Abstract

The rapid development of image quality accompanied with rendering speed inquires have been a challenge to the programmers involved in large scale volume rendering especially for medical datasets. This project purpose is to perform volume rendering using the GPU which with its parallel processing has a massive contribution in this field. The final results would allow the user to interact and explore the data using three dimensional visualization techniques. The implementation would use different type of datasets type such as: medical CT scan, binary data and segmentation, the use of CUDA framework, for this project would significantly decrease the cost of volume analysis time.

Background

In visualization of medical and scientific data, volume rendering become a common used technique which generate 2D projection of 3D dataset. In rendering nature effects such as fluids, smoke, fire the Direct Volume Rendering (DVR) executed without extracting any surface geometry. The 3D dataset decomposed of discretely sample on regular grid in 3D space, on other words, the input of volume rendering is a 3D array of scalars. The array is called volume and each element of the array is called a voxel. Each of them has one or more constant data values. Most volume rendering systems set color and opacity based on a single scalar value and utilized 2D transfer functions of data value and gradient magnitude. Define a camera in space volume and the opacity and color (RGBA) for each voxel is needed to render a 2D projection of the 3D data set.

There are many applications of volume rendering algorithms which used volumetric display methods include software-based ray casting, shell rendering, splatting, shear-warp, hardware-based texture mapping, and GPU-accelerated rendering (Zhang, Q et al, 2009). To illustrate, a volume could be displayed by extracting iso-surfaces from the volume and rendering them as polygonal meshes or by implement a direct render as a block of data. Another common technique is using marching cubes algorithm to extract the iso-surface of volume data.
Method

The simplest approach for volume rendering is using 3D textures. The slices can be orthogonal oriented to the viewer's line of sight, with minimum number of sampling errors when creating slices for close-up views.

The following steps below shows how to do volume rendering using 3D textures.

1. The data information is loaded into a 3D texture. We do this one single time for a specific information volume.

2. Now we choose the number of cuts which are called slices. The spacing of these data slices should be chosen such that the sampling rate of the texture from one slice to another is same as the sampling rate of the texture within each slice. Uniform sampling rate treats 3D texture texels as cubical voxels, which minimizes resampling artifacts.

3. Find the wanted viewpoint and view direction.

For a cubical information volume, the quantity of cuts through the volume ought to generally coordinate the determination in texels of the cuts. At the point when the review course is not along a significant pivot, the quantity of test texels changes from plane to plane. Picking the quantity of texels along every side is normally a decent rough guess.
4. Calculate a group of polygons that cut through the data perpendicular to the direction of view. To texture the slice properly with respect to the 3D texture data we need to use texture coordinate generation. Here we use OpenGL (GL_QUADS) for this purpose.

![GL_QUADS](image)

*Figure 3: OpenGL (GL_QUADS) Treats each group of four vertices as an independent quadrilateral*

Figure 4: Calculating color and opacity of a pixel inside a texture-mapped polygon.

5. For setting the required orientation of the textured images on the slices we must use the texture transform matrix.

6. Render each slice as a textured polygon, from back to front or from front to back according to the blend operation which depends on the desired effect. There a number of common blending functions used in volume visualization, such as:

   a. **Over**: Volumes blended with the over operator approximate the flow of light through a colored, transparent material. The transparency of each point in the material is determined by the value of the texel's alpha channel. Texels with higher alpha values tend to obscure texels behind them, and stand out through the obscuring texels in front of them. This method that we used in our project with this rendering equation:

      \[
      C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1} \left(1 - A_j\right) \\
      A = 1 - \prod_{j=1}^{n} \left(1 - A_j\right)
      \]

      Where \(C_i\) and \(A_i\) are the color and opacity assigned by the transfer function to the data value at sample \(i\)

   b. **Attenuate**: The *attenuate* operator simulates an X-ray of the material. The final brightness at each pixel is attenuated by the total texel density along the direction of view.
c. **Maximum intensity projection (MIP)** finds the brightest texel alpha from all the texture slices at each pixel location.

d. **Under**: give the same result as the over operator blending slices but it start from back to front and it is slower because you need to accumulate all values.

7. Recalculate the data slice positions and update the texture transformation matrix would be necessary when the viewpoint and direction of view change.

