionally, in any simulator shadows are a neccesary addition to increase the users immersion as well as its depth perception. Unfortunately, the Visualization Toolkit does not offer these features, and so in this paper a modified vtkGPUVolumeRayCastMapper is presented. This modified version allows volumetric isosurface rendering as well as receiving shadows from polygons using the usual pipeline.

Contents

1 Introduction

Volumetric rendering is a key technique in scientific and medical visualization. This rendering method enables the direct use of volumes as data for the rendering pipeline, avoiding the need of preprocessing and the possible loss of data and accuracy it could bring. This volumes usually contain one scalar per voxel, although is not completely uncommon to have up to three scalar per voxel. When the volume has only one scalar component transfer functions are used to give optical properties such as opacity and colour to the volume.

Volume rendering allows the inspection of the whole volume at once or, at least, in an interactive and intuitive mode. This makes it possible to inspect the results of a simulation as a whole, or see patients' internal organs without inspecting an MRI scan slice by slice. But sometimes the user might want to inspect certain specific values of the volume, such as the organs or bones from a CT scan or a shock wave in a simulation. Usually, in order to visualize said isosurfaces, an isosurface extraction method such as marching cubes would be used and a regular polygonal rendering would be performed with the result. Another method would be using a transfer function set to be completely opaque at the desired isosurface, but this method has the high computational cost of volume rendering without taking advantage of its strong points, i.e., the rendering of semitransparent volumes.

Another method for the visualization of isosurfaces is volumetric isosurface rendering. With this the extraction method is skipped and the volume is directly rendered by setting an isovalue. In addition, by using an intersection refinement method the step size can be increased with little or no quality loss. In this way the method offers the flexibility and quality of volume rendering without its high computational cost.

The Visualization ToolKit (VTK) offers a wide range of the said methods, but the lack of a volumetric isosurface rendering poses a problem in certain situations, such as a medical visualization using CTs. Moreover, VTK does not allow the volumes to receive the shadows cast by polygonal surfaces, a much needed feature when dealing with user's depth perception of such polygonal actors.

The lack of these features in VTK was made apparent when developing a medical simulator, for which isosurfaces and shadows were very desirable features. So, in order to fill this gap, in this paper we present a extension of the vtkOpenGLGPUVolumeRayCastMapper (called vtkOpenGLGPUVolumeRayCastMapper2 to allow compatibility and simultaneous use) which allows the use of a volumetric isosurface mode with intersection refinement and receiving shadows from polygonal actors without modifying the shadow rendering pipeline.

2 Changes on the design

The proposed changes do not require new classes by themselves, but in order to allow a easier usability and avoid potential problems with the different versions of the vtkOpenGLGPUVolumeRayCastMapper and vtkGPUVolumeRayCastMapper we have created two new classes that are basically an extension of the originals: vtkOpenGLGPUVolumeRayCastMapper2 and vtkGPUVolumeRayCastMapper2. We created both classes in order to maintain the usual distribution of code in the original classes and to maintain the same design as shown in Figure 1.

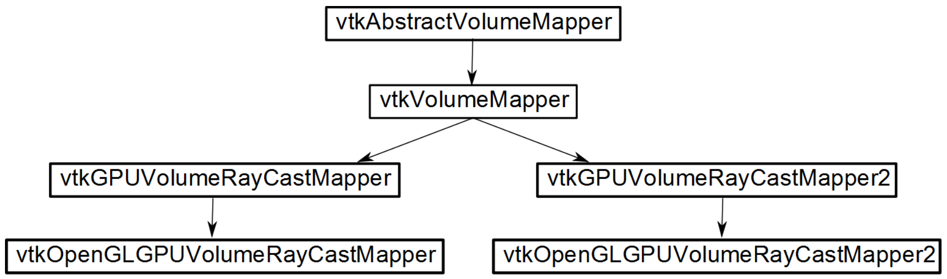
vtkAbstractVolumeMapper
vtkVolumeMapper
vtkGPUVolumeRayCastMapper
vtkGPUVolumeRayCastMapper2
vtkOpenGLGPUVolumeRayCastMapper
vtkOpenGLGPUVolumeRayCastMapper2

Figure 1: The resulting design with the new classes

To summarize the additions and changes done to the classes:

- A new rendering mode has been added, the volumetric isosurface mode. For this we created a new GLSL shader, which implements the isosurface ray traversal mode with the refinement. Additionally, a variable to set the isovalue has been added along with the methods to access and modify it.
- The shadow receiving mode has been added, which is controlled in the same way it is controlled for polygonal objects. Three new GLSL shaders have been created too, implementing different shadow modes: no shadows, hard shadows and soft shadows. Several variables and methods to access said variables have been added.
- Additionally, the already existing code for the stochastic jittering [1] technique has been uncommented and several variables and methods have been added to control it. Two GLSL shaders have also been created, stochastic jittering enabled and disabled. Note that the methods and variables for the stochastic jittering maintain the nomenclature from the original VTK class, using noise instead of stochastic jittering.

