Computer Graphics project : Volumetric Rendering

Lim-Sylvie Pou, Sacha Vakili, Jamal Gourinda
& Sylvestre Rebuffi

March 6, 2016

Abstract

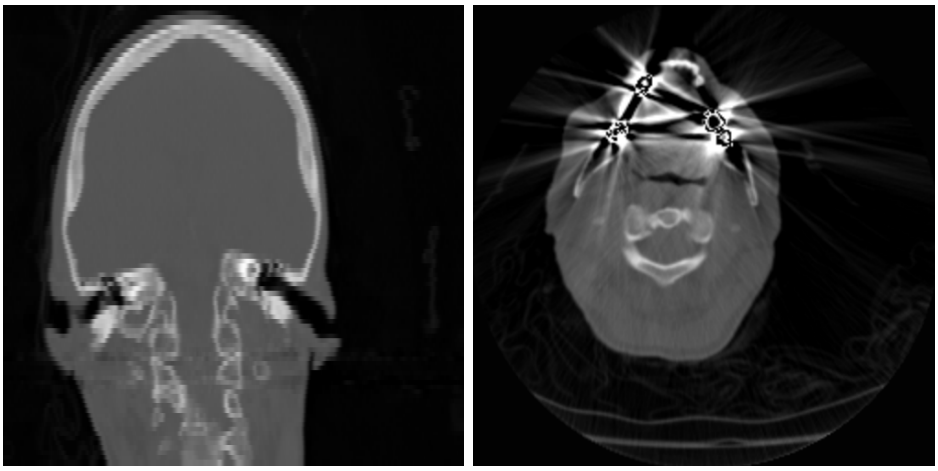
As a final project for the Computer Graphics course we chose the Volume Rendering subject. The appealing idea of volume rendering was the motive behind this choice. From a practical point of view each part of the project was assigned to member of the group as indicated in each title. The same member was responsible for the writing of the corresponding part of the final report. The code was annotated and further explained below.

1 Visualization by accumulation (Jamal Gourinda)

The goal of this section is to render volumetric visualization of the skull by accumulating the density of all voxels along one viewing direction. As suggested by the provided code this was achieved step by step. Our data is a set 2D slices arranged in a texture. The overall texture contains 100 slices with a resolution of 256×256.

1.1 Simple and accumulated views

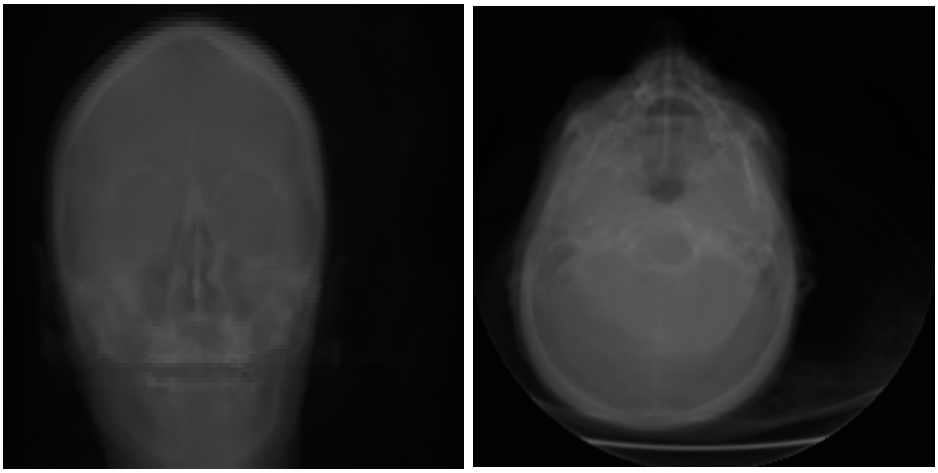
Extracting a simple vertical or horizontal views is done through fixing one of the x, y or z coordinate while letting the two others span the image. Below we illustrate some results.



(a) Vertical View : y is fixed (b) Horizontal View : z is fixed

Figure 1: Rendering of simple views

Then obtaining an accumulated image is done through looping over the "fixed" coordinate and adding the corresponding texture color. Note that naturally the range of the loop is different if the view is vertical or horizontal. The range is 100 for a horizontal view because we have that many slices. For a vertical view the range is 256 matching the y resolution of the slices. Below are some results.



