Depth Maps and Other Techniques of Stereoscopy in 3D Scene Visualization

Cs. Szabó, B. Sobota and Š. Sinčák
Dept. of Computers and Informatics, FEEI, Technical University of Košice, Košice, Slovakia
{Csaba.Szabo,Branislav.Sobota}@tuke.sk, Stefan.Sincak@student.tuke.sk

***Abstract*—Visualization of three-dimensional environments is a very complex task when one aims to achieve high fidelity. There is a plenty of hardware supporting two-dimensional visualization presenting height and width of the objects of the 3D scene. The illusion of the depth can be multiplied by using some of the known stereoscopic projection techniques. This paper presents and compares three of them: colored anaglyphs, horizontally separated projection and 2D-depth maps. Sometimes, the static environments are not enough – that is the place where the object animation comes in. This dynamics is discussed in the last part of the paper before conclusion.**

I. INTRODUCTION

Visualizing the three-dimensional virtual scene can be achieved in several ways. The simplest one is just visualizing the static objects. This method can hardly bring the right feel of being in the virtual environment and also is not very usable.

To achieve a feel of being in the virtual scene, it is very important for objects on the scene to move. They can be animated (for example, leafs on the trees) or moving (cars etc.). This, combined with the one of the mentioned stereoscopic projections, can bring the great experience for the viewer [1, 2]. Important part of the visualization core application is the editing. The user must be able to create not just prepared scenes, but his owns. This can be done by adjusting the existing scenes or by creating new scenes completely. In many visualization cores the editing part is implemented as a real time one inside the application itself. User can switch between the viewing and editing modes without the need of restarting the application.

Complex tasks, such as animating objects, can be done by using a scripting language. The advantage of usage of a scripting languages lies in the fact, that the user can modify the dynamic parts of the virtual scene without recompiling the application. He also does not have to have decent programming skills in some of the mainstream programming languages (such as Java, C#, C++ and so on).

In fact, the user does not have to be a programmer. Even a graphic artist or just a person who works with the computer software in some way can learn the basics of some scripting language. Incorporating all of these features into one single application can be very helpful for the users. They can be able to view, create and modify the virtual scenes. Moreover, they can turn on the stereoscopic projection and feel the depth of the created virtual scene.

To achieve the depth of the virtual scene, stereoscopic projection can be used. The stereoscopy technique is known for tens of years. It works on a simple principle - delivering different images to each eye of the viewer. In the computer graphic science, there are several methods that can be used to implement stereoscopic projection. Displaying the stereoscopic image on the output device (such as monitor, television etc.) can be done by using several techniques.

The anaglyph image is the simplest one. This technique combines two different images into single one. The first of these images has cyan tint, while the second has red. The user wears a simple anaglyph glass, which consists of red and cyan lenses.

Second method to deliver different images to the users' eyes is done by rendering two different images. These images are then sent to some of the specialized devices, such as three-dimensional projector. The two input images are combined and displayed on the screen. Usually the user needs polarized glasses to feel the depth of the rendered scene.

One of the newest methods to deliver the feel of the depth is using the depth map with the specialized monitor. Two images are prepared in this case - the standard virtual scene with colors, textures and so on, and the scene depth map. These two are sent as the input for the specialized monitor, which displays the virtual scene. The advantage of this method lies in the fact, that the user does not need any specialized glasses.

The organization of the paper is as follows. Section II describes the scene visualization problem and offers a solution for optimal object rendering. Sections III-IV deal with the different kinds of projections for achieving the aimed realistic feel. Section V is devoted to the dynamics of the virtual reality scene. Section VI ends the paper by a conclusion.

II. SCENE VISUALIZATION

The virtual scene can consist of one or more virtual objects. When there is only one virtual object on the scene, it is very easy to render it. The problem comes when there are more objects to be rendered. Modern visualization cores are able to render thousands of objects in a single frame and so this work needs to be able. The important part of the visualization core is the scene graph. This graph consists of scene nodes. Each of these nodes has different meaning - one can be used as a geometry (geom), transformation (T), material node and so on. Having the virtual scene in the graph allows us to use some simple optimization techniques, which help us to have constant frame rate [3, 4, 5]. Example scene graph is

978-1-4577-1974-5/11/$26.00 ©2011 IEEE

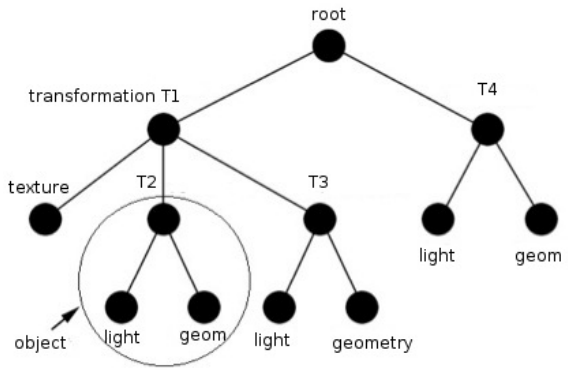

Figure 1. Example scene graph.

