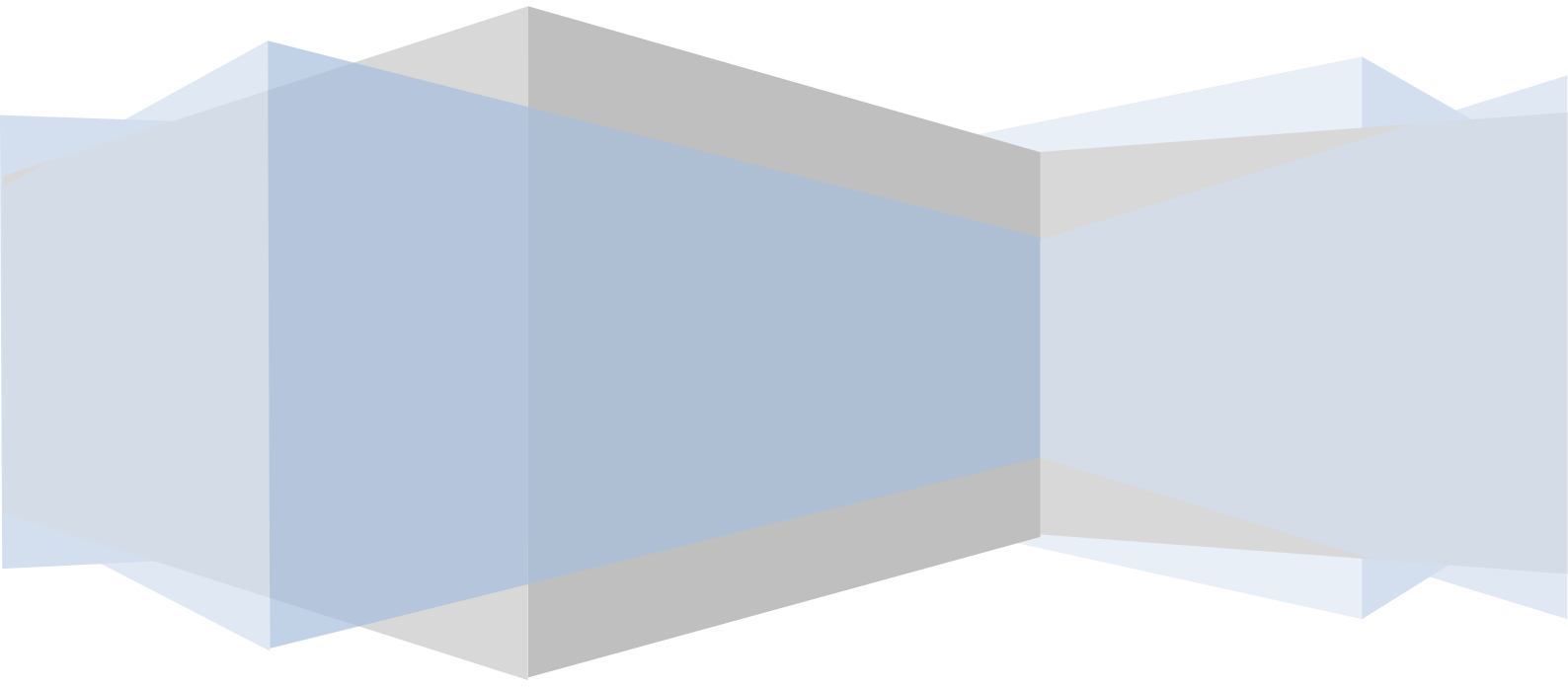
Direct Rendering of Curvilinear Volumes

Final report

Olivier Jais-Nielsen



PLAN

DIRECT RENDERING OF CURVILINEAR VOLUMES

1. INTRODUCTION

The purpose of this project is to provide a software able to display in an interactive way volumetric data sampled along cylindrical or spherical coordinates. Such a dataset consists in a three-dimensional numerical array. For a cylindrical geometry, the three directions correspond to the radial distance, the angular position and the axial position (cf. Figure 1).

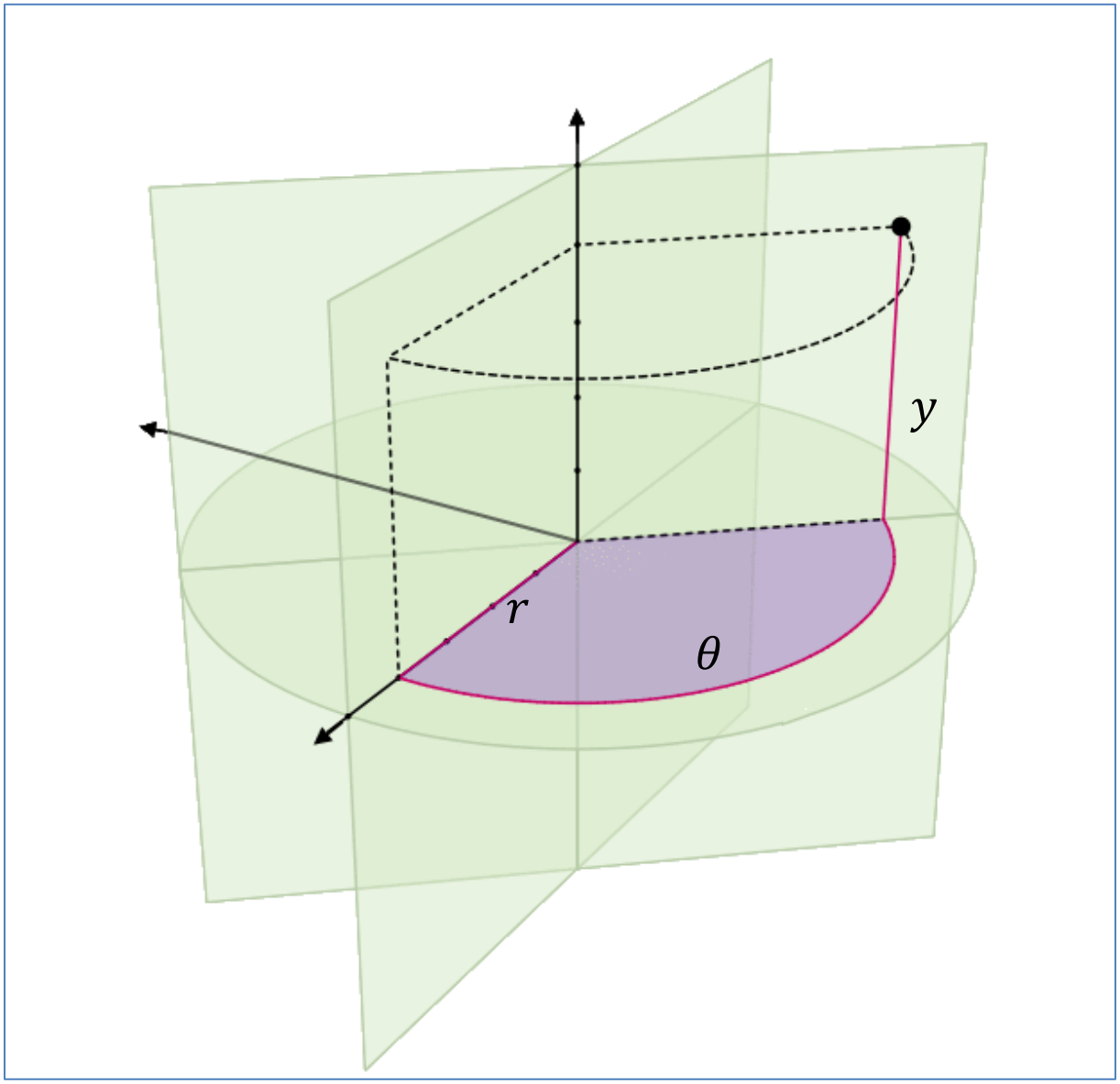


Figure 1 - A cylindrical coordinate system (1). r represents the radial distance, θ the angular position and y the axial position.

For a spherical geometry, the directions correspond to the radial distance, the longitude and the latitude (cf. Figure 2).

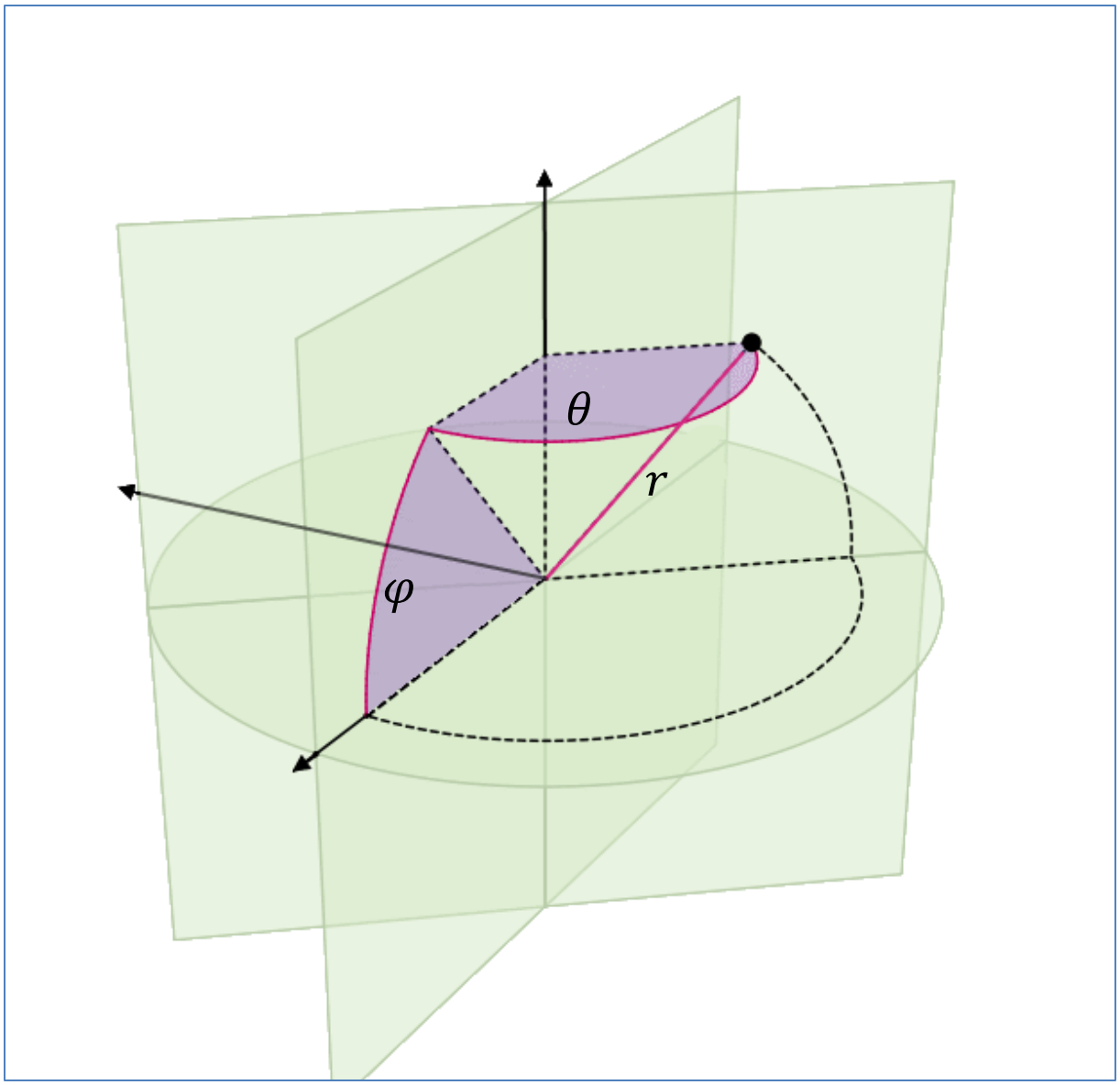


Figure 2 - A spherical coordinate system (2). r represents the radial distance, θ the longitude and φ the latitude.

The three-dimensional array is referred to as the *beam space*, a *beam* corresponding to a line defined for a particular angular position and a particular axial position in the cylindrical case and a particular longitude and a particular latitude in the spherical case. Therefore, the covered volume is sampled by the beams and the numerical values are sampled along the beams.

The actual volume is referred to as the *real space.* The goal is to visualize on a computer screen the real space using *volume ray casting*. Volume ray casting consists in defining a point of view in space and a rectangular portion of a plane in space, the *viewport*. The viewport is sampled in pixels. For each pixel, we consider the ray going from the point of view throw the pixel location. If this ray intersects the volume, equidistant sampling points are selected along the part of the ray that lies within the volume (cf. Figure 3). As the sampling points might lie between voxels, their values are interpolated from the neighboring voxel values. The final color value associated to the current pixel is computed from the values at the sampling points; this is the *composition*. Finally the viewport is displayed as an image. In practice, the point of view is at a fixed distance "behind" the center of the viewport. To visualize the volume from different distances and angles, both the point of view and the viewport are moved to keep their relative positioning.

