Real-Time Label Visualization in Massive Model Objects

Sibgrapi paper ID: 99999

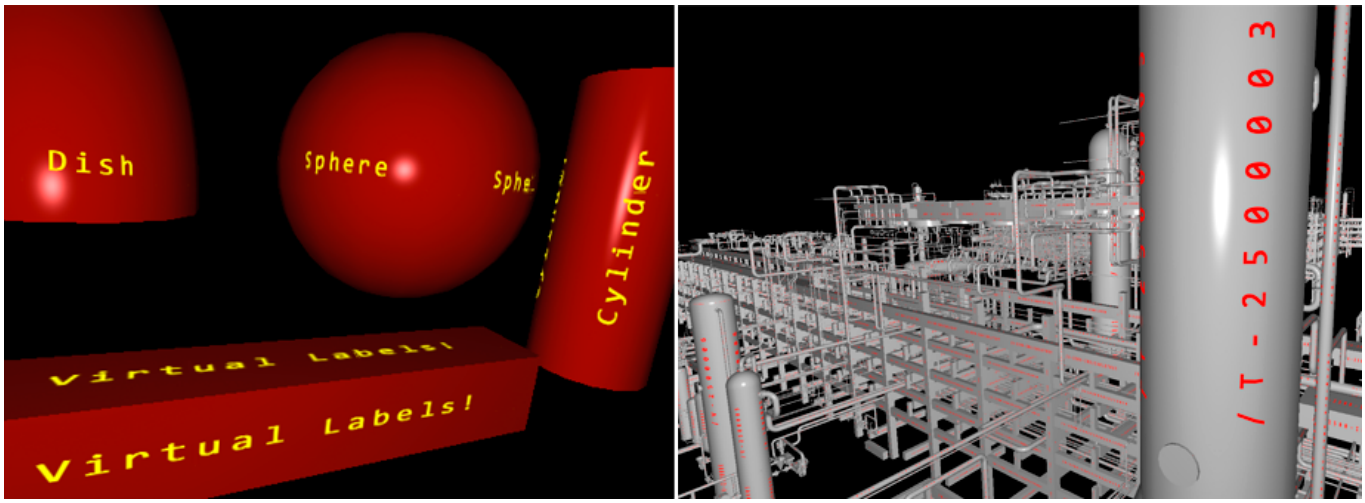

Fig. 1. Different types of objects with labels (left), CAD model of an oil refinery with coloured labels (right).

***Abstract*—Virtual Labels are used in computer graphics applications to represent textual information arranged on geometric surfaces. Such information consists of names, numbering, or other relevant data that need to be noticed quickly when a user scans the objects in the scene. This paper focuses on the so-called massive models, which have a large number of geometric primitives whose rendering presents a high computational cost and for which conventional texturing techniques can extrapolate the available computational resources. In this work we have developed a way to view, in real time, virtual labels with different information on the surfaces of objects in massive models. The technique is implemented entirely on the GPU, and shows no significant loss of performance and low memory cost. CAD models' objects are the main focus of the work, although the solution can be used in other types of objects once their texture coordinates are adjusted correctly.**

***Keywords*-Surface labeling; Real time visualization; Massive Models.**

I. Introduction

Virtual labels are textual information displayed on the surfaces of virtual objects. Such information consists of names, numbers, and any relevant information that needs to be identified when the user scans the objects in the virtual scenario. Labels could be used in games, maps or any other graphical application which renders 2D or 3D objects and needs to show text on those objects' surfaces. This paper focuses on real-time 3D visualizers of models with a large number of objects, like CAD (Computer Aided Design) models used for scientific visualization such as an oil refinery.

In industrial plants there are structures that contain labels applied to their surfaces, which display the names of these structures or other information about them, in order to help professionals to easily identify them in fieldwork. The same way that these labels help engineers identify the real structures, virtual labels can help users in identifying objects in visualization software in real time.

The main goal when displaying labels on the surfaces of virtual objects is to help the user to quickly identify relevant information about the objects being displayed on the application, as well as to provide support for guidance in the virtual scenario. In CAD visualization systems, it is usual to have a list showing the names of all the objects present in the scene. Thus, to find the name of an object, the user needs to click on the object and check in the list for the selected name, which is a more laborious process than reading labels directly on the objects. Moreover, large industrial plant models are typically massive, so called when they present great complexity, such as an oil refinery consisting of millions or billions of objects [1].

For this kind of model, there are some challenges for rendering virtual labels; one of them is related to the existence of a large number of objects with distinct names. To illustrate this problem, if a simple texturing method is used, it could be necessary to build a different texture for each object (because each object has a different name). Such an approach implies a large use of memory space that can exceed the capacity of the video card. To solve this problem, it is necessary to develop

strategies to keep in video card memory a limited number of textures per frame. It is also possible that all textures could not be stored in main memory at the same time and thus they would need to be reconstructed to form the words of a label in real time, which can affect the performance of the application.