(a) Accumulated Vertical View (b) Accumulated Horizontal View

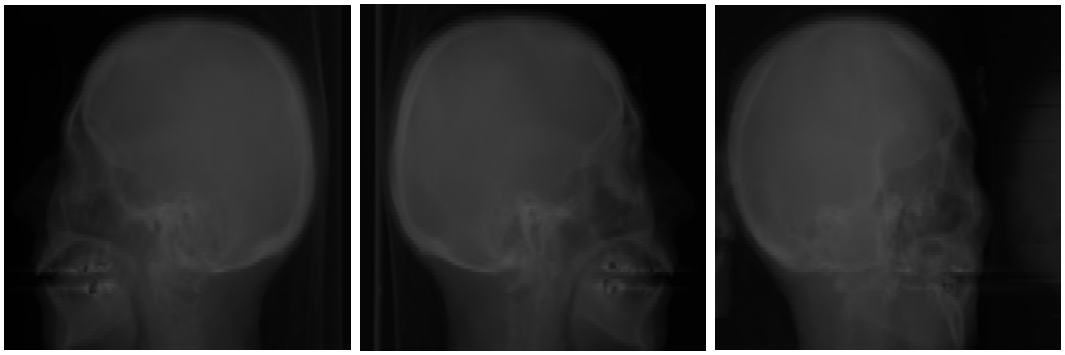
Figure 2: Rendering of accumulated views

1.2 Rotated views

In this part the goal is to render rotated views of the skull. The rotation is controlled by the *rotationAngle* parameter in the fragment shader. Because the rotation that we considered is around the z axis the implementation of this view relied on the accumulated vertical view that was displayed earlier. We then performed a classic affine rotation of the coordinates x and y to obtain the result. Below is the complete transformation that was applied :

$$\begin{aligned} x_{rot} &= (x - x_c) * cos(\theta) + (y - y_c) * sin(\theta) + x_c \\ y_{rot} &= -(x - \underbrace{x_c}_{=0.5}) * sin(\theta) + (y - \underbrace{y_c}_{=0.5}) * cos(\theta) + y_c \end{aligned}$$

The transformation is a composition of three elementary steps. First a change of coordinates centered on the center of the image (red), a classic rotation of angle θ (green) and finally a move back to the original coordinate system (blue). On top of that transform we clamp x_{rot} and y_{rot} to their natural range $[0, 1]$. Below we present some results.



(a) $\theta = -\frac{\pi}{2}$ (b) $\theta = \frac{\pi}{2}$ (c) $\theta = \frac{\pi}{2}$ without clamping

Figure 3: Rendering of rotated views : We notice that clamping the transform removes the artifacts that are observed on the edges

2 Extracting an iso-surface (Sylvestre Rebuffi)

To extract iso-surfaces, we perform a ray-marching algorithm. To do that, for each fixed (x, z), we increment the y parameter starting from 0 and we stop the incrementing loop when we are in a voxel with a density equal or above the iso-value specified by the parameter $isoValue$ in the fragment shader file. So this algorithm mimics the behaviour of a ray in the y direction stopped by a dense zone. The obtained (x, y, z) form the iso-surface for the specified iso-value and we display this iso-surface with a constant white color.

To control and specify the iso-value, we created a global variable $isoValue$ in the file "volume_render.cpp" and this variable can be controlled either by the keyboard with "u" and "d" or by the mouse motion. Then we transmit the parameter to the fragment shader file thanks to:

Listing 1: Code excerpt 1

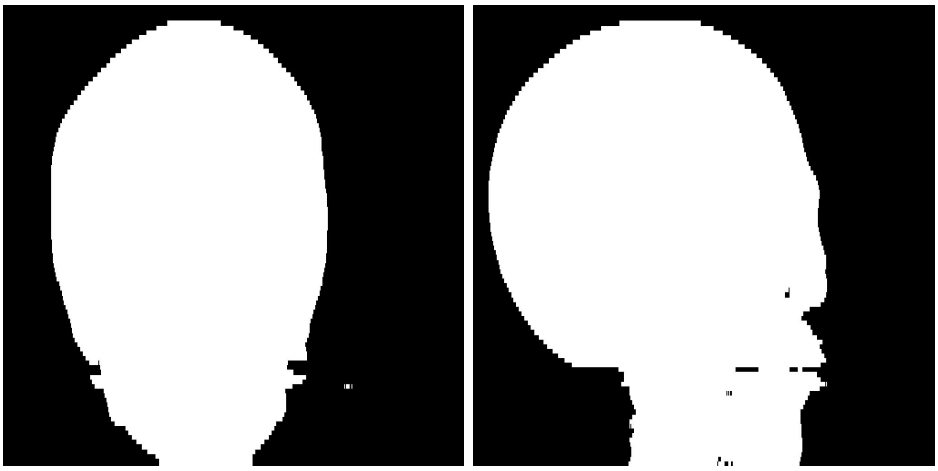
```
GLuint IsovID = glGetUniformLocation(g_glslProgram, "isoValue");
glUniform1f(IsovID, isoValue);
```