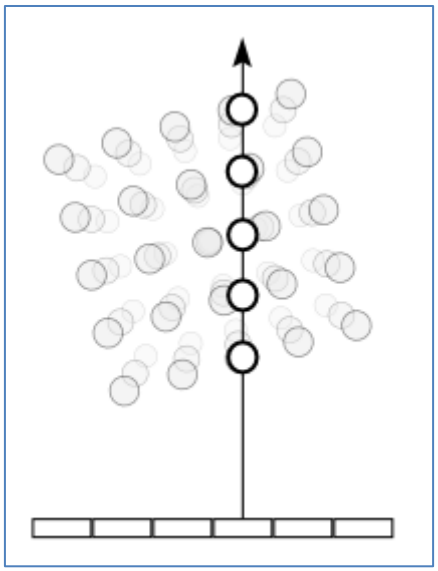


Figure 3 - Volume ray casting on a quadratic regularly sampled volume (3).

The values of all the pixels are computed in the same way, independently from one another. This is characteristic of high data parallelism. Therefore, we implement this software using Graphics Processing Units (GPU) to get real-time rendering.

2. IMPLEMENTATION DESCRIPTION

2.1. SOFTWARE USE

The software takes as input dataset Visualization ToolKit (VTK) files complying with the following criteria:

- The data should be a set of scalars.
- The data should be stored in a lookup table.
- The data should be 8-byte floating point numbers (*double*) or 2-byte positive integers (*unsigned short*).
- The data should be stored as binary using the big-endian convention.

Once the dataset is opened with the software, the type of geometry (cylindrical or spherical) is specified by the user as well as the angles characterizing the volume geometry:

- For a cylindrical geometry, the total angle is specified.
- For a spherical geometry, the longitude is specified.

The dataset is interpreted as follows:

- For a cylindrical geometry, the first dimension is mapped to the radial distance, the second to the angular position and the third to the axial position.
- For a spherical geometry, the first dimension is mapped to the longitude, the second to the latitude (it is additionally shifted so that if φ is the total latitude, the values are mapped from $-\varphi/2$ to $\varphi/2$) and the third to the radial distance.

Additionally to the command line window used to enter the preceding parameters, the software is composed of two windows: one displays the result and the other allows the user to change some parameters concerning the visualization (cf. Figure 4).

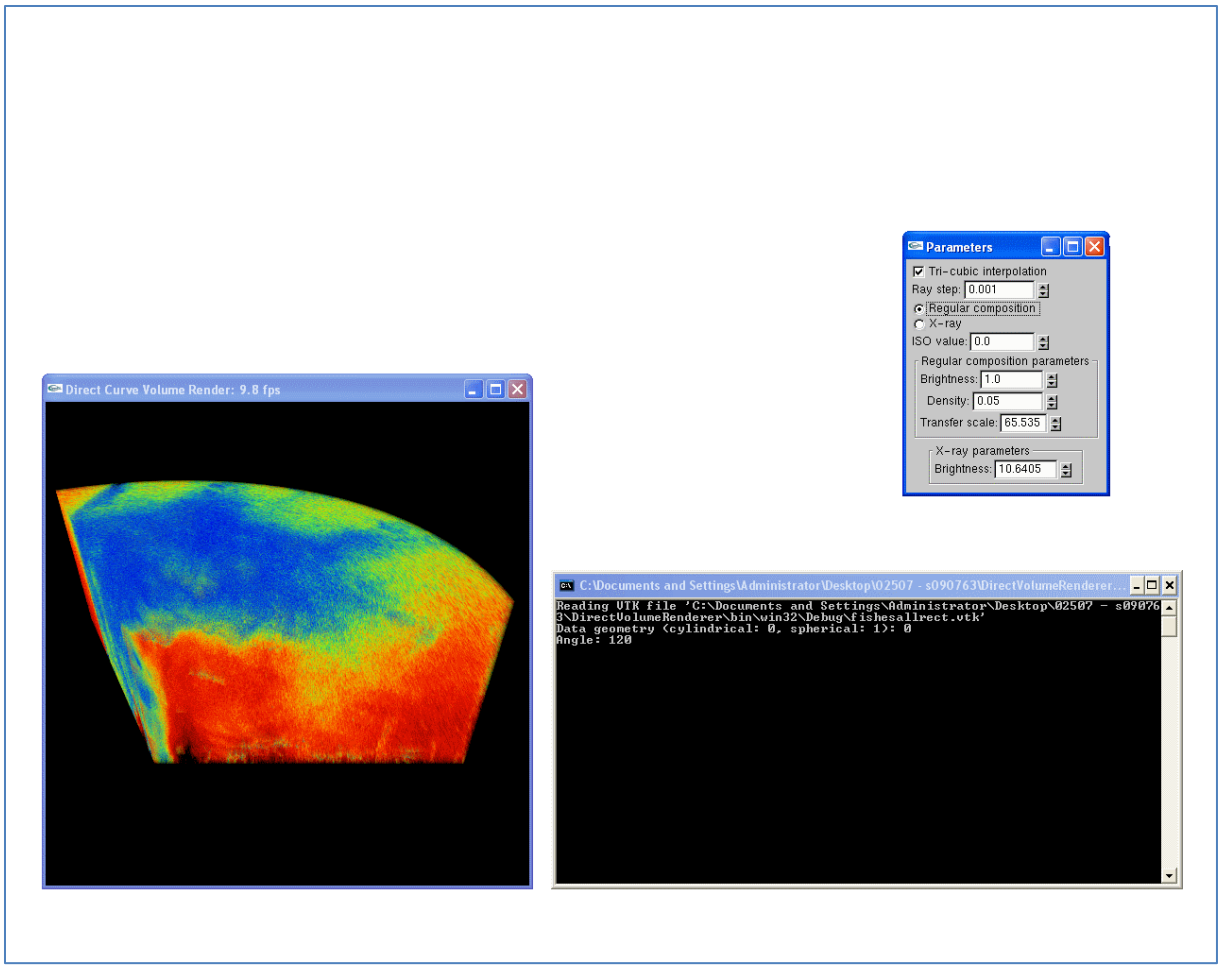


Figure 4 - The software at work.

The parameters that can be changed during visualization using the third window are the following:

- Interpolation: ticking the "Tri-cubic interpolation" checkbox switches from tri-linear interpolation to tri-cubic interpolation and vice versa (cf. section 2.2.5).
- Composition type: the two radio checkboxes allow to switch between regular composition and "x-ray" composition (cf. section 2.2.4).
- ISO value: the ISO value can be set using this box (cf. section 2.2.4).
- Regular composition parameters: the brightness, the density and the transfer scale of the regular composition can be set (cf. section 2.2.4).
- "X-ray" composition: the brightness of the "x-ray" composition can be set (cf. section 2.2.4).

The viewport can be moved with mouse gestures in the following way:

- Moving the mouse with the left button pressed results in rotation of the viewport around the volume.
- Moving the mouse with the right button pressed results in translating the viewport in its own plane.
- Moving the mouse with both the left and middle buttons pressed results in translating the viewport along the direction orthogonal to the viewport. If the volume is visible this is equivalent to zooming in or out. When the viewport gets closer to the volume than a specified distance, a plane is defined (parallel to the viewport and at this specified distance "in front of" the viewport), the *clipping plane*; the rendering is performed ignoring any part of the volume between the viewport and the clipping plane (cf. Figure 5).

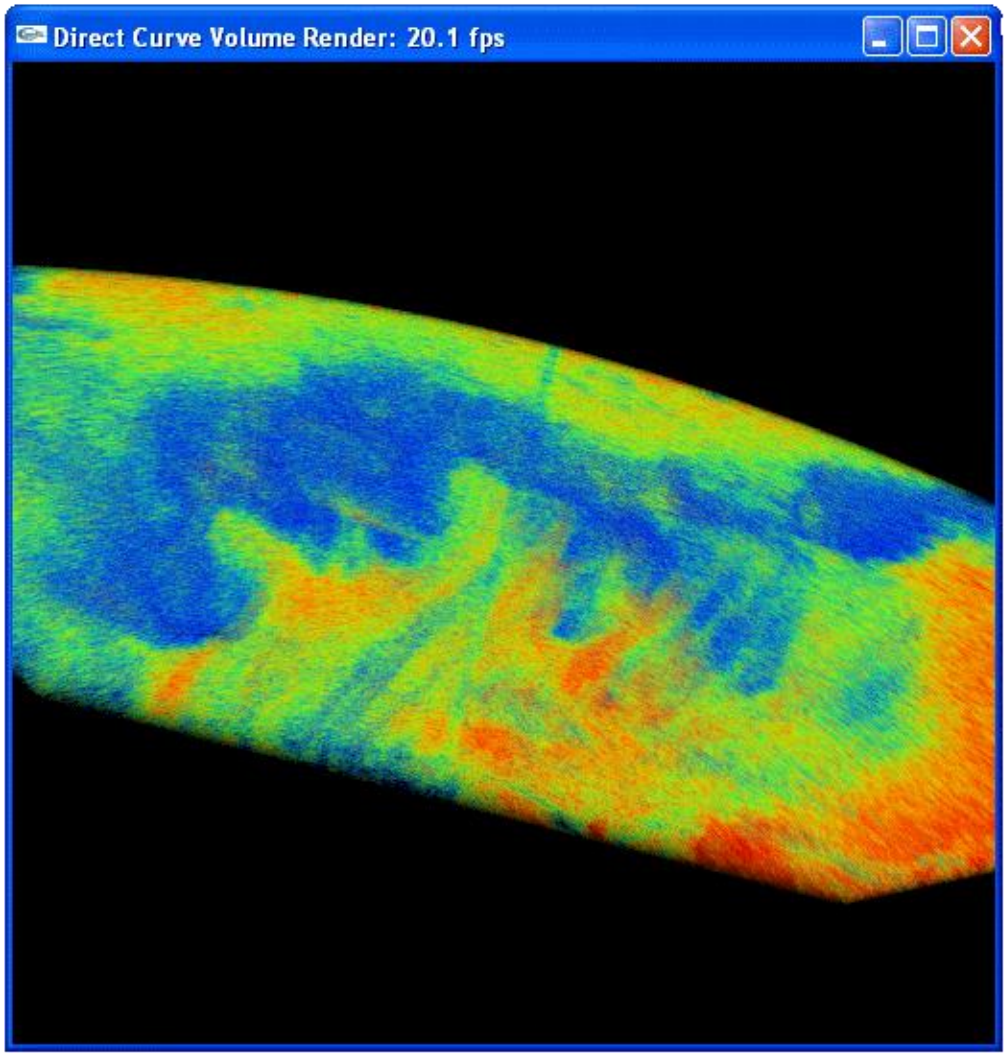


Figure 5 - Main window featuring plane clipping.

Additionally, the main window displays the current frame rate and when it is resized, the viewport is resized as well.

2.2. IMPLEMENTATION FRAMEWORK

2.2.1. COORDINATE SYSTEMS

Two coordinate systems are used simultaneously. The Cartesian (O, X, Y, Z) and either the cylindrical (cf. Figure 6) or the spherical (cf. Figure 7).

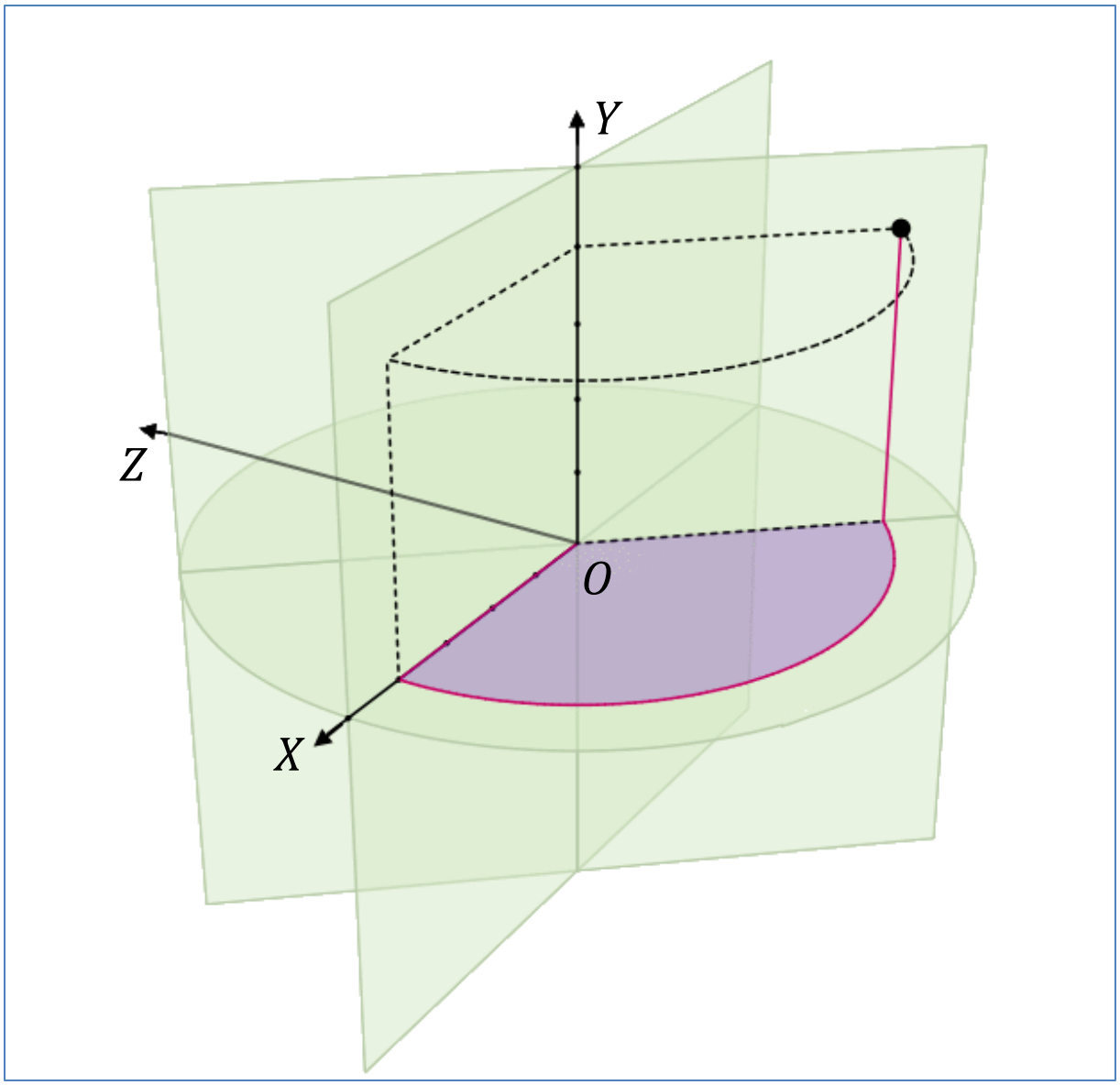


Figure 6 - Cylindrical and Cartesian coordinate systems.