Even in a scenario where one can store a different texture for each object, there is still the need to perform multiple texture context switches at every frame, which can become the bottleneck of the graphical application [2]. Another challenge is the positioning of labels on objects with different geometries and sizes, making it necessary that the virtual labels are automatically positioned by the application in all cases.

These problems (excessive use of video memory, many texture context switches per frame, and the positioning of labels on different surfaces) encourage the exploration of solutions for the real time display of labels in massive models. Since the visualization of massive models has itself a high computational cost, it is necessary to devise a solution that does not worsen the performance and spends the least possible amount of memory.

This paper presents a technique for displaying virtual labels in massive models implemented entirely in the GPU. This technique does not cause significant performance loss regardless of the number of objects that display labels and consumes little video memory with labels coded and stored in a buffer in the graphics hardware. The technique relies on a two-pass algorithm. The first pass performs the rendering of the scene and stores texture coordinate information in off-screen buffers. The second pass uses the information from these buffers to map the labels on virtual objects, calculating the final color of each pixel of the screen individually by sampling a texture that contains all characters.

This paper is organized as follows. Section 2 presents work related to the rendering of textures with textual information. Section 3 describes the technique developed in this work and Section 4 discusses how to position labels on some CAD objects. Section 5 shows the results and Section 6 concludes the paper.

II. Related work

There are only a few works that directly address the problem of real time text rendering in massive models. This section briefly discusses some of these works and shows a few others that address more general problems regarding text rendering (such as label positioning, memory management, and aliasing) and that relate them to issues if applied to the case of virtual labels in massive models.

The commercial software Aveva Review [3] can display labels on the surfaces of objects from CAD models, which have unique names. However, only cylinders and boxes display labels and not all of them show their labels at the same time — there is a strategy to select which objects display their names at each frame; apparently, it prioritizes the ones with a larger size in screen space.

Something that can make an application limit the number of objects that display labels is the number of texture context switches that may be required at each frame, since each object has a different name. To solve the problem of multiple texture context switches, NVIDIA [2] explains the concept of a "texture atlas" showing the benefits of grouping images into a single texture called an atlas, and using texture coordinates to access the relevant sub-rectangle of this atlas. In this paper we use an atlas that stores all the required characters, and the labels need to be built from this atlas.

There are other works that do not directly address massive models, but address issues related to text rendering. Qin, McCool and Kaplan [4] present a way to perform text rendering on three-dimensional objects using a vector graphics image representation of text, so that characters show very little aliasing when magnified. They use a strategy similar to the one in "Texture Sprites" [5] to place the words on 3D objects. However, they use a quadtree in the CPU to store a table of characters with different resolutions depending on the complexity of each character and a "sprites table" to arrange the words. The table is encoded in the graphics hardware as a texture, so that it can be sampled directly in the GPU. As in "Texture Sprites", the technique is presented for a single object at a time.

Cipriano and Gleicher [6] focus on text positioning on surfaces with sharp curvatures and even with holes. For this, they create meshes, called scaffolds, to store the letters and then place them so that they float over objects. A single atlas is used to store all the characters and the texture coordinates of the scaffolds are configured in order to obtain the relevant characters. One difficulty in using this solution is the introduction of new geometries in the scene, which may be undesirable in massive models rendering, especially when all objects need to display some text.

The use of virtual textures as an out-of-core solution for textures that occupy a very large space and cannot be entirely stored in RAM or VRAM was explained by Barret [7] and Mittring [8]. A "virtual texture" is stored on the hard drive and split into parts of equal size, called pages. Only the necessary pages for the graphics API are transferred to memory and stored in a "physical texture" as called by Barret. To map the virtual pages into physical pages, a page table is built.

In the case of virtual labels, they would need to be built as textures in pre-processing and arranged in a virtual texture, which could solve the problem of memory. The problem lies in how to organize the virtual texture, because the names of the objects vary in size and it would require that the page size (which is the same for all pages) be sufficient to contain the biggest name. This would imply pages with unused space, which is a waste of memory. Another way to organize the virtual texture would be with a label occupying more than one page, and objects therefore needing more than one page to store their textual content. The solution adopted in the present work took another approach, creating only the necessary labels (those that will appear on the screen) at each frame and spending little memory to save them, as will be shown in the next section.

III. Applying Labels to Objects

This section explains the procedure for displaying labels on a large number of objects and the writing of words on geometric surfaces in real-time. To do this, a two-pass algorithm in the GPU was developed. For the implementation of this algorithm we used OpenGL 2.1 [9], OpenGL Shading Language 1.2 [10] and a few extensions.