Now we have to handle the case of ray-marching when the skull is rotated. So we mix the increment loop with the rotation step described above: we keep incrementing in the y direction for each (x, z) as we keep looking in the same direction but besides that we have to perform a change of coordinates. As before, we want the central axis of the skull to stay rotation invariant so we translate the central axis, then rotate and translate back the central axis to its original position. Furthermore, as before, we have to clamp because of the "modulo" property as the rotated square give coordinates above 1. We get the final ray-marching loop:

Listing 2: Code excerpt 2

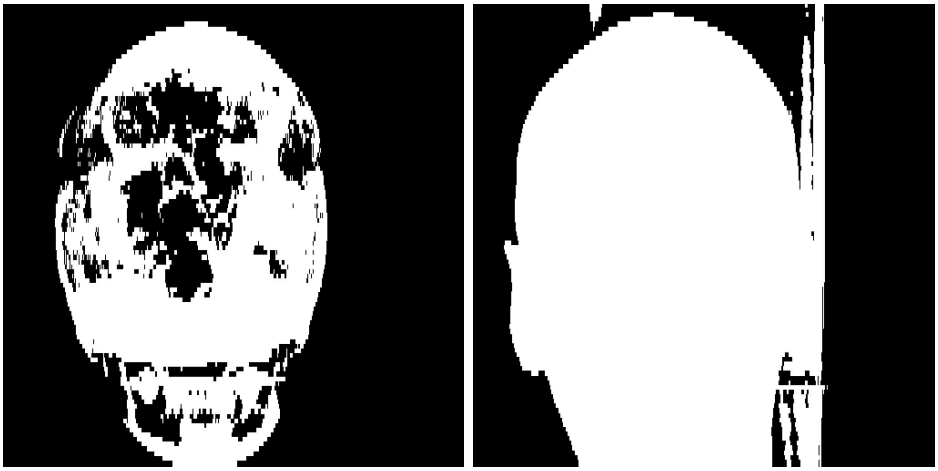
```
float x1,y1;
      x = fragmentUV.x;
      z = fragmentUV.y;
      color = vec3(0.0,0.0,0.0);
      for (int i=0; i<256; i++) {
         y = float(i)/256.;
         // Central axis has to be rotation invariant so we translate to (x,y)=(0,0),
         // then we rotate and translate the axis back to (x,y)=(0.5,0.5)
         x1= (x-0.5)*cos(rotationAngle)+sin(rotationAngle)*(y-0.5)+0.5;
         y1= -(x-0.5)*sin(rotationAngle)+cos(rotationAngle)*(y-0.5)+0.5;
         pixCoord = pixel_coordinate(x1,y1,z);
         if((texture(myTextureSamplerVolume, pixCoord).r > isoValue)&&    // isoValue
            (x1 >= 0.)&&(y1 >= 0.)&&(x1 <= 1.)&&(y1 <= 1.)) {             // Clamping
            color=vec3(1.0,1.0,1.0);
            break;
           }
    }
```

Now we show some iso-surfaces for different iso-values and rotations. For high iso-values, we have only bones and for low iso-values, we can see flesh like the ear in Figure 5:



(a) Without rotation (b) With rotation

Figure 4: Result of ray-marching for the iso-value of the skull



(a) High iso-value (b) Small iso-value

Figure 5: Result of ray-marching for different iso-values

3 Shading an iso-surface

The last step of the project is to perform the ray-marching algorithm described above in order to display the iso-surface with an RGB color first, then with the Phong model. To achieve this, we need to compute the normals.

3.1 Computing the surface normals (Sacha Vakili)

Instead of displaying the same color (white) for every element of the surface as in ray-marching, we are now interested in displaying values depending on the surface. In that aim, we want to compute the surface normals for every point (x, y, z) in the voxel.

As advised, we pre compute this values in a new texture named *dataNormals* and send it to the fragment shader. This requires thus a loop over all 256×256 pixels and all 100 slices:

Listing 3: Code excerpt 3

```
    //Compute normal from gradient for each voxel (x,y,z)
 for (int x = 0; x < 256; x++) {
  for (int y = 0; y < 256; y++) {
   for (int z = 0; z < 100; z++) {
```

For every (x, y, z) we use finite differences to compute the gradient using a forward voxel (resp. a backward voxel). For instance to compute the gradient G_x in the x direction, we use the voxel $(x + 1, y, z)$ (resp. $(x - 1, y, z)$). Since we compute the gradient in the 3 directions, we need to use 3×2 voxels. We decide to ignore border effects since we do not observe related artifacts.