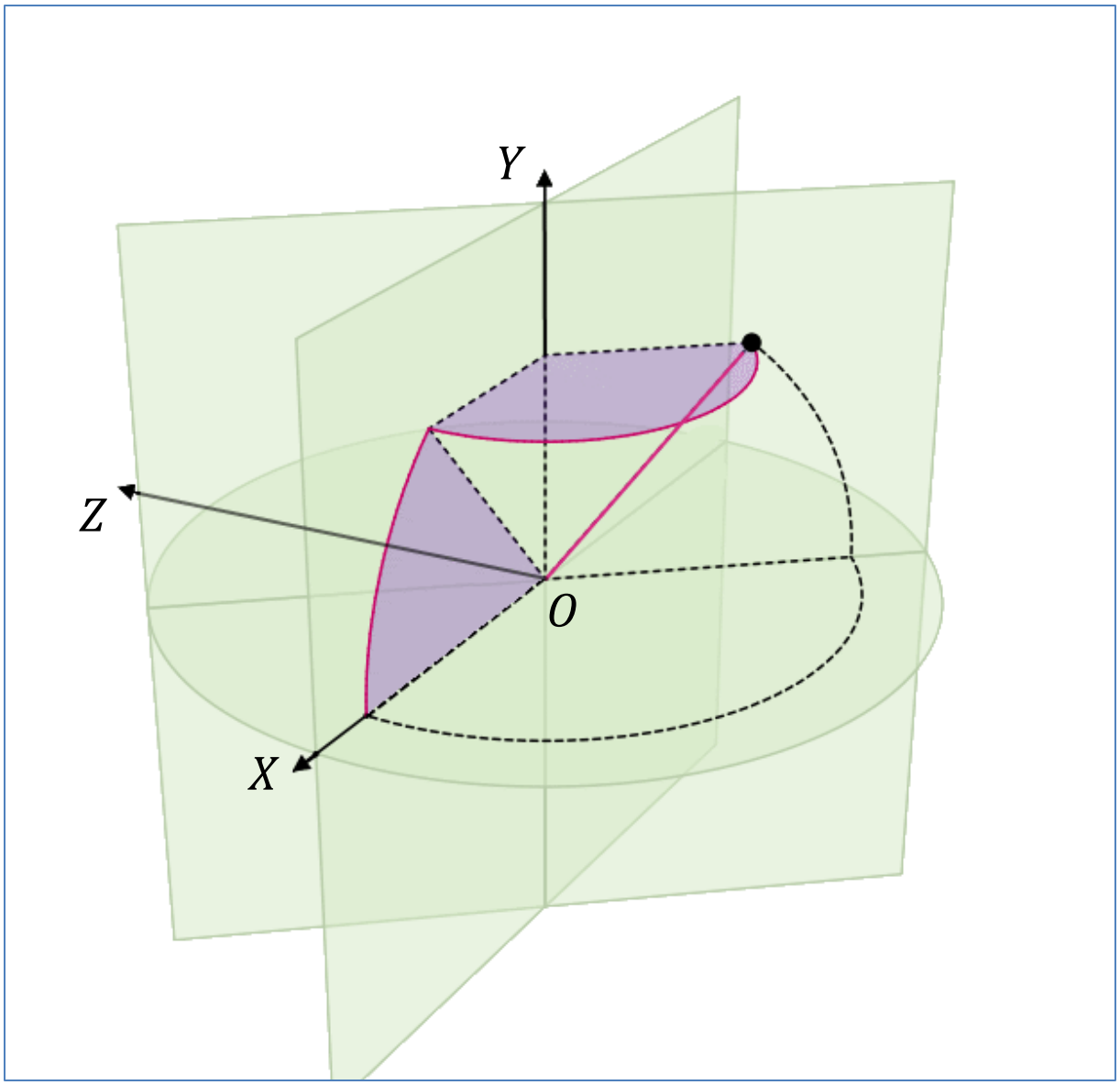


Figure 7 - Spherical and Cartesian coordinate systems.

2.2.2. BOUNDING GEOMETRY

The bounding geometry is the surface surrounding the volume.

For a cylindrical geometry, the volume is delimited by a maximal radial distance, a maximal axial position and a maximal angular position. All the points which radial distance, axial position and angular position are between 0 and the maximum corresponding value belong to the volume. The maximal radial distance and axial position relative values are deduced from the number of samples present in the beam space along the corresponding dimensions. The biggest of these values is set to 1 and the other is scaled to keep the same ratio. The maximum angular position is provided by the user.

For a spherical geometry, the volume is delimited by a maximal radial distance, a maximal latitude and a maximal longitude. All the points which radial distance, absolute value of latitude and longitude are between 0 and the maximum corresponding value belong to the volume. The maximal latitude and longitude relative values are deduced from the number of samples present in the beam space along the corresponding

dimensions. The maximum longitude is provided by the user and the maximum latitude is scaled to keep the same ratio. The maximum radial distance is set to 1.

2.2.3. VIEWPORT POSITION

The viewport and the point of view relative positions always remain the same. The point of view lies “behind” the viewport center, at a distance of 2. As the diameter of the volume is between 1 and 2 because of the scaling, this gives an optimal visibility of the volume.

2.2.4. COMPOSITION

The portion of the ray intersecting the object is sampled, starting from the point where the rays leaves the volume. At each sample, a value is interpolated from the known data (which have been scaled so their range is $[0,1]$.

- For the regular composition, each value is multiplied with the transfer scale.
  A corresponding RGBA vector is given by a piecewise linear function (the *transfer function)* mapping floating point numbers in $[0,1]$ to RGBA pixel values (as three-dimensional floating point number vectors with their coordinates varying between 0 and 1). Out of bounds inputs (out of $[0,1]$) give a vector with an alpha channel equal to 0. The function is defined by the vector between each linear portion. These vectors sample the color range between pure red and pure blue, with an alpha channel equal to 1.
  The alpha channel is then set to the density value if it was not 0.
  These RGBA vectors are accumulated by linearly interpolating a vector between the previously accumulated vector and the new one, where the new vector has for weight its alpha channel.
  The resulting RGB value multiplied by the brightness is the composited value.
  As a result, increasing the transfer scale emphasizes the differences but exclude the highest values and a low density results in a rather transparent volume.
- For the “x-ray” composition, the maximum value is kept only and multiplied by the brightness. The resulting value defines a grayscaled composited value (cf. Figure 8).

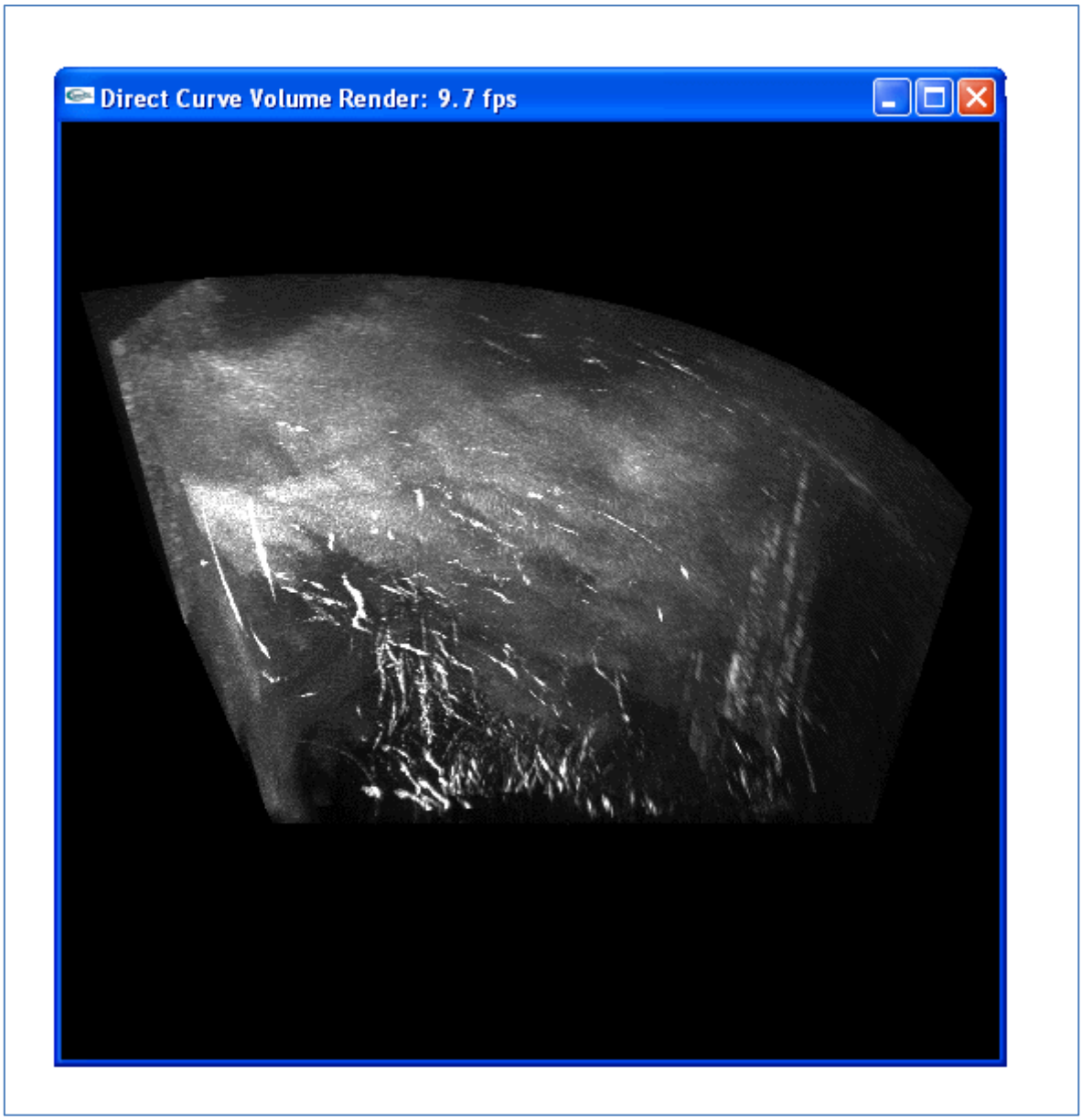


Figure 8 - Example of "x-ray" composition.

In both cases, setting an ISO value excludes the samples that are below this ISO value from the composition.

2.2.5. INTERPOLATION

As the values are regularly distributed in the beam space, the interpolation is performed in this space. To perform interpolation in the real space, one has to convert the Cartesian coordinates of the point considered to cylindrical or spherical depending on the geometry and to perform regular interpolation based on these coordinates in the dataset.

Cubic interpolation consists in the following formula (with $\hat{f}$ being the cubic interpolated function, f_i being the i-th sampled value and (w_0, w_1, w_2, w_3) being known weight functions):

$$\hat{f}(x) = w_0(x - \lfloor x \rfloor).f_{\lfloor x \rfloor - 1} + w_1(x - \lfloor x \rfloor).f_{\lfloor x \rfloor} + w_2(x - \lfloor x \rfloor).f_{\lfloor x \rfloor + 1} + w_3(x - \lfloor x \rfloor).f_{\lfloor x \rfloor + 2}$$

We have (with $\bar{f}$ being the linear interpolated function):

$$\hat{f}(x) = [w_0 + w_1](x - \lfloor x \rfloor).\bar{f}\left(\lfloor x \rfloor - \left[\frac{w_0}{w_0 + w_1}\right](x - \lfloor x \rfloor)\right) + [w_2 + w_3](x - \lfloor x \rfloor).\bar{f}\left(\lfloor x \rfloor + 1 + \left[\frac{w_3}{w_2 + w_3}\right](x - \lfloor x \rfloor)\right)$$

Therefore, the cubic interpolation consists in the calculations of two weights functions, two offset functions and two linear interpolations (4).