The first pass of the algorithm renders the scene to an off-screen color buffer and also creates a second buffer with texture coordinates per pixel and an identifier of the label which must be displayed on the geometry to which the pixel belongs. This information will be useful for the second pass when the pixels' final colors are calculated and labels are added to the scene.

To calculate the final colors of the pixels, it is necessary to know whether they should show part of a label and which textual information that label contains. For this reason, before the rendering takes place, the ASCII codes of all labels must be stored and made available for the GPU. In this work we used a 2D texture for the label's ASCII codes that is referred to as a table, relating the label index with its text.

The number of texels that a 2D texture can store is hardware dependent, but in general there is enough space for a large quantity of textual information (e.g., 256MB for a GeForce 9600GT and even more for modern graphic cards). Of course, considering that the models to be visualized are massive and their geometries can consume plenty of space in the video memory, it is not be desirable to spend that much with labels. However, if each label has 20 characters and 2,000,000 distinct labels are needed, the memory consumption will be 38MB, what isn't high considering the large amount of information stored.

Also, a texture atlas that contains all the required characters must be available in the GPU, which is treated as a two-dimensional array of characters. The atlas is an image with the graphical representation of the characters arranged in the same order of the ASCII encoding, so that it is possible to find, in that image, the position of a character by its row and column initials.

Using the atlas with all characters and the texture with ASCII codes created by the application, two rendering passes are performed to render the scene with labels applied to the geometries. We are going to explain them in detail in the sub sections below.

A. First rendering pass

Before the drawing commands of the objects are called, the CPU informs the GPU of the label identifier that must be displayed on each object (index in the texture of ASCII codes) and the number of characters of that label.

The first pass is necessary for the generation of texture coordinates for the objects. We generated the texture coordinates in the vertex shader, as will be explained in section IV. The fragment shader writes information into two off-screen buffers that are textures attached to a Frame Buffer Object (FBO). In the first buffer, the color data of the scene are written, and in the second buffer, texture coordinates interpolated per fragment, the label identifier, and the label's text size are written.

B. Second rendering pass

In order to obtain the data per fragment generated by the first pass, it was necessary to draw a screen-sized quad, whose texture coordinates range from 0 to 1 in both dimensions. This makes it possible to sample the color buffer and the buffer with texture information.

After setting new information of color and texture to a fragment of the quad, the fragment shader performs calculations to add labels to the scene. If the fragment has no label, it receives, in the first pass, a label identifier equal to zero and texture coordinates equal to zero, so that no calculations for labels are done in the second pass. The calculation for mapping labels on an object is done as follows for each fragment that has a label.

1) Find character index : The algorithm considers that the label will be written parallel to the s axis of the object's texture space. This space can vary from 0 to 1, or to a value greater than 1, to repeat the label along s. For each interval of size 1 at the s axis, one must consider that the interval will be divided in a total of columns equal to the number of characters in the label. The purpose of this step is to figure out in which column the current fragment is contained, which indicates the index of the character in the object's label.

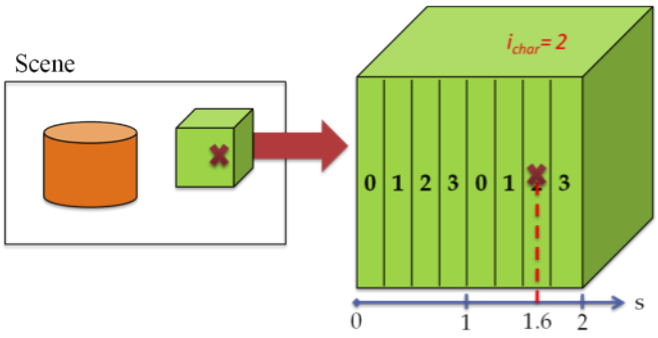

Fig. 2. Fragment with character index = 2.

For example, in Fig. 2, for a label with 4 characters ($n_{chars} = 4$), given a fragment with texture coordinate $s = 1.6$, the expected result is a character index $i_{char} = 2$. For that, we use equation (1), where s_{frac} indicates the fractional part of texture coordinate s. This fractional part indicates the offset of the fragment within one instance of the label, and is important for dealing with label repetitions.

$$i_{char} = floor(s_{frac} * n_{chars}) \quad (1)$$

2) Find the ASCII code of the character : As explained before, there is a texture that stores the ASCII codes of the characters of all labels in the scene. The goal of this step is, with the index of the character found in the previous step and

the label identifier of the current fragment, to find, within this texture, the ASCII code of the character sought.