Examples for the use of the aforementioned changes can be found in Appendix A.

3 Features

In this section the added features will be explained more thoroughly and the implementation details will be explained.

3.1 Volumetric isosurface rendering

The ray traversal for the volumetric isosurface rendering is performed much as it would be in a usual volumetric rendering, with the only difference in the accumulation of the color and opacity. In the isosurface mode if a voxel's value is above the isovalue the ray traversal stops. In this point the point is already in the isosurface, but due to the sampling distance and discrete nature of the volume artifacts appear as shown in Figure 2.

To avoid this artifacts, intersection refinement [2] is performed. By using an iterative refinement method with the Equation 1 the intersection point can be easily refined.

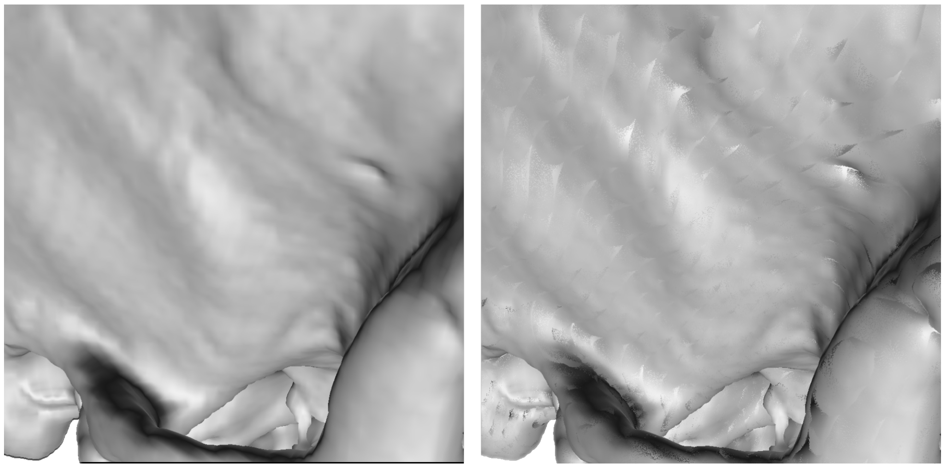

Figure 2: A comparison of isosurface rendering with (left) and without (right) intersection refinement.

$$X_{new} = (X_{next} - X_{prev}))\frac{isovalue - f_{prev}}{f_{next} - f_{prev}} + X_{prev} \tag{1}$$

With the new intersection point, the value of the new point is readed. That new value is then compared to the isovalue, if its bigger then X_{new} and the new value are assigned to X_{prev} and f_{prev} respectively. Otherwise, X_{new} and the new value are assigned to X_{next} and f_{next}. By iterating we are able to reach a much better intersection point. To perform the shading the isovalue is used to fetch the color in order to avoid artifacts due to the transfer function. Usually, using around 10 iterations gives a good enough point with very low computation cost. The results of the intersection refinement can be seen in the Figure 2. The pseudocode of the intersection refinement shader is shown in Appendix B.1.

3.2 Shadows

Shadows are a key feature to enhance both realism and depth perception as shadows help the user to locate the position of an object, i.e., a tool in a simulator. Thus they are a very desirable feature in many visualizations.

VTK includes a method to create a pipeline with shadows, using vtkShadowMapPass and vtkShadowMapBakerPass. In this pipeline, the polygonal actors are rendered as usual and after this the shadows are rendered onto the previous rendering.

When adding shadows to the volume, if the usual pipeline were to be used, the volume would have to been rendering twice, with the evident result in performance loss. In order to avoid this loss, a more common approach was taken. Instead of rendering the shadows separately, the shadowing is checked when the ray traversal is performed. As a result, shadows from the polygons can be cast onto the volume with little additional computational cost, resulting in an improved depth perception, as shown in Figure 3.

4 Conclusion

As it has been previously said, using a proper opacity transfer function, an approximate rendering result can be obtained, but this result will heavily depend on the sample distance, as the volume rendering usually

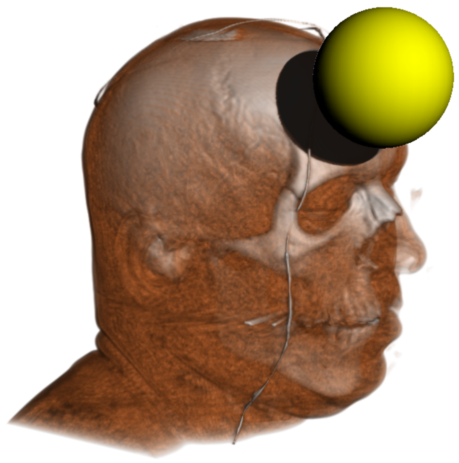
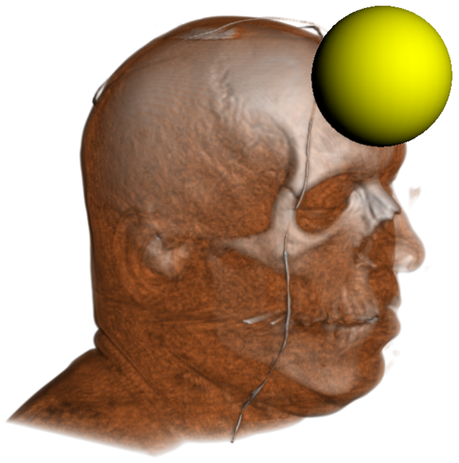

Figure 3: A comparison of rendering with (left) and without (right) shadows.

does. This makes isosurface rendering not efficient, as it needs a small sample distance to achieve good quality renderings.

On the other hand the isosurface rendering method allows, thanks to the refinement, to increase the sample distance greatly without having much impact on the visual quality, as shown in the Figure 4. Thanks to this, we are able to greatly decrease the needed rendering time for a high quality rendering.

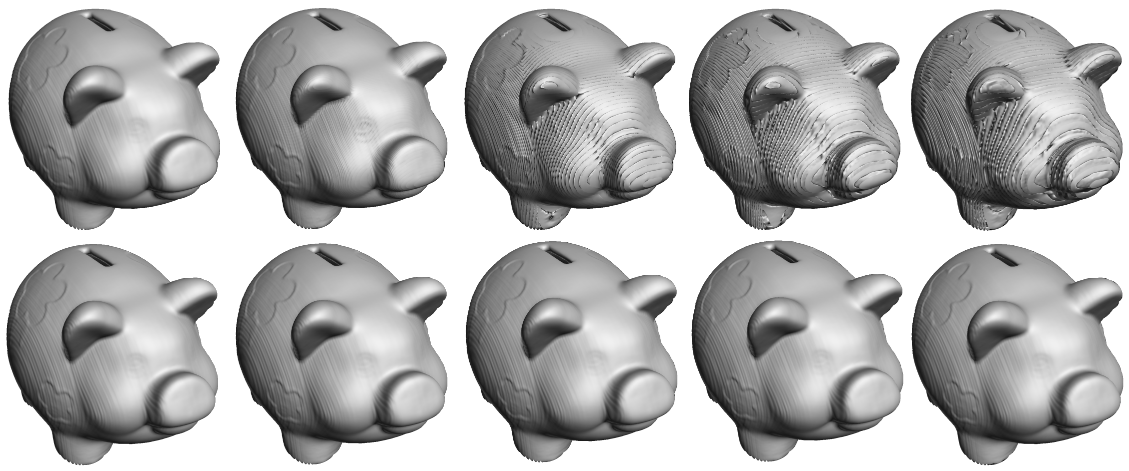

Figure 4: A comparison with different sample distances between an isosurface performed with the transfer functions (up) and a volumetric isosurface rendering with refinement (down). The sample distances are, from left to right, 0.1, 1.1, 2.1, 3.1 and 4.1

When isosurface rendering is not used, is not possible to refine the intersection thus errors in the visualization are inevitable without reducing the sample distance. But the artifacts can be rendered less visible by using the stochastic jittering. The stochastic jittering introduces a controlled noise in the starting points of the rays, transforming the wood-grain artifacts into noise, which is visually less noticeable. The code and implementation of this method is not explained in this paper as it was already coded in the class, it only needed to activate it and add control variables. More information on stochastic jittering can be found in [1]. Additionally the volumetric rendering has been enabled to receive shadows, increasing realism and depth perception.

The resulting volumetric mapper retains the speed and versatility of the original while adding the new features and options, which we esteem as desirable features in VTK.