Finally, tri-cubic interpolation can be achieved with 7 successive cubic interpolations following the schema in Figure 9.

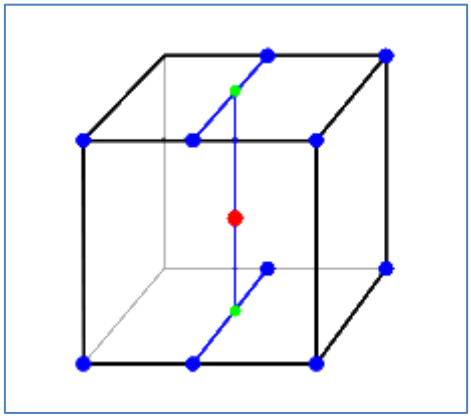


Figure 9 - Three-dimensional interpolation based on successive one-dimensional interpolations (5).

2.2.6. FRONT/BACK-FACE CULLING

In order to determine if a ray encounters the bounding geometry and at what distance from the point of view it goes in and out, two approaches are implemented: a general approach and two specialized approaches.

The general approach consists in decomposing the bounding geometry in triangles. Each ray tests for each triangle if it crosses it and at what distance and whether it is to go in or out of the volume.

The specialized approaches are the following:

- For the cylindrical geometry, testing first the front face (with the angular position equal to 0), then the back face (with the maximum angular position), then the top face (with the maximum axial position), then the bottom face (with the axial position equal to 0) then the side cylindrical face.
- For the spherical geometry, testing first the front face (with the longitude equal to 0), then the back face (with the maximum longitude), then the top conical face (with the maximum latitude), then the bottom conical face (with the minimum latitude) then the side spherical face.

Testing if a ray intersects a plane and at what distance requires solving a first degree equation. Testing if a ray intersects a cylinder, a cone or a sphere and at what distance requires solving a second degree equation.

Testing if the intersection point is inside a triangle is achieved by checking the sign of the barycentric coordinates of the point in the triangle.

Testing if the intersection point is actually in the face is achieved by comparing its coordinates in the appropriate coordinate system.

Testing if the ray goes in or comes out is achieved by a dot product with a normal vector coming out of the considered face at the intersection point.

2.3. CODE STRUCTURE

2.3.1. LANGUAGE

The software is implemented in C with some C++ features (these features do not include object oriented programming) and with the Nividia Cuda extensions. The graphical user interface uses the OpenGL Utility Toolkit (GLUT) as well as the OpenGL User Interface Library (GLUI).

All the code concerning regular CPU execution (mostly the user interface) is written in the file “main.cpp” while all the code containing Cuda extensions (GPU algorithms and memory transfer functions) are in the file “kernel.cu”.

2.3.2. GLOBAL FUNCTIONING

First the dataset is read and normalized by the program (so their range is $[0, M]$ where M is the maximum possible unsigned 2-byte integer) and the necessary information about the volume geometry is given by the user. The average value of the data is also computed.

The parameters are then initialized: the point of view is set to $(0,0,-4)$, giving a global and rather distant overview of the volume, as its diameter is always between 1 and 2. Most composition parameters are set to their natural default values, except for the “x-ray” brightness and the regular composition transfer scale. Their value is set so the global intensity displayed is that of a similar set which average value would be perfectly centered in the range of values.

The regular composition transfer function values are then set.

The both the Cuda API and the OpenGL API are initialized.

The data, the parameters and the transfer function values are loaded to the GPU memory.

Then, the user interface is created as well as a pixel buffer. The user interface main loop is then started.

At each refreshing time, threads of the main GPU-based algorithm (the *kernel)* are launched simultaneously with the current common parameters and thread repartition. Each thread computes a pixel value and writes it to the pixel buffer. The threads are grouped in blocks containing each 16x16 threads (except the last one if the current resolution is not divisible this way). The user interface displays the pixel buffer. It also computes the frame rate and displays it. If the window has been resized, the pixel buffer is reinitialized and the thread repartition is recomputed.

With such a functioning, the memory transfers with the GPU consist of the dataset and the bounding geometry description and the transfer function values when the program starts and are minimal after that: the data is read by the GPU-based algorithms from the GPU memory and written to the pixel-buffer. Therefore, the only transfers after initialization are the instant user specified parameters, including the viewport position.

2.3.3. GPU MEMORY

The Cuda framework provides different kinds of GPU memory structures (4). Because it is shared between all the threads and it remains as long as the program runs and it can take advantage of hardware tri-linear

interpolation, the dataset corresponding to the beam space is stored in the *texture memory* as a three-dimensional texture of 2-byte unsigned integers. It is set up so to return values scaled between 0 and 1 and to perform tri-linear interpolation (in the beam space) when necessary.

If the bounding geometry is defined using triangles, the triangles are stored as a one-dimensional texture of four-dimensional floating point number vectors (the last value of each vector is dummy; three consecutive vectors define a triangle points). This texture returns the same kind of vectors and doesn't perform interpolation.

The values defining the transfer function are stored in a one-dimensional texture of four-dimensional floating point number vectors. It is set up so to return the same kind of vectors and to perform linear interpolation when necessary.

Constant parameters concerning the dataset and the bounding geometry are stored in the *constant memory*, which remains also as long as the program runs.

The user-defined parameters as well as the current resolution and thread repartition are dynamically passed to the kernel instances as arguments at each refreshing time. However, the inverse view matrix (defining the transformation between the original and the current viewport position), also recomputed at each refreshing time has to be passed to the GPU by copying it to the constant memory because of limitations on the total size of the arguments when invoking the kernel.

To be accessible from the kernel, the pixel buffer is mapped to a pointer accessible from Cuda GPU-based functions. This pointer is passed as an argument to the kernel instances.

3. FUNCTION DESCRIPTIONS

3.1. DATA MANAGEMENT

- Function to compute Euclidian division of *a* by *b* and adds one to the result if *a* is not divisible by *b.*

```
inline int iDivUp(int a, int b)
```

- Function to convert degrees to radians.

```
inline double degToRad(double angle)
```

- Function to reinterpret the double precision floating point number *d* in the other endianness. It returns the new number.

```
double swap(double d)
```

- Function to read a compatible VTK file. It tries to parse the file which name is *filename*. If it succeeds, it stores the dataset dimensions in *width*, *height* and *depth.* It stores in *data* an array containing the values after having normalized them so their range is $[0, M]$ where M is the maximum possible unsigned 2-byte integer. It stores the average value of this array devided by M in *meanNorm*. It returns *true* if it succeeds and if the values of the dataset are not all the same.

```
bool readVtk(const char* filename, int* width, int* height, int*
depth, float* meanNorm, ushort** data)
```

- Function to load the bounding geometry dimensions (*size:* a three-dimensional floating point number vector corresponding to the range of each dimension defining the bounding geometry.) in the real space and the geometry type (*type:* 0 for cylindrical and 1 for spherical) to the GPU constant memory.

```
void loadGeometry(const float3* size, const char* type)
```

- Function to load the bounding geometry dimensions defined as triangles (*triangles* is an array containing four-dimensional floating point number vectors) to the GPU texture memory and the number (*n*) of triangles, the bounding geometry dimensions (*size*) and the geometry type (*type*) to the GPU constant memory.

```
loadTrianglesGeometry(const float4* triangles, const uint* n, const
float3* size, const char* type)
```

- Function to load the normalized data (*data*) to the GPU texture memory and the dimensions *dim* of the beam space in the GPU constant memory.

```
void loadData(const ushort* data, const cudaExtent* dim)
```

- Function to load the regular composition method piecewise linear function (*transferFunc* an array of four-dimensional floating point number vectors, each corresponding to a value joining to linear pieces) to the GPU texture memory.

```
void loadTransferFunc(const float4 transferFunc[])
```

- Function to initialize or re-initialize the pixel buffer.

```
void initPixelBuffer()
```

- Function to copy the inverse view matrix (*matrix*) to the GPU constant memory.

```
void copyInvViewMatrix(const float* matrix)
```

- Function to free the GPU texture memory.

```
void freeCudaBuffers()
```

- Main program exit handler. It frees the GPU texture memory and deletes the pixel buffer.

```
void cleanup()
```

3.2. GUI

- Function to initialize the GLUT display window.

```
void initGl(int argc, char** argv)
```

- GLUT display handler. It computes the current inverse view matrix, renders the new frame and displays it. Then it computes the frame rate and displays it as well.

```
void display()
```

- Function to load the inverse view matrix to the GPU memory and to call for the kernels.

```
void render()
```

- Function call the kernels with as arguments in that order, the repartition of the blocks of threads, the size of a block of thread, a pointer linked to the pixel buffer, the display width, the display height, the ray sampling step, wether or not to use tri-cubic interpolation, the density, the brightness for regular composition, the brightness for “x-ray” composition, the ISO value, the regular composition transfer scale and the composition type (0 for regular and 1 for “x-ray”).

```
void renderKernel(const dim3 &gridSize, const dim3 &blockSize, uint*
output, uint dispWidth, uint dispHeight, float rayStep, int triCubic,
float density, float regBrightness, float xrayBrightness, float
isoValue, float transferScale, int compoType)
```

- Function to compute and display the frame rate.

```
void computeFPS()
```

- GLUT window resize handler. It reinitialize the pixel buffer.

```
void reshape(int x, int y)
```

- GLUT mouse handler.

```
void mouse(int button, int state, int x, int y)
```

- GLUT/GLUI idle handler.

```
void idle()
```

3.3. MAIN ALGORITHMS

- GPU function to compute π.

```
__device__ float pi()
```

- GPU functions to perform multiplication of matrix *M* with vector *v.*

```
__device__
float4 mul(const float3x4 &M, const float4 &v)
__device__
Float3 mul(const float3x4 &M, const float3 &v)
```

- GPU function to compute the determinant of bi-dimensional vectors *u* and *v.*.

```
__device__
float det(float2 u, float2 v)
```

- GPU function to compute the Cartesian coordinates of the point which cylindrical coordinates are in *point.*

```
__device__
float3 cylToCart(float3 point)
```

- GPU function to compute the cylindrical coordinates of the point which Cartesian coordinates are in *point.*

```
__device__
float3 cartToCyl(float3 point)
```

- GPU function to compute the Cartesian coordinates of the point which spherical coordinates are in *point.*

```
__device__
float3 spherToCart(float3 point)
```

- GPU function to compute the spherical coordinates of the point which Cartesian coordinates are in *point.*

```
__device__
float3 cartToSpher(float3 point)
```

- GPU function to compute the barycentric coordinates of point *point* in triangle defined by *P0, P1, P2.*

```
__device__
float3 baryTriangleCoefs(const float3 &P0, const float3 &P1, const
float3 &P2, const float3 &point)
```

- GPU function to compute the barycentric coordinates of point *point* in triangle defined which first point has index *firstInd* in the texture containing the triangles data.

```
float3 baryTriangleCoefs(uint firstInd, float3 point)
```

- GPU function to check if a point is in a triangle based on its barycentric coordinates in the triangle.

```
__device__
bool inTriangle(float3 baryCoefs)
```

- GPU function to check if the ray coming from point *rayOrig* and going along the direction *rayVect* intersect the bounding geometry defined by the triangles. It stores the distance along the ray where it goes in the bounding geometry in *inAbs* and the distance where it goes out of the geometry in *outAbs*. It returns *true* if it could find the intersections and if they are coherent and false otherwise.

```
__device__
bool rayIntersectTriangles(const float3 &rayOrig, const float3
&rayVect, float* inAbs, float* outAbs)
```

- GPU function similar to the preceding but for cylindrical geometry without using the triangles.

```
__device__
bool rayIntersectCyl(const float3 &rayOrig, const float3 &rayVect,
float* inAbs, float* outAbs)
```

- GPU function similar to the preceding but for spherical geometry without using the triangles.

```
__device__
bool rayIntersectSpher(const float3 &rayOrig, const float3 &rayVect,
float* inAbs, float* outAbs)
```

- GPU function similar to the preceding but for both geometry with or without using the triangles.

```
__device__
bool rayIntersect(float3 rayOrig, float3 rayVect, float* inAbs,
float* outAbs)
```

- GPU function which returns as first components of its returned vector the two coordinates at which a linear interpolation with the weight returned as third component gives the cubic interpolation at the coordinate *x.*

```
__device__
float3 cubicInterpMixture(float x)
```

- GPU function which performs tri-cubic interpolation at the texture coordinates *texCoord.*

```
__device__
float cubicInterpolate(const float3 &texCoord)
```

- GPU function which performs interpolation for cylindrical geometry at point *point.* The value *outVal* is returned if the point is out of range. A tri-cubic interpolation is performed if *triCubic* is not set to 0.

```
__device__
float interpolateCyl(const float3 &point, float outVal, int triCubic)
```

- GPU function similar to the preceding but for spherical geometry.

```
__device__
float interpolateSpher(const float3 &point, float outVal, int
triCubic)
```

- GPU function similar to the preceding for both geometry.

```
__device__
```

```
float interpolate(float3 point, float outVal, int triCubic)
```

- GPU function to convert an RGBA color vector in and unsigned integer, suitable for being displayed in the frame buffer.

```
__device__
uint rgbaFloatToInt(float4 rgba)
```

- GPU kernel that performs the ray tracing with as arguments in that order, a pointer linked to the pixel buffer, the display width, the display height, the ray sampling step, wether or not to use tri-cubic interpolation, the density, the brightness for regular composition, the brightness for "x-ray" composition, the ISO value, the regular composition transfer scale and the composition type (0 for regular and 1 for "x-ray").

```
__global__
void kernel(uint* output, uint dispWidth, uint dispHeight, float
step, int triCubic, float density, float regBrightness, float
xrayBrightness, float isoValue, float transferScale, int compoType)
```

4. IMPLEMENTATION CHOICES

4.1. FRONT/BACK-FACE CULLING WITH OPENGL

A possibility for computing the intersections between the rays and the bounding primitive is to render the primitive using OpenGL, to enable back-face culling to have the entry point and front-face culling to have the exit point and to get the depth-buffer. However, this implies two additional rendering. Moreover, accessing the depth buffer from Cuda is not straightforward.For those reasons and as Cuda provides very good performances, it was decided to perform these calculations analytically.