Fig. 3 shows an example of how the characters are stored in the texture with textual information that, in this case, uses two texels to store each label ($texelsPerLabel = 2$). Since a texel has four color components, in two texels it is possible to store up to eight characters. The values in green on Fig. 3 are the label identifiers and in red are the indices of each character within its labels (highlighted in the figure, the index found on the previous step $i_{char} = 2$). The values in blue show that the texture in the given example possesses four texels horizontally and four vertically and, at the same time, represent an index for each texel along the s and t axes. As each label in the example occupies 2 texels, it is possible to store 2 labels per row of the texture ($labelsPerRow = 2$).

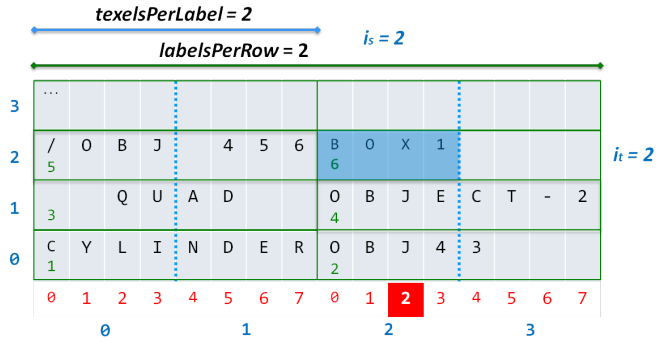

Fig. 3. ASCII codes stored in a 2D texture.

To obtain the character's ASCII code, it is first necessary to find the texel containing the character. In the example of Fig. 3, as highlighted, for a fragment in which label identifier (id) is 6 and $i_{char} = 2$, we will find the texel with horizontal index $i_s = 2$ and vertical index $i_t = 2$ (values in blue on the figure). These calculations are done according to equations (2-3) below:

$$i_s = ((id - 1) \bmod labelsPerRow) * texelsPerLabel + floor(i_{char}/4) \quad (2)$$

$$i_t = floor(\frac{id - 1}{labelsPerRow}) \quad (3)$$

Using the indices above, the texture coordinates of the texel must be found so that the 2D texture can be sampled, according to equations (4-5), where w and h are, respectively, the width and height of the texture, in texels.

$$s_{temp} = \frac{i_s + 0.5}{w} \quad (4)$$

$$t_{temp} = \frac{i_t + 0.5}{h} \quad (5)$$

The purpose of equations (4) and (5) is to find the texture coordinates whose values represent the texel's center, and with a nearest neighbor filter, the texel can be correctly retrieved from the texture. The obtained texel contains the correct character in one of its RGBA components. To finally find the ASCII code in the correct component, the calculation shown in equation (6) is done, with i_{char} mod 4 an index that goes from 0 to 3 due to the RGBA components, and $char$, the ASCII code we are looking for:

$$char = texel[i_{char} \bmod 4] \quad (6)$$

In the example shown in Fig. 3, the expected result will be the code of the 'X' character.

3) Sample the character atlas: Knowing the character's code, it is then necessary to find its graphical representation in the texture atlas containing all characters. More specifically, the correct texel within the bounds of a character's image in the atlas must be found. For that, the offset of the fragment in the geometry's texture space must be calculated and related to an offset in the atlas, as shown in Fig. 4.

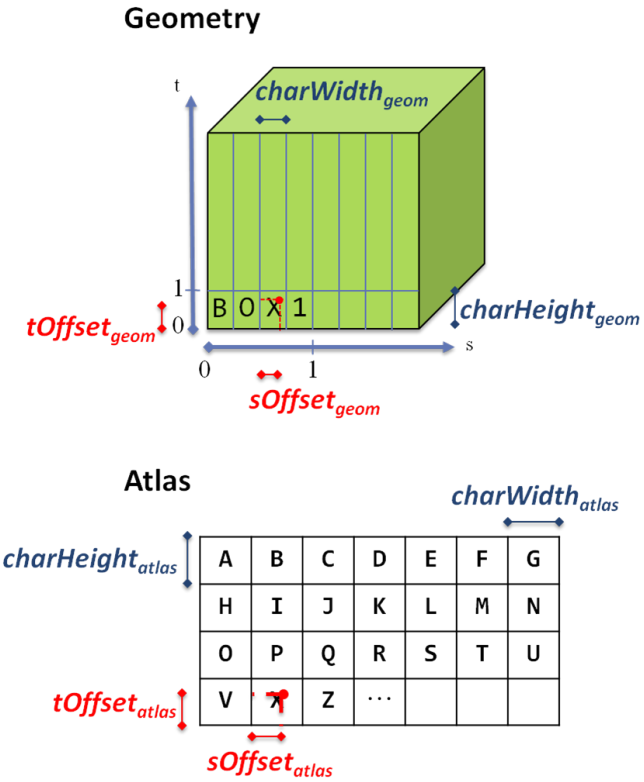

Fig. 4. On top, a fragment and its offsets from the beginning of the character on the object. Below, the related offsets in the atlas.

To calculate the offset along the s axis in the geometry texture space ($sOffset_{geom}$), at first, one must calculate the width of the character in a 0 to 1 range. This value ($charWidth_{geom}$) is found simply by dividing 1 by the number of characters in the label. The offset is calculated with equation (7), using once again, the fractional part of the

s coordinate of the fragment (s_{frac}) and the character index obtained previously (i_{char}).

$$sOffset_{geom} = s_{frac} - (i_{char} * charWidth_{geom}) \quad (7)$$

The calculation of the offset along the t axis ($tOffset_{geom}$) is simpler, since the height of the character ($charHeight_{geom}$) is always 1, even considering the vertical repetition of text. This way, $tOffset_{geom}$ is simply the fractional part of the t coordinate of the fragment.

To find the related offsets in the atlas's texture space, it is necessary only to multiply the offsets found in the geometry's texture space by the ratio of the character's size in the geometry's texture space to the size of the same in the atlas's texture space, a value known since the creation of the atlas. This way:

$$sOffset_{atlas} = sOffset_{geom} * \frac{charWidth_{atlas}}{charWidth_{geom}} \quad (8)$$

$$tOffset_{atlas} = sOffset_{geom} * charHeight_{atlas} \quad (9)$$

In equation (9), the denominator of the fraction doesn't appear because, as we mentioned earlier, $charHeight_{geom}$ is always 1. The offsets calculated with equations (8-9) must be added to the character's initial position in the atlas, to find the texture coordinate and then to sample the atlas in the correct position to obtain a color for the current fragment.

With the color obtained from the atlas and the color of the object (which is stored in the color buffer created in the first pass) it is possible to calculate the fragment's final color treating the label as a sticker.

With the presented technique, it is noticeable that, independently of the number of objects loaded by the application, label calculation is needed only for the pixels of the screen, which justifies the use of the technique for massive models. In the second pass, some equations with simple operations are calculated, and a few texture accesses are needed. In relation to these accesses, the algorithm benefits from hardware resources for texturing, since the fragments next to each other, that are processed in parallel, in most cases, need to access near parts of the textures. The technique for displaying labels did not present any significant performance impact with massive models, as will be shown in Section V. The next section discusses the way to position labels in objects of CAD models.

IV. Label Positioning

In this section we are going to explain the procedure to position labels on some surfaces of CAD models, with the goal of providing to the user a good way to visualize the text, independently of the angle of vision. We are going to show that, with the correct definition of texture coordinates, the algorithm for displaying labels works for objects of different shapes and sizes. The proposed algorithm considers that the texture coordinates of the geometries must be defined during the first pass. Those coordinates can vary from 0 to 1, or from 0 to any value greater than 1 to repeat the text along the geometry.

For all presented primitives, the texture coordinates are generated in the vertex shader of the first pass. Such procedure can be done because the developed graphics engine uses a primitive instantiation technique, which shapes the different geometries in the vertex program from a grid of vertices built in the CPU. For that, the vertex program needs to know information regarding the dimension, scale, position and orientation of the object and those data need to be informed by the CPU before the drawing of each object. The scale and dimension data are essential to automatic generation of texture coordinates for the geometries with different shapes and sizes, and for having labels with a varying character quantity.

Firstly, the technique was tested in simple geometries, such as quads — since it was easy to extend the idea to other objects. Texture coordinates can be defined in several ways, but it was concluded that it would be better to centralize the labels vertically on the objects so that the visualization of the 3D models wouldn't be polluted. Horizontally, it would be better to repeat the text as many times as necessary, so that the user is able to visualize the text or easily knows how to find it from any position from which he/she is examining an object.

To centralize the text vertically, only a determined area of the object should have coordinates varying from 0 to 1. Thus, the fragments whose texture coordinates are outside that interval are identified in the fragment shader of the first pass and associated to a label identifier equal to zero — which makes the second pass to calculate the final color of those fragments without the mapping of labels. The procedure to centralize the text calculates the ideal number of times that a label must be repeated along the surface (considering the object's size). Initially an ideal number of label repetitions must be defined based on a visual observation of a unit quad, and then, those parameters will provide a base for the automatic texture coordinate generation for quads of any size — for instance, if it was noticed that vertically it was ideal to repeat the text 3 times so that the label wasn't stretched or squeezed, then in a quad of size 5 the text could be repeated 15 times. However, text should be displayed only at the center of the object; therefore, the texture coordinates should be shifted from 0 to 15 to -7 to 8, totaling 15 repetitions, in such a way that the area which goes from 0 to 1 stays exactly in the center of the object.

To repeat the text horizontally it is also necessary to visually identify how many characters fit in a unit quad to obtain a good visual effect. For instance, if it is identified that a unitary quad fits 6 characters, then a quad of size 5 fits 30 characters. If a label has, for example, the text "Quad 0" of 6 characters, then it is possible to repeat the text 5 times along the s axis. Therefore, knowing a base for unit quads, it is possible to create simple equations for quads of any size.

One thing that has also to be considered for the calculation of the number of repetitions is a relation between the width and

height of the objects. If the scale to be applied to the texture coordinates is calculated separately for each dimension, the bigger the geometry, the bigger is the disproportion between the object's size and the label's size. If a quad has dimensions 1 x 1 or 100 x 100 the screen area occupied by the label is the same, but it could be larger in the second case when an equal increase in both dimensions of the text's area is desirable, and no label repetition is actually needed for either dimension. Repetition of text is only needed when one dimension of the object is bigger than the other. Therefore, the ratio between the object's dimensions must be considered for a better label display. If the ratio between width and height is 1, then a scale of 1 must be applied to the s and t coordinates. However, if the width is 100 times bigger than the height, the scale that should be applied to s is around 100 times greater than the one that should be applied to t. By taking into consideration the ratio between the object's dimensions, there is no disproportion between the label and the object's size (Fig. 5).

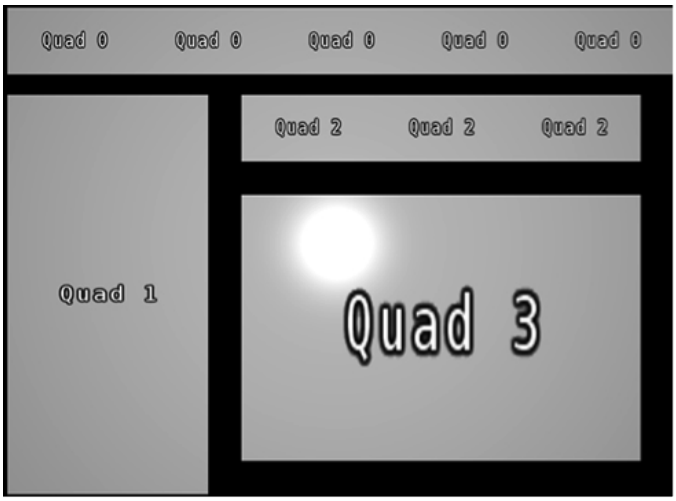

Fig. 5. Quads of different sizes and automatic positioning of labels.

Other primitives that have their texture coordinates adjusted to display labels are boxes, cylinders, spheres and dishes. In the case of boxes, the calculations are about the same as the quads, however, different texture coordinates must be defined for each of the box's faces.

For cylinders, the texture coordinates of the vertices are set so that the s-axis of the texture space is parallel to the y-axis of the object space — axis along the height of the cylinder — and the t-axis is parallel to the x-axis. With this, the text is repeated several times along s depending on the height of the object and, around the cylinder, it was agreed that the text should be repeated four times, so it could be easily visualized no matter which side of the cylinder the user is looking at. The calculation for finding scales, which must be applied to the texture coordinates, and depends on the cylinder's size and the number of characters of the labels, is all based on the calculations presented for quads.

In the case of spheres, text is displayed only once over t, on the object's center, and it can be repeated several times along s depending on the sphere's diameter. The primitive called dish, as found in CAD models, has the shape of a half-ellipsoid and, with its information of radius and height, it is possible to parameterize its surface. Fig. 1 (left) shows boxes, cylinders, spheres and dishes with labels.

The positioning of labels on objects like cones and pyramids could be a bit more complex due to the different dimensions of the bases of those objects. With these, the texture coordinates must be calculated in a way to compensate that difference, so that the text won't be stretched proportionally to the size increase of one base in relation to the other. Despite that, the calculation has not been done until the present moment, and, with the technique explained in this work, the positioning of labels is possible with more primitives than in the related works [3] — which allow the labels to be displayed on cylinders and boxes only.

V. Results

The results are going to be presented in two parts. Initially, we are going to show the result of the technique applied to CAD models. Then, we are going to present performance tests to show the effectiveness of the proposed technique.

A. Labels in CAD Models

The display of text with the technique presented in this work can be used for different purposes, but the motivation to create it was to display labels on massive model objects and more specifically, on CAD models.

For the rendering of CAD models we built an engine that uses a primitive instantiation technique, explained in the previous section. The engine renders all CAD model objects, placing labels on the previously mentioned objects: cylinders, boxes, spheres and dishes.