Listing 4: Code excerpt 4

```
    // Pixel coordinates in the texture for the finite difference
    int u_x_fwd, v_x_fwd, u_y_fwd, v_y_fwd, u_z_fwd, v_z_fwd,
        u_x_bwd, v_x_bwd, u_y_bwd, v_y_bwd, u_z_bwd, v_z_bwd;

    pixel_coordinate(x + 1, y, z, u_x_fwd, v_x_fwd) ;
    pixel_coordinate(x - 1, y, z, u_x_bwd, v_x_bwd);

    pixel_coordinate(x, y + 1, z, u_y_fwd, v_y_fwd);
    pixel_coordinate(x, y - 1, z, u_y_bwd, v_y_bwd);

    pixel_coordinate(x, y, z + 1, u_z_fwd, v_z_fwd);
    pixel_coordinate(x, y, z - 1, u_z_bwd, v_z_bwd);
```

For a given direction x (resp. y or z), we decide to compute the gradient using the value of the channel R (resp. G or B). We also compute a normalization factor N_G:

Listing 5: Code excerpt 5

```
    // Finite difference
    double G_x = (getTextureR(u_x_fwd, v_x_fwd, dataVolume, width, height) -
                  getTextureR(u_x_bwd, v_x_bwd, dataVolume, width, height)) / 2.0;
    double G_y = (getTextureG(u_y_fwd, v_y_fwd, dataVolume, width, height) -
                  getTextureG(u_y_bwd, v_y_bwd, dataVolume, width, height)) / 2.0;
    double G_z = (getTextureB(u_z_fwd, v_z_fwd, dataVolume, width, height) -
                  getTextureB(u_z_bwd, v_z_bwd, dataVolume, width, height)) / 2.0;

    // Normalizing
    double N_G = sqrt(G_x*G_x + G_y*G_y + G_z*G_z);
```

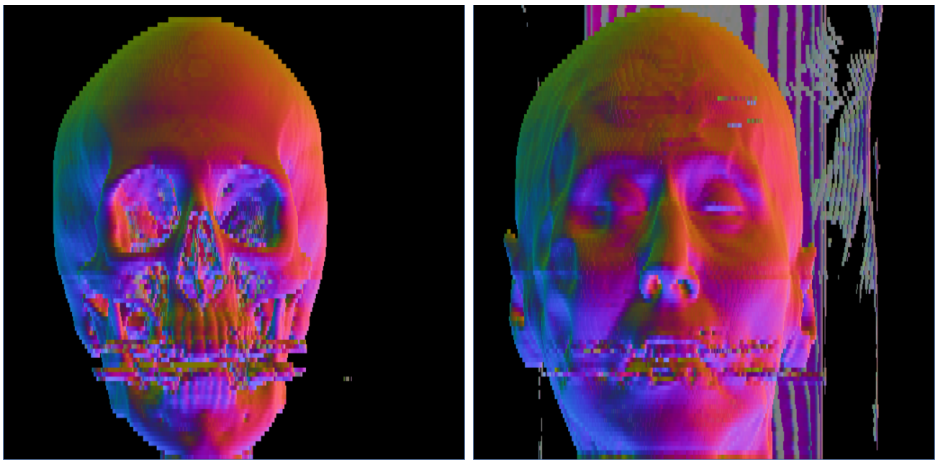
Then we map the computed values to $[0, 255]$ using the fact that the normalized surface normal $\frac{-G}{N_G} \in [-1, 1]$:

Listing 6: Code excerpt 6

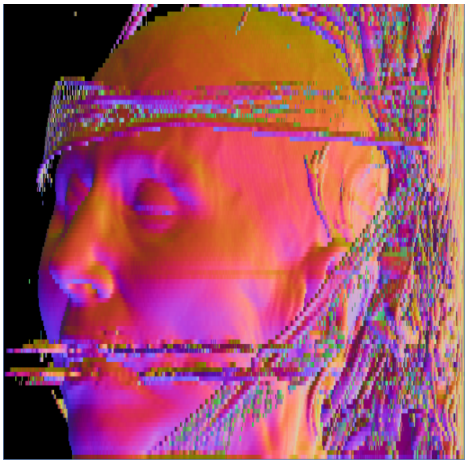
```
    // Mapping between 0 and 255
    unsigned char N_x = (unsigned char)((-G_x / N_G + 1.0) / 2 * 255.);
    unsigned char N_y = (unsigned char)((-G_y / N_G + 1.0) / 2 * 255.);
    unsigned char N_z = (unsigned char)((-G_z / N_G + 1.0) / 2 * 255.);

    // Assign values to dataNormals
    int u, v;
    pixel_coordinate(x, y, z, u, v);
    setTextureR(u, v, dataNormals, width, height, N_x);
    setTextureG(u, v, dataNormals, width, height, N_y);
    setTextureB(u, v, dataNormals, width, height, N_z);
```