4.2. INTERPOLATION

The use of hardware interpolation is a real gain in performances that is largely compensating the time spent computing the coordinates in the beam space.

For cubic interpolation, the current algorithm can be changed as is can be formulate so the only values needed (other than the linearly interpolated values) can be reduced to values of functions on $[0,1]$. Thus a possibility is to pre compute these functions values and put them in textures with interpolation. The resulting would be cubic interpolation with only four access to hardware linear interpolation. However it appeared the memory access was slower than the computation of the function values at run time.

4.3. BOUNDING GEOMETRY

The use of triangles is slower for the basic cylindrical or spherical geometries that were considered here: although I avoids second degree equations, the number of triangles is most of the time much higher than the number of surfaces considered in the other description of the bounding geometry. Therefore, the kernel computes the ray intersection faster with specific descriptions of the bounding geometries. However, using triangles might be useful for defining a complex bounding geometry covering a subset of the global dataset.

5. CONCLUSION

The resulting software is rather user-friendly and very flexible: it allows the user to precisely set the parameters in order to target the data to be displayed and to make his own choice between performances and precision while the animation remains smooth in most reasonable configurations. As shown in Figure 8 and Figure 10, it is possible to detect particular elements of the dataset using the different composition types and to find the appropriate parameters quite intuitively.

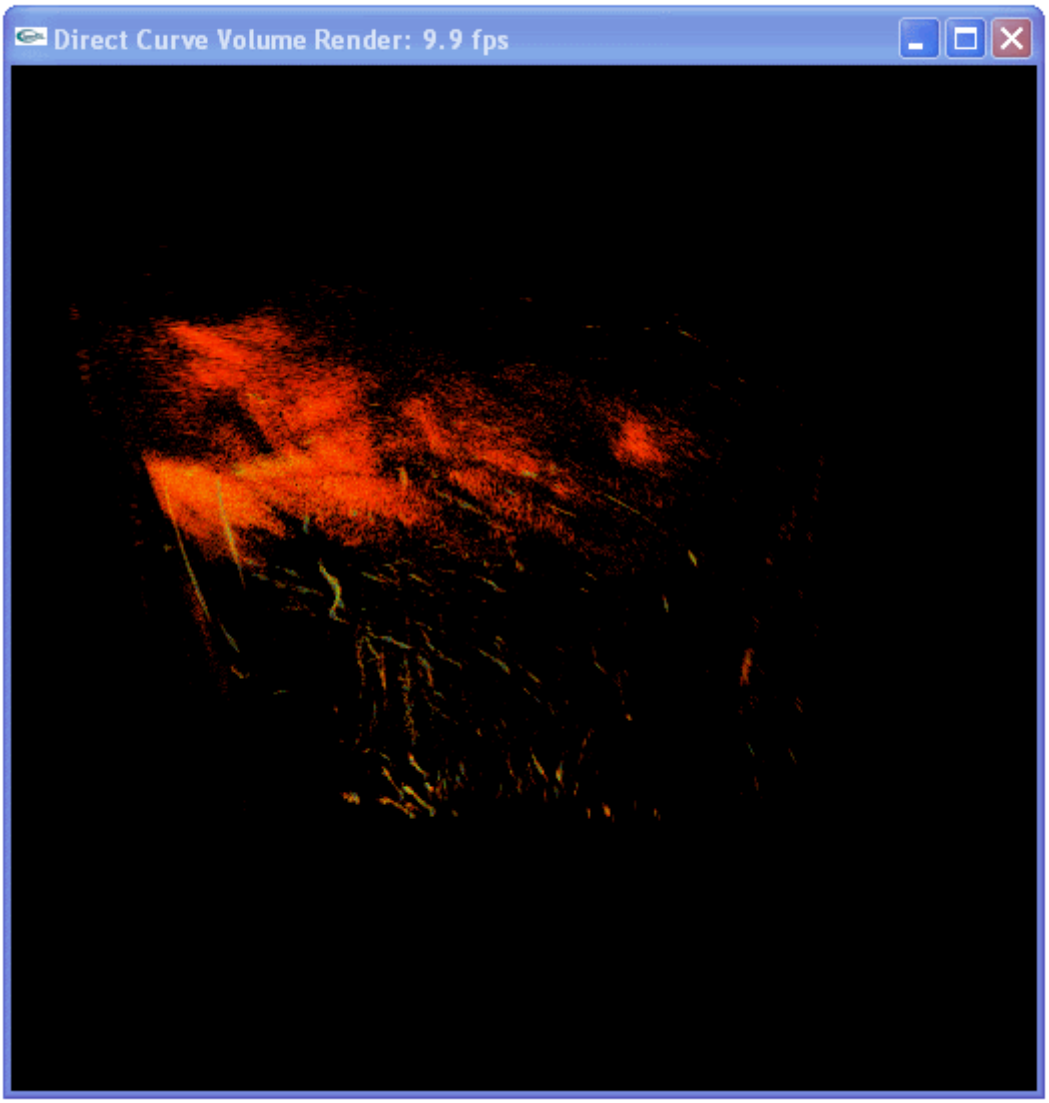


Figure 10 - Result of a configuration keeping only particular elements displayed.

BIBLIOGRAPHY

1. **Stolfi, Jorge.** Cylindrical coordinate system. *Wikipedia.* [Online] 05 03, 2009. http://en.wikipedia.org/wiki/Cylindrical_coordinate_system.

2. —. Spherical coordinate system. *Wikipedia.* [Online] 05 03, 2009.

3. **Thetawave.** Volume ray casting. *Wikipedia.* [Online] 01 25, 2006.

4. Fast Third-Order Texture Filtering. [book auth.] Christian Sigg and Markus Hadwiger. *GPU Gems 2.* 20.

5. **Marmelad.** Trilinear interpolation. *Wikipedia.* [Online] 07 10, 2008. http://en.wikipedia.org/wiki/Trilinear_interpolation.

6. **NVIDIA Corporation.** CUDA Programming Guide. Version 2.3.

APPENDIX

CODE: “MAIN.CPP”

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <cutil_inline.h>
#include <cutil_math.h>
#include <driver_functions.h>
#include <GL/glew.h>
#include <GL/glut.h>
#include <GL/glui.h>
#include <cuda_gl_interop.h>

typedef unsigned int uint;
typedef unsigned short ushort;
typedef unsigned char uchar;

extern "C" void loadData(const ushort* data, const cudaExtent* dim);
extern "C" void loadTrianglesGeometry(const float4* triangles, const uint* n, const float3*
    size, const char* type);
extern "C" void loadGeometry(const float3* size, const char* type);
extern "C" void loadTransferFunc(const float4 transferFunc[]);
extern "C" void freeCudaBuffers();
extern "C" void copyInvViewMatrix(const float* matrix);
extern "C" void renderKernel(const dim3 &gridSize, const dim3 &blockSize, uint* output, uint
    dispWidth, uint dispHeight, float rayStep, int triCubic, float density, float
    regBrightness, float xrayBrightness, float isoValue, float transferScale, int compoType);


uint dispWidth, dispHeight;
int triCubic;
float rayStep, density, regBrightness, xrayBrightness, isoValue, transferScale;
int compoType;
dim3 gridSize, blockSize;
float3 viewRotation, viewTranslation;
float invViewMatrix[12];
int ox, oy;
int buttonState = 0;
GLuint pbo = 0;
uint timer = 0;
int fpsCount = 0;
uint frameCount = 0;
int mainWindow;

// Debug function
void printVect(const float3 &vect)
{
    printf("\nx: %f, y: %f, z: %f", vect.x, vect.y, vect.z);
}

// Euclidian division + 1 if not divisible
inline int iDivUp(int a, int b)
{
    return (a % b != 0) ? (a / b + 1) : (a / b);
}

inline double degToRad(double angle)
{
    return 4.0 * atan(1.0) * angle / 180.0;
}

// Render image using CUDA
void render()
{
    // Copy inverse view matrix to device
    copyInvViewMatrix(invViewMatrix);

    // Map PBO to get CUDA device pointer
    uint* output;
    cutilSafeCall( cudaGLMapBufferObject((void**) &output, pbo) );
    cutilSafeCall( cudaMemset(output, 0, dispWidth * dispHeight * 4) );
```

```
    // Call CUDA kernel, writing results to PBO

    renderKernel(gridSize, blockSize, output, dispWidth, dispHeight, rayStep, triCubic,
    density, regBrightness, xrayBrightness, isoValue, transferScale, compoType);

    cutilCheckMsg( "Kernel failed" );

    cutilSafeCall( cudaGLUnmapBufferObject(pbo) );
}

// Compute and display framerate
void computeFPS()
{
    frameCount++;
    fpsCount++;
    if (fpsCount == 1) {
        char fps[256];
        float ifps = 1.f / (cutGetAverageTimerValue(timer) / 1000.f);
        sprintf(fps, "Direct Curve Volume Render: %3.1f fps", ifps);
        glutSetWindowTitle(fps);
        fpsCount = 0;
        cutilCheckError( cutResetTimer(timer) );
    }
}

// GLUT display handler
void display()
{
    cutilCheckError( cutStartTimer(timer) );

    // Use OpenGL to build view matrix
    GLfloat modelView[16];
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glRotatef(-viewRotation.x, 1.0, 0.0, 0.0);
    glRotatef(-viewRotation.y, 0.0, 1.0, 0.0);
    glTranslatef(-viewTranslation.x, -viewTranslation.y, -viewTranslation.z);
    glGetFloatv(GL_MODELVIEW_MATRIX, modelView);
    glPopMatrix();

    // Compute inverse view matrix
    invViewMatrix[0] = modelView[0]; invViewMatrix[1] = modelView[4]; invViewMatrix[2] =
    modelView[8]; invViewMatrix[3] = modelView[12];
    invViewMatrix[4] = modelView[1]; invViewMatrix[5] = modelView[5]; invViewMatrix[6] =
    modelView[9]; invViewMatrix[7] = modelView[13];
    invViewMatrix[8] = modelView[2]; invViewMatrix[9] = modelView[6]; invViewMatrix[10] =
    modelView[10]; invViewMatrix[11] = modelView[14];

    render();

    // Draw image from PBO
    glClear(GL_COLOR_BUFFER_BIT);
    glDisable(GL_DEPTH_TEST);
    glRasterPos2i(0, 0);
    glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
    glDrawPixels(dispWidth, dispHeight, GL_RGBA, GL_UNSIGNED_BYTE, 0);
    glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
    glutSwapBuffers();
    glutReportErrors();

    cutilCheckError( cutStopTimer(timer) );

    computeFPS();
}

// Program atrexit handler
void cleanup()
{
```

```
    freeCudaBuffers();
    cutilSafeCall( cudaGLUnregisterBufferObject(pbo) );
    glDeleteBuffersARB(1, &pbo);
}

// Initiate PBO
void initPixelBuffer()
{
    if (pbo) {
        // Delete old buffer
        cutilSafeCall( cudaGLUnregisterBufferObject(pbo) );
        glDeleteBuffersARB(1, &pbo);
    }

    // Create pixel buffer object for display
    glGenBuffersARB(1, &pbo);
    glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
    glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB, dispWidth * dispHeight * sizeof(GLubyte) * 4,↙
     0, GL_STREAM_DRAW_ARB);
    glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);

    cutilSafeCall( cudaGLRegisterBufferObject(pbo) );
}

// GLUT reshape handler
void reshape(int x, int y)
{
    dispWidth = x;
    dispHeight = y;

    gridSize = dim3(iDivUp(dispWidth, blockSize.x), iDivUp(dispHeight, blockSize.y));

    initPixelBuffer();

    glViewport(0, 0, x, y);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);
}

// GLUT/GLUI idle handler
void idle()
{
    if (glutGetWindow() != mainWindow)
    {
        glutSetWindow(mainWindow);
    }
    glutPostRedisplay();
}

// GLUT mouse handler
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN)
        buttonState  |= 1<<button;
    else if (state == GLUT_UP)
        buttonState = 0;

    ox = x;
    oy = y;
    glutPostRedisplay();
}

// GLUT motion handler
void motion(int x, int y)
{
```

```
    float dx, dy;
    dx = x - ox;
    dy = y - oy;

    if (buttonState == 3) {
        // left + middle = zoom
        viewTranslation.z += dy / 100.0;
    }
    else if (buttonState & 2) {
        // middle = translate
        viewTranslation.x += dx / 100.0;
        viewTranslation.y -= dy / 100.0;
    }
    else if (buttonState & 1) {
        // left = rotate
        viewRotation.x += dy / 5.0;
        viewRotation.y += dx / 5.0;
    }

    ox = x;
    oy = y;
    glutPostRedisplay();
}

// initialize GLUT
void initGl(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(dispWidth, dispHeight);
    mainWindow = glutCreateWindow("Direct Curve Volume Render");

    glewInit();
    if (!glewIsSupported("GL_VERSION_2_0 GL_ARB_pixel_buffer_object")) {
        fprintf(stderr, "Required OpenGL extensions missing.");
        exit(-1);
    }
}

// Swap between big-endian double and little-endian double
double swap(double d)
{
    union
    {
        double value;
        uchar bytes[];
    } in, out;

    in.value = d;

    for (uint i = 0; i < sizeof(double); i++)
    {
        out.bytes[i] = in.bytes[sizeof(double) - i - 1];
    }
  return out.value;
}

// Read data from an 'unsigned_short' binary VTK file
bool readVtk(const char* filename, int* width, int* height, int* depth, float* meanNorm,      ↙
    ushort** data)
{
    printf("Reading VTK file '%s'\n", filename);
    FILE *fp = fopen(filename, "rb");
    if (fp == NULL)
    {
        return false;
    }
    char line[1000];
    char dataType[1000];
    int w, h, d;
```

```
while(fgets(line, 1000, fp) != NULL)
{
    if (sscanf(line, "DIMENSIONS %d %d %d", &w, &h, &d))
    {
        *width = w;
        *height = h;
        *depth = d;
    }
    if (sscanf(line, "SCALARS %s %s", dataType, dataType) * w * h * d)
    {
        break;
    }
}
if (strlen(dataType) * w * h * d == 0)
{
    return false;
}


if (strcmp(dataType, "unsigned_short") == 0)
{
    fseek(fp, -w * h * d * sizeof(ushort) + 1, SEEK_END);

    ushort* rawData = (ushort*) malloc(sizeof(ushort) * w * h * d);
    size_t dataRead = fread(rawData, sizeof(ushort), w * h * d, fp);
    fclose(fp);
    if (dataRead == 0)
    {
        return false;
    }

    double minVal = (double) rawData[0];
    double maxVal = (double) rawData[0];
    for (uint i = 0; i < w * h * d; i++)
    {
        minVal = min(minVal, (double) rawData[i]);
        maxVal = max(maxVal, (double) rawData[i]);
    }

    float meanNormVal = 0.1f;
    *data = (ushort*) malloc(sizeof(ushort) * w * h * d);
    for (uint i = 0; i < w * h * d; i++)
    {
        float val = (float) ((((double) rawData[i]) - minVal) / (maxVal - minVal));
        meanNormVal += val;
        (*data)[i] = (ushort) (USHRT_MAX * val);
    }
    free(rawData);
    meanNormVal /= w * h * d;
    *meanNorm = meanNormVal;


    return meanNormVal > 0;
}
if (strcmp(dataType, "double") == 0)
{
    fseek(fp, -w * h * d * sizeof(double), SEEK_END);
    double* rawData = (double*) malloc(sizeof(double) * w * h * d);
    size_t dataRead = fread(rawData, sizeof(double), w * h * d, fp);
    fclose(fp);
    if (dataRead == 0)
    {
        return false;
    }

    double minVal = swap(rawData[0]);
    double maxVal = swap(rawData[0]);
    for (uint i = 0; i < w * h * d; i++)
    {
        minVal = min(minVal, swap(rawData[i]));
```

```
            maxVal = max(maxVal, swap(rawData[i]));
        }

        float meanNormVal = 0.1f;
        *data = (ushort*) malloc(sizeof(ushort) * w * h * d);
        for (uint i = 0; i < w * h * d; i++)
        {
            float val = (float) ((swap(rawData[i]) - minVal) / (maxVal - minVal));
            meanNormVal += val;
            (*data)[i] = (ushort) (USHRT_MAX * val);
        }
        free(rawData);
        meanNormVal /= w * h * d;
        *meanNorm = meanNormVal;

        return meanNormVal > 0;
    }
    return false;
}

int main(int argc, char* argv[])
{
    // Read VTK file
    int width, height, depth;
    float meanNorm;
    ushort* data;
    if (!readVtk(argv[1], &width, &height, &depth, &meanNorm, &data))
    {
        fprintf(stderr, "Could not read VTK file '%s'", argv[1]);
        exit(-1);
    }

    // Get geometry type
    char geomType;
    printf("Data geometry (cylindrical: 0, spherical: 1): ");
    scanf("%d", &geomType);

    // Get geometry size
    float3 geomSize;
    switch(geomType)
    {
    case 0:
        printf("Angle: ");
        scanf("%f", &geomSize.y);
        geomSize.x = width / fmaxf(width, depth);
        geomSize.y = degToRad(geomSize.y);
        geomSize.z = depth / fmaxf(width, depth);
        break;
    case 1:
        printf("Longitude: ");
        scanf("%f", &geomSize.y);
        geomSize.x = 1;
        geomSize.y = degToRad(geomSize.y);
        geomSize.z = geomSize.y * height / (2.0f * width);
        break;
    default :
        fprintf(stderr, "Wrong geometry");
        exit(-1);
    }

    // Parameters default values
    dispWidth = 512;
    dispHeight = 512;
    viewTranslation = make_float3(0.0, 0.0, -4.0f);
    blockSize = dim3(16, 16);
    triCubic = 1;
    rayStep = 0.001;
    compoType = 0;
    density = 0.05f;
    regBrightness = 1.0f;
```

```
isoValue = 0.0f;
transferScale = 1.0f / (2.0f * meanNorm);
xrayBrightness = 1.0f / (2.0f * meanNorm);
float4 transferFunc[9] = {{0.0, 0.0, 0.0, 0.0},
                          {1.0, 0.0, 0.0, 1.0},
                          {1.0, 0.5, 0.0, 1.0},
                          {1.0, 1.0, 0.0, 1.0},
                          {0.0, 1.0, 0.0, 1.0},
                          {0.0, 1.0, 1.0, 1.0},
                          {0.0, 0.0, 1.0, 1.0},
                          {0.0, 0.0, 1.0, 1.0},
                          {0.0, 0.0, 0.0, 0.0}};

// Initiate GLUT
initGl(argc, argv);

// Initiate device
cudaGLSetGLDevice(cutGetMaxGflopsDeviceId());

// Load geometry information
loadGeometry(&geomSize, &geomType);

// Load data
cudaExtent dataDim = make_cudaExtent(width, height, depth);
loadData(data, &dataDim);
free(data);

// Load transfer function
loadTransferFunc(transferFunc);

// Initiate timer
cutilCheckError( cutCreateTimer(&timer) );

// Set GLUT handlers
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMotionFunc(motion);

// Initiate PBO
initPixelBuffer();

// Display GUI
GLUI* glui = GLUI_Master.create_glui("Parameters");

glui->add_checkbox("Tri-cubic interpolation", &triCubic);

GLUI_Spinner* stepSpinner = glui->add_spinner("Ray step:", GLUI_SPINNER_FLOAT, &rayStep);
stepSpinner->set_float_limits(0.0001f, 0.1f);
stepSpinner->set_float_val(rayStep);

GLUI_RadioGroup* compositionRadio = glui->add_radiogroup(&compoType);
glui->add_radiobutton_to_group(compositionRadio,"Regular composition");
glui->add_radiobutton_to_group(compositionRadio,"X-ray");

GLUI_Spinner* isoValueSpinner = glui->add_spinner("ISO value:", GLUI_SPINNER_FLOAT, &     ↙
isoValue);
isoValueSpinner->set_float_limits(0.0f, 1.0f);
isoValueSpinner->set_float_val(isoValue);

GLUI_Panel* regularCompoPanel = glui->add_panel("Regular composition parameters");

GLUI_Spinner* regBrightnessSpinner = glui->add_spinner_to_panel(regularCompoPanel,       ↙
"Brightness:", GLUI_SPINNER_FLOAT, &regBrightness);
regBrightnessSpinner->set_float_limits(0.0f, 100.0f);
regBrightnessSpinner->set_float_val(regBrightness);

GLUI_Spinner* densitySpinner = glui->add_spinner_to_panel(regularCompoPanel, "Density:", ↙
GLUI_SPINNER_FLOAT, &density);
densitySpinner->set_float_limits(0.0f, 0.1f);
```

```
    densitySpinner->set_float_val(density);

    GLUI_Spinner* scaleSpinner = glui->add_spinner_to_panel(regularCompoPanel, "Transfer scale:", GLUI_SPINNER_FLOAT, &transferScale);
    scaleSpinner->set_float_limits(0.0f, 100.0f);
    scaleSpinner->set_float_val(transferScale);

    GLUI_Panel* xrayCompoPanel = glui->add_panel("X-ray parameters");

    GLUI_Spinner* xrayBrightnessSpinner = glui->add_spinner_to_panel(xrayCompoPanel, "Brightness:", GLUI_SPINNER_FLOAT, &xrayBrightness);
    xrayBrightnessSpinner->set_float_limits(0.0f, 100.0f);
    xrayBrightnessSpinner->set_float_val(xrayBrightness);

    glui->set_main_gfx_window(mainWindow);
    GLUI_Master.set_glutIdleFunc(idle);

    // Main loop
    glutMainLoop();

    // Exit handlers
    cudaThreadExit();
    atexit(cleanup);

    return 0;
}
```

CODE: “KERNEL.CU”

```
#ifndef _KERNEL_CU_
#define _KERNEL_CU_

#include <cutil_inline.h>
#include <cutil_math.h>

typedef unsigned int uint;
typedef unsigned short ushort;
typedef struct
{
    float4 m[3];
} float3x4;

__constant__ float3x4 invViewMatrix;
__constant__ cudaExtent dataDim;
__constant__ uint nTriangles;
__constant__ float3 geomSize;
__constant__ char geomType;
__constant__ bool geomIsTriangles;

cudaArray *dataArray = 0;
texture<ushort,  3, cudaReadModeNormalizedFloat> dataTex;

cudaArray *trianglesArray = 0;
texture<float4, 1, cudaReadModeElementType> trianglesTex;

cudaArray *transferFuncArray;
texture<float4, 1, cudaReadModeElementType> transferTex;


__device__
float pi()
{
    return 4.0f * atanf(1.0f);
}

__device__
float4 mul(const float3x4 &M, const float4 &v)
{
    float4 r;
    r.x = dot(v, M.m[0]);
    r.y = dot(v, M.m[1]);
    r.z = dot(v, M.m[2]);
    r.w = 1.0f;
    return r;
}

__device__
float3 mul(const float3x4 &M, const float3 &v)
{
    float3 r;
    r.x = dot(v, make_float3(M.m[0]));
    r.y = dot(v, make_float3(M.m[1]));
    r.z = dot(v, make_float3(M.m[2]));
    return r;
}

__device__
float det(float2 u, float2 v)
{
    return (u.x * v.y) - (u.y * v.x);
}

__device__
float3 cylToCart(float3 point)
{
    return make_float3(point.x * cosf(point.y), point.z, -point.x * sinf(point.y));
}

__device__
```

```
float3 cartToCyl(const float3 &point)
{
    float angle = atan2f(-point.z, point.x);
    if (angle < 0)
    {
        angle += 8.0f * atanf(1.0f);
    }
    return make_float3(sqrtf((point.z * point.z) + (point.x * point.x)), angle, point.y);
}

__device__
float3 spherToCart(float3 point)
{
    return make_float3(point.x * cosf(point.z) * cosf(point.y), point.x * sinf(point.z), -
    point.x * cosf(point.z) * sinf(point.y));
}

__device__
float3 cartToSpher(const float3 &point)
{
    float r = sqrtf((point.z * point.z) + (point.x * point.x));
    float R = sqrtf((point.x * point.x) + (point.y * point.y) + (point.z * point.z));
    float longit = atan2f(-point.z, point.x);
    if (longit < 0)
    {
        longit += 2.0f * pi();
    }
    float lat = atan2f(point.y, r);
    return make_float3(R, longit, lat);
}

__device__
float3 baryTriangleCoefs(const float3 &P0, const float3 &P1, const float3 &P2, const float3 &
    point)
{
    // (u, v) orthonormal basis of the triangle plane
    float3 u = normalize(P1 - P0);
    float3 v = normalize(cross(cross(u, P2 - P0), u));

    float2 P0P = {dot(point - P0, u), dot(point - P0, v)};
    float2 P1P = {dot(point - P1, u), dot(point - P1, v)};
    float2 P2P = {dot(point - P2, u), dot(point - P2, v)};

    float A = det(P1P, P2P);
    float B = det(P2P, P0P);
    float C = det(P0P, P1P);

    float delta = A + B + C;
    return make_float3(A / delta, B / delta, C / delta);
}

__device__
float3 baryTriangleCoefs(uint firstInd, float3 point)
{
    return baryTriangleCoefs(make_float3(tex1D(trianglesTex, firstInd)), make_float3(tex1D
    (trianglesTex, firstInd + 1)), make_float3(tex1D(trianglesTex, firstInd + 2)), point);
}

__device__
bool inTriangle(float3 baryCoefs)
{
    return baryCoefs.x >= 0 && baryCoefs.y >= 0 && baryCoefs.z >= 0;
}

__device__
bool rayIntersectTriangles(const float3 &rayOrig, const float3 &rayVect, float* inAbs, float*
     outAbs)
{
    bool inAbsFound = false;
    bool outAbsFound = false;
```

```
    float inAbsVal, outAbsVal;
    for (uint i = 0; i < nTriangles; i++) {
        float3 P0 = make_float3(tex1D(trianglesTex, 3 * i));
        float3 P1 = make_float3(tex1D(trianglesTex, (3 * i) + 1));
        float3 P2 = make_float3(tex1D(trianglesTex, (3 * i) + 2));
        float3 n = cross(P1 - P0, P2 - P0);
        float d = dot(n, rayVect);
        if (d > 0)
        {
            float s = dot(n, P0 - rayOrig) / d;
            float3 baryCoefs = baryTriangleCoefs(3 * i, rayOrig + (s * rayVect));
            if (inTriangle(baryCoefs))
            {
                outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
                outAbsFound = true;
            }
        }
        if (d < 0)
        {
            float s = dot(n, P0 - rayOrig) / d;
            float3 baryCoefs = baryTriangleCoefs(3 * i, rayOrig + (s * rayVect));
            if (inTriangle(baryCoefs))
            {
                inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
                inAbsFound = true;
            }
        }
    }

    if (inAbsFound && !outAbsFound)
    {
        outAbsVal = inAbsVal + 2 * geomSize.x / length(rayVect);
    }
    if (!inAbsFound && outAbsFound)
    {
        inAbsVal = outAbsVal - 2 * geomSize.x / length(rayVect);
    }

    *inAbs = inAbsVal;
    *outAbs = outAbsVal;

    return outAbsVal >= inAbsVal && outAbsFound && inAbsFound;
}

__device__
bool rayIntersectCyl(const float3 &rayOrig, const float3 &rayVect, float* inAbs, float*
    outAbs)
{
    float inAbsVal = 0;
    float outAbsVal = 0;
    bool inAbsFound = false;
    bool outAbsFound = false;

    float3 lastPoint = cylToCart(geomSize);

    // Front rectangle
    float3 P0 = {0, 0, 0};
    float3 P1 = {geomSize.x, 0, 0};
    float3 P2 = {0, geomSize.z, 0};
    float3 n = cross(P1 - P0, P2 - P0);
    float d = dot(n, rayVect);
    if (d != 0)
    {
        float s = dot(n, P0 - rayOrig) / d;
        float3 P = rayOrig + s * rayVect;
        if (P.x >= 0 && P.x <= geomSize.x && P.y >= 0 && P.y <= geomSize.z)
        {
            if (d > 0)
            {
                outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
```

```
            outAbsFound = true;
        }
        else
        {
            inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
            inAbsFound = true;
        }
    }
}

// Back rectangle
P0 = make_float3(lastPoint.x, 0, lastPoint.z);
P1 = make_float3(0, 0, 0);
P2 = make_float3(0, geomSize.z, 0);
n = cross(P1 - P0, P2 - P0);
d = dot(n, rayVect);
if (d != 0)
{
    float s = dot(n, P0 - rayOrig) / d;
    float3 P = rayOrig + s * rayVect;
    float x = dot(P, P0 - P1) / length(P0 - P1);
    if (x >= 0 && x <= geomSize.x && P.y >= 0 && P.y <= geomSize.z)
    {
        if (d > 0)
        {
            outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
            outAbsFound = true;
        }
        else
        {
            inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
            inAbsFound = true;
        }
    }
}

// Top face
P0 = make_float3(0, geomSize.z, 0);
n = make_float3(0, 1, 0);
d = dot(n, rayVect);
if (d != 0)
{
    float s = dot(n, P0 - rayOrig) / d;
    float3 P = cartToCyl(rayOrig + s * rayVect);
    if (P.x <= geomSize.x && P.y <= geomSize.y)
    {
        if (d > 0)
        {
            outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
            outAbsFound = true;
        }
        else
        {
            inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
            inAbsFound = true;
        }
    }
}

// Bottom face
P0 = make_float3(0, 0, 0);
n = make_float3(0, -1, 0);
d = dot(n, rayVect);
if (d != 0)
{
    float s = dot(n, P0 - rayOrig) / d;
    float3 P = cartToCyl(rayOrig + s * rayVect);
    if (P.x <= geomSize.x && P.y <= geomSize.y)
    {
        if (d > 0)
```

```
            {
                outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
                outAbsFound = true;
            }
            else
            {
                inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
                inAbsFound = true;
            }
        }
    }

    // Lateral face
    float2 u = {rayVect.x, rayVect.z};
    float2 v = {rayOrig.x, rayOrig.z};
    float a = dot(u, u);
    float b = 2 * dot(u, v);
    float c = dot(v, v) - (geomSize.x * geomSize.x);
    float delta = b * b - 4 * a * c;
    if (delta >= 0)
    {
        float s[2] = {(-b - sqrtf(delta)) / (2 * a), (-b + sqrtf(delta)) / (2 * a)};
        float3 P[2] = {rayOrig + s[0] * rayVect, rayOrig + s[1] * rayVect};

        float3 PCyl[2] = {cartToCyl(P[0]), cartToCyl(P[1])};
        float d[2] = {P[0].x * rayVect.x + P[0].z * rayVect.z, P[1].x * rayVect.x + P[1].z * rayVect.z};
        bool PInFace[2] = {PCyl[0].y <= geomSize.y && PCyl[0].z >= 0 && PCyl[0].z <= geomSize.z, PCyl[1].y <= geomSize.y && PCyl[1].z >= 0 && PCyl[1].z <= geomSize.z};

        for (uint i = 0; i <= 1; i++)
        {
            if (PInFace[i])
            {
                if(d[i] > 0)
                {
                    outAbsVal = outAbsFound ? fmaxf(outAbsVal, s[i]) : s[i];
                    outAbsFound = true;
                }
                if(d[i] < 0)
                {
                    inAbsVal = inAbsFound ? fminf(inAbsVal, s[i]) : s[i];
                    inAbsFound = true;
                }
                if(d[i] == 0)
                {
                    outAbsVal = outAbsFound ? fmaxf(outAbsVal, s[i]) : s[i];
                    outAbsFound = true;
                    inAbsVal = inAbsFound ? fminf(inAbsVal, s[i]) : s[i];
                    inAbsFound = true;
                }
            }
        }
    }

    if (inAbsFound && !outAbsFound)
    {
        outAbsVal = inAbsVal + 2 * geomSize.x / length(rayVect);
    }
    if (!inAbsFound && outAbsFound)
    {
        inAbsVal = outAbsVal - 2 * geomSize.x / length(rayVect);
    }

    *inAbs = inAbsVal;
    *outAbs = outAbsVal;

    return outAbsVal >= inAbsVal && outAbsFound && inAbsFound;
}
```

```
__device__
bool rayIntersectSpher(const float3 &rayOrig, const float3 &rayVect, float* inAbs, float*
    outAbs)
{
    float inAbsVal = 0;
    float outAbsVal = 0;
    bool inAbsFound = false;
    bool outAbsFound = false;

    float3 lastPoint = spherToCart(geomSize);

    // Top and bottom faces
    float2 u = {rayVect.x, rayVect.z};
    float2 v = {rayOrig.x, rayOrig.z};
    float t2 = tanf((pi() / 2.0f) - geomSize.z);
    t2 = t2 * t2;
    float a = dot(u, u) - (rayVect.y * rayVect.y * t2);
    float b = 2 * (dot(u, v) - (rayVect.y * rayOrig.y * t2));
    float c = dot(v, v) - (rayOrig.y * rayOrig.y * t2);
    float delta = b * b - 4 * a * c;
    if (delta >= 0)
    {
        float s[2] = {(-b - sqrtf(delta)) / (2 * a), (-b + sqrtf(delta)) / (2 * a)};
        float3 P[2] = {rayOrig + s[0] * rayVect, rayOrig + s[1] * rayVect};

        float3 PSpher[2] = {cartToSpher(P[0]), cartToSpher(P[1])};
        float d[2] = {(PSpher[0].z / fabs(PSpher[0].z)) * dot(make_float3(-sinf(PSpher[0].z)
    * cosf(PSpher[0].y), cosf(PSpher[0].z), sinf(PSpher[0].z) * sinf(PSpher[0].y)), rayVect),
                      (PSpher[1].z / fabs(PSpher[1].z)) * dot(make_float3(-sinf(PSpher[1].z)
    * cosf(PSpher[1].y), cosf(PSpher[1].z), sinf(PSpher[1].z) * sinf(PSpher[1].y)), rayVect)}
    ;
        bool PInFace[2] = {PSpher[0].y <= geomSize.y && PSpher[0].x <= geomSize.x, PSpher[1].
    y <= geomSize.y && PSpher[1].x <= geomSize.x};

        for (uint i = 0; i <= 1; i++)
        {
            if (PInFace[i])
            {
                if(d[i] > 0)
                {
                    outAbsVal = outAbsFound ? fmaxf(outAbsVal, s[i]) : s[i];
                    outAbsFound = true;
                }
                if(d[i] < 0)
                {
                    inAbsVal = inAbsFound ? fminf(inAbsVal, s[i]) : s[i];
                    inAbsFound = true;
                }
                if(d[i] == 0)
                {
                    outAbsVal = outAbsFound ? fmaxf(outAbsVal, s[i]) : s[i];
                    outAbsFound = true;
                    inAbsVal = inAbsFound ? fminf(inAbsVal, s[i]) : s[i];
                    inAbsFound = true;
                }
            }
        }
    }

    // Front face
    float3 P0 = make_float3(0, 0, 0);
    float3 n = make_float3(0, 0, 1);
    float d = dot(n, rayVect);
    if (d != 0)
    {
        float s = dot(n, P0 - rayOrig) / d;
        float3 P = rayOrig + s * rayVect;
        float3 PSpher = cartToSpher(P);
```

```
    if (PSpher.x <= geomSize.x && fabs(PSpher.z) <= geomSize.z && P.x >= 0)
    {
        if (d > 0)
        {
            outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
            outAbsFound = true;
        }
        else
        {
            inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
            inAbsFound = true;
        }
    }
}

// Back face
float3 P1 = lastPoint;
float3 P2 = make_float3(lastPoint.x, -lastPoint.y, lastPoint.z);
n = cross(P1 - P0, P2 - P0);
d = dot(n, rayVect);
if (d != 0)
{
    float s = dot(n, P0 - rayOrig) / d;
    float3 P = rayOrig + s * rayVect;
    float3 PSpher = cartToSpher(P);
    if (PSpher.x <= geomSize.x && fabs(PSpher.z) <= geomSize.z && dot(P, P2 - P0) >= 0)
    {
        if (d > 0)
        {
            outAbsVal = outAbsFound ? fmaxf(outAbsVal, s) : s;
            outAbsFound = true;
        }
        else
        {
            inAbsVal = inAbsFound ? fminf(inAbsVal, s) : s;
            inAbsFound = true;
        }
    }
}

// Lateral face
a = dot(rayVect, rayVect);
b = 2 * dot(rayVect, rayOrig);
c = dot(rayOrig, rayOrig) - (geomSize.x * geomSize.x);
delta = b * b - 4 * a * c;
if (delta >= 0)
{
    float s[2] = {(-b - sqrtf(delta)) / (2 * a), (-b + sqrtf(delta)) / (2 * a)};
    float3 P[2] = {rayOrig + s[0] * rayVect, rayOrig + s[1] * rayVect};

    float3 PSpher[2] = {cartToSpher(P[0]), cartToSpher(P[1])};
    float d[2] = {dot(P[0], rayVect), dot(P[1], rayVect)};
    bool PInFace[2] = {PSpher[0].y <= geomSize.y && fabs(PSpher[0].z) <= geomSize.z,
PSpher[1].y <= geomSize.y && fabs(PSpher[1].z) <= geomSize.z};

    for (uint i = 0; i <= 1; i++)
    {
        if (PInFace[i])
        {
            if(d[i] > 0)
            {
                outAbsVal = outAbsFound ? fmaxf(outAbsVal, s[i]) : s[i];
                outAbsFound = true;
            }
            if(d[i] < 0)
            {
                inAbsVal = inAbsFound ? fminf(inAbsVal, s[i]) : s[i];
                inAbsFound = true;
            }
            if(d[i] == 0)
```

```
                {
                    outAbsVal = outAbsFound ? fmaxf(outAbsVal, s[i]) : s[i];
                    outAbsFound = true;
                    inAbsVal = inAbsFound ? fminf(inAbsVal, s[i]) : s[i];
                    inAbsFound = true;
                }
            }
        }
    }
    if (inAbsFound && !outAbsFound)
    {
        outAbsVal = inAbsVal + 2 * geomSize.x / length(rayVect);
    }
    if (!inAbsFound && outAbsFound)
    {
        inAbsVal = outAbsVal - 2 * geomSize.x / length(rayVect);
    }

    *inAbs = inAbsVal;
    *outAbs = outAbsVal;

    return outAbsVal >= inAbsVal && (outAbsFound || inAbsFound);
}

__device__
float3 cubicInterpMixture(float x)
{
    float i = floor(x);
    float a = x - i;
    float a2 = a * a;
    float a3 = a * a2;
    float w0 = -a3 + (2 * a2) - a;
    float w1 = a3 - (2 * a2) + 1;
    float w2 = -a3 + a2 + a;
    float w3 = a3 - a2;
    return make_float3(i - (w0 / (w0 + w1)), i + 1 + (w3 / (w2 + w3)), w2 + w3);
}

__device__
float cubicInterpolate(const float3 &texCoord)
{
    float3 mixX = cubicInterpMixture(texCoord.x);
    float3 mixY = cubicInterpMixture(texCoord.y);
    float3 mixZ = cubicInterpMixture(texCoord.z);

    float vX0Y0Z0 = (float) tex3D(dataTex, mixX.x, mixY.x, mixZ.x);
    float vX0Y1Z0 = (float) tex3D(dataTex, mixX.x, mixY.y, mixZ.x);
    float vX0Y0Z1 = (float) tex3D(dataTex, mixX.x, mixY.x, mixZ.y);
    float vX0Y1Z1 = (float) tex3D(dataTex, mixX.x, mixY.y, mixZ.y);
    float vX1Y0Z0 = (float) tex3D(dataTex, mixX.y, mixY.x, mixZ.x);
    float vX1Y1Z0 = (float) tex3D(dataTex, mixX.y, mixY.y, mixZ.x);
    float vX1Y0Z1 = (float) tex3D(dataTex, mixX.y, mixY.x, mixZ.y);
    float vX1Y1Z1 = (float) tex3D(dataTex, mixX.y, mixY.y, mixZ.y);

    float vY0Z0 = lerp(vX0Y0Z0, vX1Y0Z0, mixX.z);
    float vY1Z0 = lerp(vX0Y1Z0, vX1Y1Z0, mixX.z);
    float vZ0 = lerp(vY0Z0, vY1Z0, mixY.z);

    float vY0Z1 = lerp(vX0Y0Z1, vX1Y0Z1, mixX.z);
    float vY1Z1 = lerp(vX0Y1Z1, vX1Y1Z1, mixX.z);
    float vZ1 = lerp(vY0Z1, vY1Z1, mixY.z);

    return lerp(vZ0, vZ1, mixZ.z);
}


__device__
float interpolateCyl(const float3 &point, float outVal, int triCubic)
{
    float lengthStep = geomSize.x / dataDim.width;
```

```
    float angleStep = geomSize.y / dataDim.height;
    float heightStep = geomSize.z / dataDim.depth;

    float3 cylCoord = cartToCyl(point);
    float3 texCoord = cylCoord / make_float3(lengthStep, angleStep, heightStep);

    if (texCoord.x < 0 || texCoord.x > dataDim.width - 1 || texCoord.y < 0 ||  texCoord.y >
    dataDim.height - 1 || texCoord.z < 0 ||  texCoord.z > dataDim.depth - 1)
    {
        return outVal;
    }
    if (triCubic)
    {
        return cubicInterpolate(texCoord);
    }
    return (float) tex3D(dataTex, texCoord.x, texCoord.y, texCoord.z);
}

__device__
float interpolateSpher(const float3 &point, float outVal, int triCubic)
{
    float lengthStep = geomSize.x / dataDim.depth;
    float longStep = geomSize.y / dataDim.width;
    float latStep = 2 * geomSize.z / dataDim.height;

    float3 spherCoord = cartToSpher(point) + make_float3(0, 0, geomSize.z);
    float3 texCoord = make_float3(spherCoord.y / longStep, spherCoord.z / latStep, spherCoord
    .x / lengthStep);

    if (texCoord.x < 0 || texCoord.x > dataDim.width - 1 || texCoord.y < 0 ||  texCoord.y >
    dataDim.height - 1 || texCoord.z < 0 ||  texCoord.z > dataDim.depth - 1)
    {
        return outVal;
    }
    if (triCubic)
    {
        return cubicInterpolate(texCoord);
    }
    return (float) tex3D(dataTex, texCoord.x, texCoord.y, texCoord.z);
}

__device__
float interpolate(float3 point, float outVal, int triCubic)
{
    switch(geomType)
    {
    case 0:
        return interpolateCyl(point, outVal, triCubic);
    case 1:
        return interpolateSpher(point, outVal, triCubic);
    }
}

__device__
bool rayIntersect(float3 rayOrig, float3 rayVect, float* inAbs, float* outAbs)
{
    if (geomIsTriangles)
    {
        return rayIntersectTriangles(rayOrig, rayVect, inAbs, outAbs);
    }
    switch(geomType)
    {
    case 0:
        return rayIntersectCyl(rayOrig, rayVect, inAbs, outAbs);
    case 1:
        return rayIntersectSpher(rayOrig, rayVect, inAbs, outAbs);
    }
}

__device__
```

```
uint rgbaFloatToInt(float4 rgba)
{
    rgba.x = __saturatef(rgba.x);
    rgba.y = __saturatef(rgba.y);
    rgba.z = __saturatef(rgba.z);
    rgba.w = __saturatef(rgba.w);
    return (uint(rgba.w*255)<<24) | (uint(rgba.z*255)<<16) | (uint(rgba.y*255)<<8) | uint
    (rgba.x*255);
}

__global__
void kernel(uint* output, uint dispWidth, uint dispHeight, float step, int triCubic, float
    density, float regBrightness, float xrayBrightness, float isoValue, float transferScale,
    int compoType)
{

    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

    float u = (2.0f * x / (float) dispWidth) - 1.0f;
    float v = (2.0f * y / (float) dispHeight) - 1.0f;

    float3 O = make_float3(mul(invViewMatrix, make_float4(0.0f, 0.0f, 0.0f, 1.0f)));
    float3 vect = normalize(make_float3(u, v, -2.0f));
    vect = mul(invViewMatrix, vect);

    float s = sqrtf(u * u + v * v + 4.0f) / 2.0f;

    float inAbs, outAbs;
    bool ok = rayIntersect(O, vect, &inAbs, &outAbs);


    if (ok && outAbs >= s)
    {
        inAbs = fmaxf(s, inAbs);

        switch(compoType)
        {
        case 0:
            float4 sum = make_float4(0.0f);
            for (float s = outAbs; s >= inAbs; s -= step)
            {
                float sample = interpolate(O + s * vect, 0, triCubic);

                if (sample >= isoValue)
                {
                    // Lookup in transfer function texture
                    float4 col = tex1D(transferTex, sample * transferScale);

                    // Accumulate result
                    sum = lerp(sum, col, col.w * density);
                }
            }

            if ((x < dispWidth) && (y < dispHeight)) {
                // Write output color
                uint i = __umul24(y, dispWidth) + x;
                output[i] = rgbaFloatToInt(sum * regBrightness);
            }
            break;

        case 1:
            float maxVal = 0.0f;
            for (float s = outAbs; s >= inAbs; s -= step)
            {
                float sample = interpolate(O + s * vect, 0, triCubic);
                if (sample >= isoValue)
                {
                    maxVal = fmaxf(maxVal, sample);
                }
```

```
            }
            maxVal *= xrayBrightness;

            if ((x < dispWidth) && (y < dispHeight)) {
                // Write output color
                uint i = __umul24(y, dispWidth) + x;
                output[i] = rgbaFloatToInt(make_float4(maxVal, maxVal, maxVal, 1.0f));
            }
        }
    }
}

extern "C"
void loadData(const ushort* data, const cudaExtent* dim)
{
    cutilSafeCall( cudaMemcpyToSymbol(dataDim, dim, sizeof(cudaExtent)) );

    // create 3D array
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<ushort>();
    cutilSafeCall( cudaMalloc3DArray(&dataArray, &channelDesc, (*dim)) );

    // copy data to 3D array
    cudaMemcpy3DParms copyParams = {0};
    copyParams.srcPtr = make_cudaPitchedPtr((void*) data, (*dim).width * sizeof(ushort), (*
    dim).width, (*dim).height);
    copyParams.dstArray = dataArray;
    copyParams.extent = (*dim);
    copyParams.kind = cudaMemcpyHostToDevice;
    cutilSafeCall( cudaMemcpy3D(&copyParams) );

    // set texture parameters
    dataTex.normalized = false;
    dataTex.filterMode = cudaFilterModeLinear;
    dataTex.addressMode[0] = cudaAddressModeClamp;
    dataTex.addressMode[1] = cudaAddressModeClamp;
    dataTex.addressMode[2] = cudaAddressModeClamp;

    // bind array to 3D texture
    cutilSafeCall( cudaBindTextureToArray(dataTex, dataArray, channelDesc) );
}

extern "C"
void loadTransferFunc(const float4 transferFunc[])
{
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float4>();
    cutilSafeCall( cudaMallocArray( &transferFuncArray, &channelDesc, 9, 1) );
    cutilSafeCall( cudaMemcpyToArray( transferFuncArray, 0, 0, transferFunc, 9 * sizeof
    (float4), cudaMemcpyHostToDevice) );

    transferTex.filterMode = cudaFilterModeLinear;
    transferTex.normalized = true;
    transferTex.addressMode[0] = cudaAddressModeClamp;

    cutilSafeCall( cudaBindTextureToArray( transferTex, transferFuncArray, channelDesc));
}

extern "C"
void loadTrianglesGeometry(const float4* triangles, const uint* n, const float3* size, const
    char* type)
{
    cutilSafeCall( cudaMemcpyToSymbol(nTriangles, n, sizeof(uint)) );
    cutilSafeCall( cudaMemcpyToSymbol(geomSize, size, sizeof(float3)) );
    cutilSafeCall( cudaMemcpyToSymbol(geomType, type, sizeof(char)) );
    bool isTriangles = true;
    cutilSafeCall( cudaMemcpyToSymbol(geomIsTriangles, &isTriangles, sizeof(bool)) );

    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float4>();
    cutilSafeCall( cudaMallocArray( &trianglesArray, &channelDesc, 3 * (*n), 1) );
    cutilSafeCall( cudaMemcpyToArray( trianglesArray, 0, 0, triangles, 3 * (*n) * sizeof
    (float4), cudaMemcpyHostToDevice) );
```

```
    trianglesTex.filterMode = cudaFilterModePoint;
    trianglesTex.normalized = false;
    trianglesTex.addressMode[0] = cudaAddressModeClamp;

    cutilSafeCall( cudaBindTextureToArray( trianglesTex, trianglesArray, channelDesc) );
}

extern "C" void loadGeometry(const float3* size, const char* type)
{
    cutilSafeCall( cudaMemcpyToSymbol(geomSize, size, sizeof(float3)) );
    cutilSafeCall( cudaMemcpyToSymbol(geomType, type, sizeof(char)) );
    bool isTriangles = false;
    cutilSafeCall( cudaMemcpyToSymbol(geomIsTriangles, &isTriangles, sizeof(bool)) );
}

extern "C"
void copyInvViewMatrix(const float* matrix)
{
    cutilSafeCall( cudaMemcpyToSymbol(invViewMatrix, matrix,  sizeof(float4) * 3) );
}

extern "C"
void renderKernel(const dim3 &gridSize, const dim3 &blockSize, uint* output, uint dispWidth, ↙
    uint dispHeight, float rayStep, int triCubic, float density, float regBrightness, float  ↙
    xrayBrightness, float isoValue, float transferScale, int compoType)
{
    kernel<<<gridSize, blockSize>>>(output, dispWidth, dispHeight, rayStep, triCubic, density↙
    , regBrightness, xrayBrightness, isoValue, transferScale, compoType);
}

extern "C"
void freeCudaBuffers()
{
    cutilSafeCall( cudaFreeArray(dataArray) );
    cutilSafeCall( cudaFreeArray(trianglesArray) );
    cutilSafeCall( cudaFreeArray(transferFuncArray) );
}

#endif
```