### GPU Implementation

The rendering pipeline mode in GPU for a graphics process model follow a fixed structure. The process begins by taking volume data and evaluate the vertices. In the vertex processing converts each vertex into a 2D screen position, and opacity to determine its color. In the primitive assembly stage, vertices are collected and converted into polygons. (Quads in this project). The rasterization stage fills these quads with pixels known as “fragments” which wind up in the frame buffer. The value for each pixel that were rasterized are computed, based on the color and opacity this process called parameter interpolation. Finally accumulate the final fragment color, its opacity with the existing data stored at the associated 2D location in the frame buffer. The frame buffer controller interfaces to physical memory used to hold actual pixel values displayed on screen.

In this project we used six datasets with different sizes and dimensions such as, visualization of male head with 128x256x256 dimensions and 8,192 KB. Volume rendering describing the visualization of 3D data which visualizes the sampled functions of all three spatial dimensions by evaluating of the 2D projections.
The CUDA implantation technique was following this main steps:

For each pixel

1. For each f(i,j,k) along row line from pixel
2. Check f(i,j,k) in classification tables
3. If new substance
   a. Find surface normal/ compute color
   b. Weight color by opacity
   c. Accumulate color contribution
4. Pixel gets accumulated color

For each block executes the following in parallel on GPU

1. Render image using CUDA
   Map backbuffer to get CUDA device pointer
   cudaGraphicsMapResources(1, &cuda_pixBuffObj_resource, 0)
   cudaMemcpy(dataOutput, 0, width*height * 4));
   Call CUDA kernel, writing results to backbufer
   render_kernel(gridSize, blockSize, dataOutput, width, height, intensity, bright,
   transOffset, transScale);
   Display results using OpenGL (called by GLUT)
   Now encode for performing the operations

2. Load raw data from disk
   void *loadRawFile(char *filename, size_t size)
   First initialize OpenGl context, se we can properly set the GL for CUDA then use
   command-line specified CUDA device
   Then load volume data and synchronize
   cudaMemcpy(dataOutput, 0, width*height * 4));
   Calculate new grid size
   gridSize = dim3(iDivUp(width, blockSize.x), iDivUp(height, blockSize.y));
   Call CUDA kernel, writing result to Buffer
   copyInverseView(invertView, sizeof(float4) * 3);
   End CPU and GPU Timer

3. Then free CUDA Buffer Memory
   cudaFree(device_output);
   free(host_output);
   cleanup();

The program starting by calculating the CPU time using clock function. Then, use command-line
to specified CUDA device by initialize OpenGL at first and set it up for CUDA. After setting up
OpenGL to work with CUDA, we start calculating GPU Time by creating start and stop events
and recording the start event. The command line attempt to get the file name and file path and
pass an error message in case the file not found, otherwise, by getting the volume size we can
load the dataset.
size_t size = voxelSize.width*voxelSize.height*voxelSize.depth*sizeof(voxelType);
    void *host_volume = loadRawFile(path, size);
    initCuda(host_volume, voxelSize);

The grid size was calculated by using dim3 which is an integer vector type that can be used in CUDA code. Its most common application is to pass the grid and block dimensions in a kernel invocation. We divide the width and height of the dataset by the block size to get different values depending on the dataset size and bits.

gridSize = dim3(iDivUp(width, blockSize.x), iDivUp(height, blockSize.y));

The displaying process start by building view matrix for the final result and applies subsequent matrix operations to the model view matrix stack. We use OpenGL functions to push element in the matrix. Then, Copies count bytes from the memory area pointed to by Inverse view matrix to the memory area pointed by its size bytes from the start of symbol inverseD4 which is A 3-by-4 matrix for affine transformations of 3D geometry.

checkCudaErrors(cudaMemcpyToSymbol(inverseD4, InverseView, sizeIV));

Next, the render function map pixel buffer object for getting CUDA device pointer and map the count graphics resources for that object to be available for access by CUDA. Then we call the CUDA kernel to write the result in the pixel buffer object.

Device_render <<< gridSize, blockSize >>> (device_output, ImgWeight, ImgHeight, intensity, bright, transOffset, transScale);

for parsing along with the texels in the textures and to present 2D result we need 2D thread index

    un_int x = blockIdx.x*blockDim.x + threadIdx.x;
    un_int y = blockIdx.y*blockDim.y + threadIdx.y;

Making slices as a samples, calculate texel(texture pixel) in lines, each texel has its origin value and direction. We need to transform vector using matrix.

struct Line
{
    float3 origin;
    float3 direction;
};

Find intersection of line with the cube boundaries, then reorder intersection to find smallest and largest value on each axis by clamp to near plan.

    float largest_tmin = fmaxf(fmaxf(tmin.x, tmin.y), fmaxf(tmin.x, tmin.z));
    float smallest_tmax = fminf(fminf(tmax.x, tmax.y), fminf(tmax.x, tmax.z));

March along lines from front to back to accumulate color, over compositing algorithm implements the following equation while reading from 3D texture and lookup in transfer function texture
\[ \hat{C}_i = C_i + (1 - A_i) \hat{C}_{i+1} \]
\[ \hat{A}_i = A_i + (1 - A_i) \hat{A}_{i+1} \]

Where \( \hat{C}_i \) and \( \hat{A}_i \) are the accumulated color and opacity from the front of the volume.