The models used are based in files, which contain information for each object to be drawn (position, orientation, scale and parameters specific to each primitive). Those object parameters are sent to the GPU as texture coordinates, since that was the fastest way found, considering the limitations of OpenGL 2.1 and GLSL 1.2. Besides that instantiation technique, the engine uses no other optimizations, but, with the built engine we can render a large number of objects at interactive rates, as will be shown in the next section.

Fig. 6 shows labels applied to the objects of an oil refinery CAD model. As it can be noticed, it's possible to display labels on a large number of CAD model objects and, also, that most of the geometries that exist in that model are boxes or cylinders.

The character atlas can have several variations to facilitate the reading of labels. For instance, in Fig. 6, white letters with black borders were used, which can be good to contrast the labels with different background colors. It is also possible to use an atlas with a black color for the characters, making it simple to modify that color within the fragment shader of the second rendering pass when the texture is sampled. Fig. 1 (right) shows colored labels on a CAD model, from an atlas with black characters with an alpha component, whose color was changed in the GPU.

Fig. 6. CAD model of an oil refinery with labels.

Fig. 7. Test scene with 100,000 quads. Version with labels (1920x1200).

B. Performance Tests

The technique presented in this work, for the display of labels on geometric surfaces, uses two rendering passes, and the first adds little complexity to the CAD graphics engine — in the first pass, calculations are added only to generate the texture coordinates and for the fragment shader to write information on two buffers instead of one. In the second pass several per pixel operations are executed and also some accesses to different textures. Therefore, it's important to evaluate the performance of the proposed technique, especially when the application utilizes higher resolutions. For that, tests were done and are presented below.

For the first tests, a scene with quads with random dimensions and positions was created, with texture coordinates generated in the simplest way, so that labels occupy the entire area of the objects and, therefore, calculations related to labeling were done for all the quads' pixels. The object's labels had different characters and sizes. The same scene was used to compare the performance of the versions with and without labels, whereas in the first case the algorithm does just the first pass and doesn't perform any calculation related to labeling, not even the generation of texture coordinates for the geometries. One of the created scenes, with 100,000 objects and resolution of 1920x1200 is shown in Fig. 7. It is worth mentioning that labels have whitespace characters before and after the text, and that the entire surface of each quad has a label.

For all tests we used a computer with the operational system Windows 7 64 bits, an Intel Core I7 870 processor with 2.93 GHz, 8GB of RAM and a GeForce GTX 580 graphics card.

Table I presents the comparison of rates in frames per second (FPS) obtained for a fixed camera position, for scenes with different number of objects, at a resolution of 1920x1200. One can note that the difference of FPS is small in scenes with fewer objects and virtually nonexistent in scenes with more objects.

TABLE I
COMPARATIVE PERFORMANCE TESTS FOR A FIXED CAMERA POSITION WITHOUT AND WITH THE USE OF LABELS (RESOLUTION: 1920X1200).

Number of Objects	FPS (without labels)	FPS (with labels)
10,000	457	439
50,000	106	105
100,000	54	54
200,000	27	27
500,000	11	11
1,000,000	6	6

Table II shows the result of another random scene of quads, but this time we measured the average frame rate in FPS obtained when navigating throughout the scene for 50 seconds. The exact same camera path was used for the scenes with and without labels. Tests were performed in two resolutions: 600x400 and 1920x1200 and it is once again shown that the performance difference tends to be null for a scene with a larger number of objects. In scenes with fewer objects, the difference reaches 5% but the rates are still very high. There have been some cases with the 600x400 resolution in which the resultant average was slightly higher when using labels. Such a difference is insignificant and can be explained by the interference of events external to the program, like processes of the operating system.

TABLE II
COMPARATIVE PERFORMANCE TESTS FOR CAMERA TRAJECTORY WITHOUT AND WITH THE USE OF LABELS.

Number of Objects	Resolution	FPS (without labels)	FPS (with labels)
10,000	600 x 400	511.6	485.3
	1920 x 1200	488.1	470.1
50,000	600 x 400	106.7	107.9
	1920 x 1200	102.0	99.9
500,000	600 x 400	10.6	10.6
	1920 x 1200	10.6	10.5
1,000,000	600 x 400	4.5	5.3
	1920 x 1200	5.4	5.4
1,200,000	600 x 400	4.6	4.5
	1920 x 1200	4.5	4.5

The last tests were performed with a CAD model of a

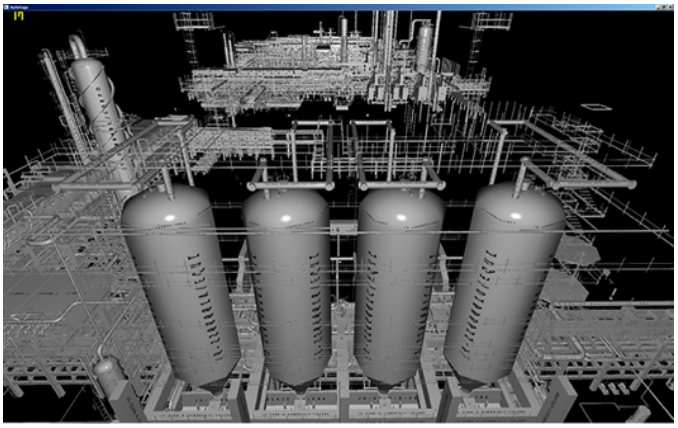

Fig. 8. Test scene with CAD model.