Once in the Fragment Shader, we now display the RGB color given by the $dataNormals$ texture. The following Figure illustrates the obtained results. We can observe that "opposite" parts of the face (e.g. left/right, top/bottom) have "opposite" colors (e.g. blue/red, yellow/pink):



(a) Skull - $isoValue = 0.4$ (b) Face - $isoValue = 0.2$



(c) Face & straps - $isoValue = 0.07$

Figure 6: Iso-surface normals with different values of $isoValue$

3.2 Shading an iso-surface with a Phong model (Lim-Sylvie Pou)

In this part, we have first to define the light direction, the view direction, the normal vector and the reflected view direction in the shader.

We assume that the light direction and the view direction are the same. Indeed, we suppose that the light comes from the observer. Since the image initially obtained by ray-marching corresponds to the skull seen from the front, we create a light direction vector in the x and y coordinates. We can choose for example as the light direction vector the following one : $(x, y, z) = (0.8, 0.2, 0.0)$.

Moreover, we want the light direction to remain attached to the camera when we rotate

the volume. As we have done in the second project, we defined the light direction as the vector $(x, y, z) = (sin(\text{rotationAngle}), cos(\text{rotationAngle}), 0.0)$. In this way, as we rotate the skull, the image will still be enlightened.

We have defined the normals in the above part. However, the light coordinates are in [-1,1] whereas the pixel coordinates are in [0,1]. We have to map the coordinates from the gradient and we obtain the following vector :

$$(x, y, z) = normalize(2.0 * texture(myTextureSamplerNormals, pixCoord).rgb - 1.0)$$

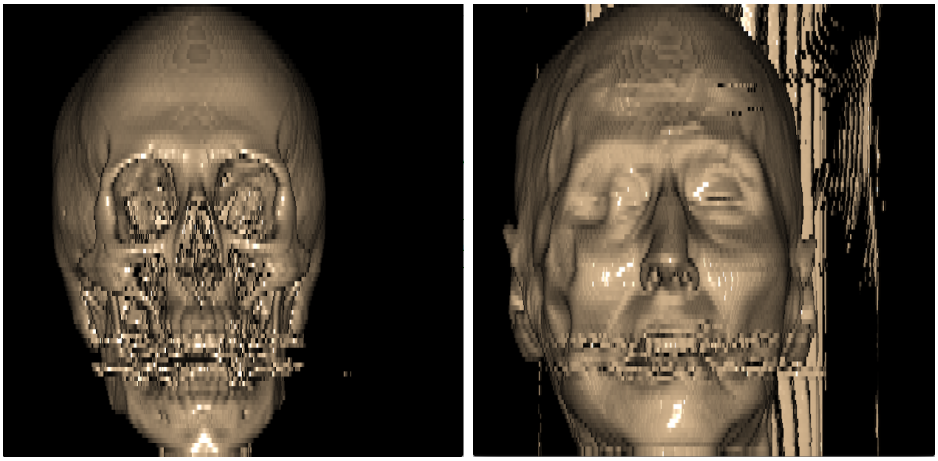
The reflected view direction is simply obtained by $(x, y, z) = reflect(-L, N)$, with L the light direction and N the normal vector.

Finally, we want to compute diffuse and specular shading with a Phong model. To control the diffuse color, the specular color and the specular exponent of the model, we created two global vectors *d_col*, *s_col* and the global variable *specular_exponent* in the file "volume_render.cpp".

Lastly, in the ray-marching loop, we substitute the color by :

$$color = max(dot(-L, N), 0.0) \times d\_col + max(dot(R, V), 0.0)^{specular\_exponent} \times s\_col.$$

We choose a tan brown color for the diffuse color and white for the specular color. The following figures illustrate the iso-surface with a Phong model with different values of the *specular_exponent* and the $isoValue$.



(a) $spec\_exp = 10 - isoValue = 0.4$ (b) $spec\_exp = 100 - isoValue = 0.2$

Figure 7: Iso-surface with a Phong model

Now we show a video highlighting the objective of the project, ie. the volumetric rendering of an MRI scan of a human head.

Figure 8: Animated GIF - Phong model with variation of the $isoValue$