Figure 5: accumulated color and opacity from the front of the volume

```cpp
for (int i = 0; i < maximumSteps; i++)
{
    // reading from 3D texture
    // performs a texture lookup in a given 3D sampler. May also use pre
    // computed derivatives if those are provided
    float texSample = tex3D(d_texture, position.x*0.5f + 0.5f, position.y*0.5f + 0.5f, position.z*0.5f + 0.5f);

    // performs a texture lookup in a given 1D sampler and,
    // in some cases, a shadow comparison. May also use pre computed derivatives
    if those are provided
    float4 column = tex1D(TraFuncTex, (texSample - transOffset)*transScale);
    column.w *= intensity;

    // pre Alpha Blending multiply
    column.x *= column.w;
    column.y *= column.w;
    column.z *= column.w;

    // "over" operator for front-to-back blending
    sum = sum + column*(1.0f - sum.w);

    // exit early if opaque
    if (sum.w > opacityThreshold)
        break;
    intersect += InterStep;
    if (intersect > interFar)
        break;
    position += theStep;
}

sum *= bright;
```
// write resulting color
device_output[y*ImgWeight + x] = RGBAFloatToInt(sum);

Then start drawing the resulting image by specifying the alignment requirements for the start of each pixel row in memory. Drawing by textures start by copy from the pixel buffer object to textures.

![Figure 6: Volume Sample and compositing](image)

Images are created by sampling the volume along all viewing lines and accumulating the resulting optical properties

**RESULTS**

We have tested our code on three datasets and we obtained the following results. The below figure shows the Volume Render of Human Head. The CPU time and GPU time can also be seen in the image.

![Figure 7: the result of CT scan to Visible Male Head (VisMale.raw)](image)

In the same way we do the volume rendering for different data files. These data files are differentiated based on the voxel size per bits. The CPU time, GPU time and the Speedup are
calculated in the same way as they are done for the human head. The below table shows in detail the information about the three data files that are volume rendered.

<table>
<thead>
<tr>
<th>Data sets</th>
<th>Voxel size /bits</th>
<th>File size (KB)</th>
<th>CPU Time (Ms)</th>
<th>GPU Time (Ms) GTX 850M</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>VisMale.raw</td>
<td>128x256x256 8 bit</td>
<td>8,192</td>
<td>97.00</td>
<td>6.28</td>
<td>15.44x</td>
</tr>
<tr>
<td>golfball.raw</td>
<td>512x256x256 8 bit</td>
<td>32,768</td>
<td>179.00</td>
<td>15.13</td>
<td>11.83x</td>
</tr>
<tr>
<td>walnut-core-mask.raw</td>
<td>400x296x352 16 bit</td>
<td>40,700</td>
<td>207.00</td>
<td>17.66</td>
<td>11.72x</td>
</tr>
</tbody>
</table>

From the table above we can observe that as the voxel size and bits increases then the size of file increases and thus increases the CPU time and GPU time. The following graph shows the Run Time for different sizes of Data Sets. We can see that the Run Time of GPU is fairly consistent for all sizes whereas the CPU Run Time varies.

The graph between the speedup and dataset file size gives us the performance of the code. Speedup decreases fairly with increasing size of Data Set.
The below figures shows the volume rendering of the remaining data sets. The second is a golf ball. The last one is binary as well as probabilistic segmentation of the walnut core data set segments.

![Figure 8: (Walnut-core-mask.raw) is Binary as well as probabilistic segmentations of the walnut core data set segments](image1.png)

![Figure 9: golfball.raw](image2.png)

**CONCLUSION**

This project was a great challenge because it required massive knowledge of GPU programming with CUDA and OpenGL along with principals of parallel computing. We were able to perform the most desirable algorithm for volume rendering because it represents fast result, but not the best result of course. After managing make our program as simple as possible according to our standing and looking for many examples and solution for this problem which we were exciting to achieve, we were able to test our program in many available datasets. CUDA programming and Visual Studio tool kit we are able to successfully implement the .Then we obtained the speed up for certain data files which are above 10 . Thus the project requirement is satisfied.
REFERENCES


Data Sets from:

