In this third and last lab of Advanced Visualization we have learned about Volume Rendering. To do so, we switched from working with surfaces of meshes and specifically on how to illuminate them to focus on how to visualize a volume of our scene i.e., a three-dimensional region. All the images are shown with the default parameters except the ones indicated below them.

We have learned among others to work with 3D textures, to browse them with the help of an auxiliary mesh and a Ray-marching algorithm configuring ourselves the ray and using it to move through the volume and defining our result with our computations. To improve our engine, we have implemented techniques like jittering to avoid artifacts, a transfer function to visualize different densities of the volume, clipping by generating a plane and, finally, the implementation of isosurfaces to be able to see and shader the surface of the volume.

First, some changes were made to the framework with respect to LAB 2's state considering the difference in structure and content. First, since the volume is still represented as a node in the scene, in this case we changed the SceneNode constructor methods so now the mesh corresponds to a cube and the material to a VolumeMaterial and setting the properties of it (this will be explained later). Another thing related to the SceneNode class that has been changed is the *renderInMenu()* method so that in this case only shows options related to volume rendering, where we added a volume selector and an isosurface mode. Finally, we eliminated the option to add meshes and lights, so that in this case there will be only one of each and there is no skybox.

As usual we start by creating a subclass that inherits from StandardMaterial called VolumeMaterial (for the moment we will leave out the isosurfaces). There are a lot of attributes in this class, where beginning with the actual parameters for the computations there is a float for the intensity by which we will multiply the final color of each pixel, another float to define the length of each step of the ray, a Vector4 for the clipping plane and a threshold to decide at each step of the ray if we want to add the value to the result or not. In addition, we have added some flags as booleans to decide whether to apply jittering, the clipping plane and use the transfer function.

And now that the transfer function has been introduced, there is an actual set of attributes relating to it, where we have an integer to describe the number of different intervals the volume has and an integer to each limit, where each interval has its corresponding *Vector4* describing a color and its alpha value, where this colors will be used to create the transfer function texture using the *setTransferFunction()* method. This method consists in creating the imaged that will be used in the texture with a size of 129x1 with RGBA channels, where we cross each of the pixels setting the corresponding color in relation to with interval they belong using *setPixel()* and, finally, we create and set the texture corresponding to the transfer function.

About this, it is important to clarify the use of *Vector4* and *Color* to represent colors in the context of setting the transfer function, being this related to the fact that the ImGUI color selector uses a *Vector4* float vector in range $[0.0, 1.0]$ but the color that has to be used in *setPixel()* corresponds to a uint8 in the integer range $[0, 255]$. For this reason, the colors represented in the attributes are defined as *Vector4* so that they can be changed using ImGUI and, in order to set a pixel color, we define a *Color* multiplying each value by 255 so they are in the correct range.

Regarding the constructor of this class, only a simple definition is made, where we create the texture for the volume and set the shader as the *basic* and *volume* ones, where, finally, we assign default values to the step, intensity, clipping plane and threshold attributes, where the *step* is set to 0.01 because is the one that gives good results for every volume implemented without an extreme performance cost. That being said, in order not to always create a VolumeMaterial, *setVolumeProperties()* was implemented where, depending on the selected volume, it loads the corresponding volume and create its 3D texture as well as sets the threshold so that some noise in the result is not rendered and the intervals and colors for each transfer function are set as well.

In relation to the options in ImGUI, there is an option to apply jittering, we can activate and change the transfer function and we can set and activate a clipping plane as well as change the color, step length, intensity, and threshold.

Now we will enter in the shader *volume.fs* where we will start by configuring our ray. The initial position will be the point on the surface of the cube mesh where we are in that fragment. The direction will be a vector that goes in the direction from the camera to that point on the surface in local coordinates, transforming the camera position to local coordinates multiplying it by the inverse model matrix of the mesh and using the interpolated vertex position for the point in the surface and the initial *samplePos*. Finally, in order to define the vector of our step we will multiply the direction by the step length.

If the step length is too large, it could happen that some relevant information of the density of a layer in our volume was not detected because it is between two steps, and since all the rays have the same length, it could give us an erroneous result, introducing here jittering to reduce this errors. From our noise texture we will read a value depending on which pixel of the window we are in, dividing it by the size of the texture. The value obtained will be multiplied by the step vector and added to our initial position. That way each ray will start at a different position (but similar enough) avoiding losing a layer's information.

Once configured we can start to advance in our ray. We start evaluating if we want to paint that region of the volume or not according to the clipping plane. We will use the equation of the plane $ax + by + cz + d$ to evaluate in which side of the plane we are in each step and depending on the result ($results > 0$ have to be hide) we compute the value or make one more step in the ray.

If we are in a region that we must paint, we will proceed to read the corresponding value from our 3D texture. To do this we will have to change from local coordinate space to texture coordinate space, where our local coordinates go from $[-1,-1,-1]$ to $[1,1,1]$ and we want to change to texture coordinates that go from $[0,0,0]$ to $[1,1,1]$. For this we apply:

$$f(x) = \frac{x+1}{2}$$

The result will be a *vec3* that we can use to read in our *texture3D* the value of the density at that point of the volume (only the *x* value of the texture since it is in grayscale).

Once we have the density value, we can start the classification stage where two different modes have been implemented: a simple classification and coloring using transfer functions. For the first one it's simply (d,d,d,d) and, for the transfer functions, we used the LUT created by the framework mentioned before. We have created a LUT for each volume and they are one dimensional textures with a color and an alpha defined for each possible value of d. Depending on which densities we want to highlight or differentiate, a corresponding color or transparency is assigned. Related to each volume and being each color predefined with alpha 1.0:

Bonsai: Green color is mapped for the densities around the range of the ones corresponding to the leaves, where brown is for the wood and orange for the flowerpot.

Abdomen: Meat color has been selected for the densities of the skin, red for the organs and muscles and, finally, white for the bones.

Daisy: We didn't know what this was so only one layer was defined with lilac color.

Foot: Meat color was selected for skin densities and white for the bones.

Orange: Orange color was selected for the outside densities and yellow for the inside densities.

Teapot: Brown color was selected for the table densities, green for the actual teapot and red for what we think is a lobster, where it can be very differentiated from the green of the teapot.

The number of colors used and the ones selected are to give the best visual result to the initiation. It is important not to forget to multiply the color read in the LUT by the alpha read before modifying the final color so that transparent areas do not affect the result, where if we do not apply this, at the color accumulation in the composition (which now will be explained), the colors of the parts completely transparent will be accumulated too and will affect the result, which is incorrect.

Finally, regarding the composition, depending on whether the value of d is greater than our threshold or not, we would add the calculated color to the pixel color iteratively. Where this threshold has been implemented in a similar way of the *First* composition mode to avoid painting some parts like the least dense zones that are not part of the volume and are more like noise.

With this we would have completed the necessary calculations for a step in the ray. Now we would have to evaluate if we are in a case of early termination, that is, if we should take another step or if we should finish it now and define the value of our pixel. To do this we must check if the value of alpha is already 1.0 or greater, in which case no more color could be added and therefore it would be useless to continue, and we should also check if the position of the next step is inside or outside our auxiliary mesh, where taking into account we are using local coordinates we only have to check that the sample position is inside the range $[-1,-1,-1]$ to $[1,1,1]$.

Once we get an early termination and we get out of the loop the only thing left is to color our pixel with the result that we got from the ray, multiplied by the material attributes intensity and color to modify that result to our liking. With this we would have finished explaining the main pipeline of how to render the density of a volume given a 3D texture.

Now we should mention that in the ImGUI options there is also one to activate the isosurface mode, where once activated we will go to a very similar pipeline but with some extras. Isosurfaces are the points of the volume that have the same value, in this case density, and it help us to visualize the shape of that volume. Since it is a surface, to visualize it better we will need illumination, where the one used in this lab is the Phong model, but in the data of the volume we do not have the necessary information to calculate it. We need to know the normal of the surface. A value that will help us is the gradient since, like the normal, it is also perpendicular to the surface. The gradient can be approximated by taking samples of the points around a point on the isosurface and making partial derivatives with that data.

With that we will have the data needed by Phong's model. Implementing in this lab an illumination model helps us to visualize how the light would behave in our volume. In order to implement it, the first thing that happens when we activate the flag is that we change the material of our *SceneNode*, where the new material is *VolumeMaterialPhong* and in this case it is a subclass inheriting from *VolumeMaterial*, so it has all its attributes. In addition, we add the *Vector3* attributes for the Phong ambient, diffuse and specular components of the material and a float to control the shininess. We initialize these attributes in the constructor with the white color as default to not alter the result of the light components and a shininess of 256 to have a very sharp specular result. Finally, we will assign it our new shader: *volume_isosurfaces.fs*.

Before explaining the shader, it is important to remark that in order the set the corresponding uniforms, we will reuse the ones from *VolumeMaterial* where the material

components are added and, considering we are modelling illumination, the ambient light as well as the light position, diffuse and specular components are uploaded to the shader too, where only one light will be used. All the material parameters as well as the ones from the light can be completely changed in ImGUI.

Regarding the new shader contains practically the same as the *volume.fs*, where the changes start at the classification stage, after reading the density value in the 3D texture. Now, to decide if we want to compute a value or not, instead of using the iterative computation of the discretized volume-rendering integral composition scheme that we used before, where all the values we were interested in were added to the final result following the approximation of the integral, we will use the *First* composition scheme, where only one value will be saved in the final color, which will be the first value that is higher than a threshold chosen by us.

Once we have decided that we do want to calculate the candidate value to be part of the final pixel value we will start by approximating the gradient ∇f at the point using the following equation:

$$\nabla f(x,y,z) \approx \frac{1}{2h}\begin{pmatrix} f(x+h,y,z) - f(x+h,y,z) \\ f(x,y+h,z) - f(x,y+h,z) \\ f(x,y,z+h) - f(x,y,z+h) \end{pmatrix}$$

Where x, y, z will be the position where we are in the ray, $f(x,y,z)$ will be to read the d value at that position (adding or subtracting h and changing from local coordinates to texture coordinates) in the 3D texture and h corresponds to the step in relation to calculate the gradient. This value has been selected taking into account the different complexities of the volumes used, where for volumes that need a lower step length to be perfectly visualized like the abdomen the value for h need to be lower (like 0.001) to be able to take into account the little imperfection, but for simpler volumes like the teapot, a very low h can produce very blocky results, so a values of 0.01 was found to be very optimized for each volume, where a high value will be a bad election for every possible volume. Anyway, it can be changed, and the negative result of the gradient will be used as the normal of our surface.

After that, we will proceed to apply Phong equation beginning with the surface color, where if the transfer function mode is selected it will be used and if not, *u_color* will be used. Now that we have color, normal and position of the surface, plus a light, its position, and attributes, we can apply Phong equation at its entirety, where considering that the position of the isosurface to paint is in local coordinates, we will multiply it by the model matrix of the mesh in order to transform it to world coordinates in order to correctly apply Phong equation.

Once the Phong illumination value is computed, we assign the color to the final pixel with the alpha as 1.0 so there are no transparencies. When the last value that meets the condition is computed, its result will be saved as the final color. Once we encounter an early termination, we can assign the pixel value and finish. Regarding the nature of Phong equation, in this case the brightness value does not have any effect. There's also an option to visualize the normal vectors instead of the illuminated surface.

Finally, cases where part of the isosurface is behind another are very important if the one in the front is not completely opaque. The problem here is because we are using a mesh there are problems with the alpha blending inside of it, so if an alpha value is set, this part will look flat and behind the others. So, in this case to simplify and not have this problem, if the alpha is different than 1.0, these parts will not be painted, so that we can stop the ray computation after the isosurface is painted and consider transparencies.

Another thing to consider is the process to scale the cube in order to correctly represent the volume, where in this case the corresponding size in each dimension is multiplied for the corresponding spacing, having then the actual size of the volume where, in order to scale it, every size will be divided by the maximum of each so that the *1.0* in scale will be this maximum and the rest will be scaled accordingly.

Regarding all the examples shown is interesting to look at and see the problems that can be seen, where the first and most important is that, due to similar values in density, in order to apply the transfer function to an isosurface, some colors can appear in parts where they do not belong, being a possible solution (in cases like the bonsai) to use a higher step length to have a little bit less precision, but some other errors can appear because of the lower precision. Another thing is that in many cases the noise at lower densities is mixed with the actual volume and it is very difficult to remove by the means of this LAB, where the isosurface level corresponds to the threshold in *VolumeMaterial*. Finally, to avoid finishing in the middle of the volume, the maximum iterations value has been set to 100000 so it always ends with early termination.

As a final optimization, a volume manager has been implemented in the *Volume* class to store all the loaded volumes so that they do not need to be loaded when changing volumes, achieving a better performance where every volume will be loaded at the beginning of the execution. This volume manager consists in an *std::map<std::string, Volume*>* where all the loading process has been implemented in the *Get()* method, which has different loaders for the different volume formats.

Now it's time to try it yourself in the application!