5 Future work

- Right now the mapper only uses one light for the illumination, having all or more light into account may be a desirable expansion
- The shadows are only received from one light, the same as the lighting, and it may be desirable to expand this along the number of lights
- Although volumes can receive shadows, they do not cast shadows. It is a desirable feature to expand this mapper and the VTK passes for the shadows in order to allow the use shadows from the volumes.

Future releases will look into these features as well as other possible improvements over the existing code.

A Usage examples

Along with the source code of the new classes, a example application is provided, where the examples presented below are demonstrated. The minimum version of VTK to use these classes is 5.8, as the shadows pipeline was changed from 5.6 to 5.8.

A.1 Isosurface mode usage

The usage method for this rendering mode slightly differs from the usual choice of blending mode in volumetric rendering. The reasoning behind this difference is that to maintain the usual pipeline, further changes should be made to other classes, and it was deemed undesirable at the current development stage. If in the future these classes are merged into VTK the usage method will change to the usual VTK mode. An example of how to set the isosurface to `value`:

```
vtkGPUVolumeRayCastMapper2 * mapper = vtkGPUVolumeRayCastMapper2::New();

...

mapper->SetIsovalue(value);
mapper->SetBlendModeIsosurface(true);
```

When `BlendModeIsosurface` is set to true, the blending mode chosen by the usual means will have no effect, and it will be always rendered as isosurface. To use any other blending mode, `BlendModeIsosurface` has to be set to false. Note that the original blending mode is not changed by changing `BlendModeIsosurface` so just setting it to false will force the mapper to fall back to the previous blending mode.

A.2 Receiving shadows

Enabling the shadows for the volume rendering is performed as usual in VTK. A rendering pipeline including `vtkShadowMapPass` and `vtkShadowMapBakerPass` must be set up, and the polygonal objects properties for the shadows are set in the original way. In order to enable the volume to receive shadows, a `vtkShadowMapBakerPass::RECEIVER` must be set (regardless of to 0 or 1). If in addition to shadows, soft shadows are wanted they must be activated by setting `SoftShadows` to true. The offset of the soft shadow is controlled with `SoftShadowsOffset` and it represents the offset in pixels. Note that although it represents pixels, it does not need to be integer as linear interpolation is used for the shadow texture.

```
vtkGPUVolumeRayCastMapper2 * mapper = vtkGPUVolumeRayCastMapper2::New();

...

vtkVolume * volume = vtkVolume::New;
volume->SetMapper(mapper);

vtkInformation * info = vtkInformation::New();
info->Set(vtkShadowMapBakerPass::RECEIVER(), 0);

volume->SetPropertyKeys(info);

//For soft shadows:
mapper->SetSoftShadows(true);
mapper->SetSoftShadowsOffset(1.0);
```

A.3 Using stochastic jittering (Noise)

To enable the use of stochastic jittering `Noise` must be set to true. Additionally, the noise texture size and the maximum inserted stochastic jittering can be controlled through `NoiseTextureSize` and `MaxNoiseFactor` respectively.

```
vtkGPUVolumeRayCastMapper2 * mapper = vtkGPUVolumeRayCastMapper2::New();

...

mapper->SetNoise(true);
mapper->SetMaxNoiseFactor(0.75);
mapper->SetNoiseTextureSize(64);
```

B Pseucode of the changes

B.1 Isosurface shader

Here the pseudocode for the isosurface rendering with intersection refined is presented.

```
while $InsideTheVolume(CurrentPoint)$ do
  $Value = ValueFromPoint(CurrentPoint)$
  if $Value \geq Isovalue$ then
    for $i = 0 \rightarrow 10$ do
      $NewPoint = (CurrentPoint - PreviousPoint) * \frac{Isovalue - PreviousValue}{CurrentValue - PreviousValue} + PreviousPoint$
      $NewValue = ValueFromPoint(NewPoint)$
      if $NewValue < IsoValue$ then
        $PreviousValue = NewValue$
        $PreviousPoint = NewPoint$
      else
        $CurrentValue = NewValue$
        $CurrentPoint = NewPoint$
      end if
    end for
    $Shade(IsoValue, NewPoint)$
    $return$
  end if
  $CurrentPoint = NextPoint$
end while
```

References

[1] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 2, 4

[2] Markus Hadwiger, Patric Ljung, Christof Rezk Salama, and Timo Ropinski. Advanced illumination techniques for gpu volume raycasting. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 1:1–1:166, New York, NY, USA, 2008. ACM. 3.1