TABLE III
COMPARATIVE PERFORMANCE TESTS FOR CAMERA TRAJECTORY WITHOUT AND WITH THE USE OF LABELS IN A CAD MODEL.

Number of Objects	Resolution	FPS (without labels)	FPS (with labels)
300,000	600 x 400	15.8	15.5
	1920 x 1200	16.1	16.1

real oil refinery, rendering all the existing primitives in the model, except for triangle meshes. Cylinders, boxes, spheres and dishes exhibit labels and the total number of rendered objects was 300,000, mostly boxes and cylinders, as shown in Fig. 8. The texture coordinates of the objects were defined as explained in section IV, being generated automatically depending on the size of the objects and the number of characters of the labels. For a fixed camera position, shown in Fig. 8 we obtained 17 FPS for a scene with and without labels. The results of the tests when navigating the scene with the same camera path for a scene with and without labels can be found in Table III , which shows that the performance loss was little or none with the use of labels.

VI. CONCLUSION

In this paper we developed an algorithm entirely in the GPU to display, in real time, labels on the surfaces of geometric primitives of massive models, and found that it presents good performance and visual results. One of the challenges identified was that, when dealing with massive models, each of its objects could have a different textual content needing distinct labels. Therefore, if we used a different texture for each label, multiple texture context switches would be needed by frame, which would certainly severely compromise the performance of the application. Likewise, the use of one texture per object would imply a high memory cost and the development of strategies to maintain, in video memory, a limited number of textures per frame.

With the algorithm developed in this paper, only the labels that need to be shown on screen are built per frame, in the GPU, and the label mapping is done for each fragment by sampling an atlas with the graphical representation of the characters and a buffer with the ASCII codes that contain textual information of the labels. Therefore, we don't need to deal with problems related to texture context switches and also the problem of building and storing textures with the images of each label in video memory before the objects are rendered. We need only to store the atlas and the buffer with ASCII codes. The storage of ASCII codes uses much less memory than storing the images of labels.

The problem of positioning labels on objects of CAD models with different sizes and shapes was solved for cylinders, boxes, dishes and spheres. It was identified that, for the user to be able to see the labels independently of how the camera points to an object, text repetition was a good choice.

As a future work we want to add labels to other kinds of objects from CAD models, such as cones and pyramids, and also to improve the visual quality of labels with the use of anti-aliasing techniques. We intend to study the use of dynamic labeling, to display operational data such as temperature and pressure directly on the objects' surfaces. We also want to explore the use of labels in domains other than CAD, once the technique presented here for the generation of virtual labels is not restricted to the CAD domain.

REFERENCES

[1] S. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, *Real-time Massive Model Rendering*, ser. Synthesis Lectures on Computer Graphics and Animation. Morgan and Claypool, August 2008, vol. 2, no. 1.

[2] NVIDIA Corporation, "Improve batching using texture atlases," https://developer.nvidia.com/sites/default/files/akamai/tools/files/Texture_Atlas_Whitepaper.pdf, 2004.

[3] Aveva Review Home Page, http://www.aveva.com, 2010.

[4] Z. Qin, M. D. McCool, and C. S. Kaplan, "Real-time texture-mapped vector glyphs," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ser. I3D '06, 2006, pp. 125–132.

[5] S. Lefebvre, S. Hornus, and F. Neyret, "Texture sprites: Texture elements splatted on surfaces," ACM SIGGRAPH, April 2005.

[6] G. Cipriano and M. Gleicher, "Text scaffolds for effective surface labeling," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1675–1682, Nov. 2008.

[7] S. Barret, "Sparse virtual textures," http://silverspaceship.com/src/svt, 2008.

[8] M. Mittring, "Advanced virtual texture topics," in *ACM SIGGRAPH 2008 Games*, ser. SIGGRAPH '08, 2008, pp. 23–51.

[9] OpenGL 2.1 Reference Pages, http://www.opengl.org/sdk/docs/man2/, 2006.

[10] The OpenGL Shading Language 1.2, http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf, 2006.