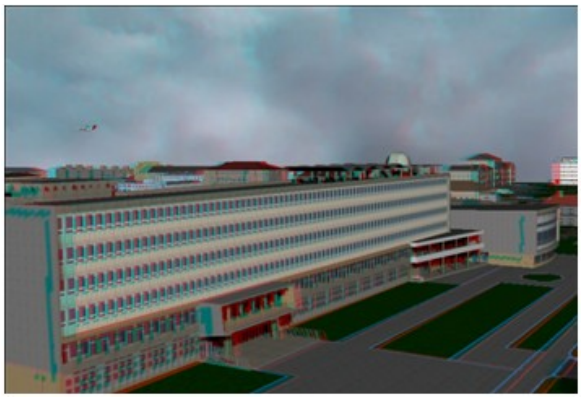

Figure 2. Scene visualized using anaglyph.

shown on Fig. 1. The main function of these optimization techniques is not rendering the invisible virtual objects.

In this work, the OpenSceneGraph [1, 2] is used. This library is able to work with the scene graph and render the objects in it. Several features need to be implemented to have the fully working visualizing subsystem. One of them is the management of the virtual objects in the scene. There needs to be an array of them, from which one can select the desired object. To know, which object to select one can use a unique identification number. All other subsystems can access the desired object just by calling the proper getter method of the visualizing subsystem and setting the object identification number as parameter.

Next part that has to be implemented is the scene saving and loading. It is not very user-friendly, when the scene needs to be built every time the application starts. The created scene should be saved to disk. When the application starts the next time, user can load the scene again. Several methods can be used to save the virtual scene to disk. The first and simplest one is saving the scene as one big three dimensional graphic model directly from some editing software (Google SketchUp, Autodesk Maya etc.). The second method is to use some internal binary structure to hold the scene information data. Loading and saving the scene in our own binary format is probably the fastest method. The text formats, like XML can be used too.

In this work, the text format has been chosen to save the scene data. To be more precise, it is not only the text format of our own. The scripting language has been used to hold the virtual scene data. With this method, the user can not only save and load the scene from the disk. He can also edit the saved scene and insert some dynamic parts into the scene. These include animating object movement, animating camera movement, playing sounds, music, setting the skybox and others.

III. Visualization using anaglyphs or separated images

To increase the feel of depth from the virtual scene, several stereoscopic projections can be implemented. The first of them is anaglyph projection (see Fig. 2). It combines two different views into one image [5,6,7]. Each of these views has cyan or red tint. User must wear anaglyph glasses to be able to see the effect.

The OpenSceneGraph library has direct support for visualizing the scene using anaglyph projection [1, 2]. However, several features need to be implemented. For example, adjusting the virtual distance between the rendering cameras (simulating the distance between viewer's eyes).

The second method is to render the two different images separately. These two are combined to the one image but they are not overlapping as with the anaglyph mode, nor have any tint. The final image is divided into two parts - left and right. Each of them has the same size. An example is presented on Fig. 3: the image for the left eye is located on the left side of the final image, and the image for the right eye is located on the right side of the image.

As with the anaglyph projection, the OpenSceneGraph library supports this type of transformation natively [1, 2]. The same features have to be implemented into the visualizing subsystem as with the anaglyph projection.

The hardest part comes with implementing the projection with using the virtual scene depth map. The OpenSceneGraph library not only does not support this mode, but it does not directly support rendering the scene depth map [1, 2]. Both of these features have to be implemented.

IV. Visualization using depth map

Some modern monitors are able to create the feel of the depth without forcing the viewer to use any specialized hardware (such as head mounted displays, anaglyph or polarized glasses etc.). As the input, they use the standard visualized scene (e.g. image) and its depth map (see Fig. 4).

Several things need to be implemented to the work. First one is visualizing the virtual scene depth map. There

Figure 3. Horizontally separated projections.

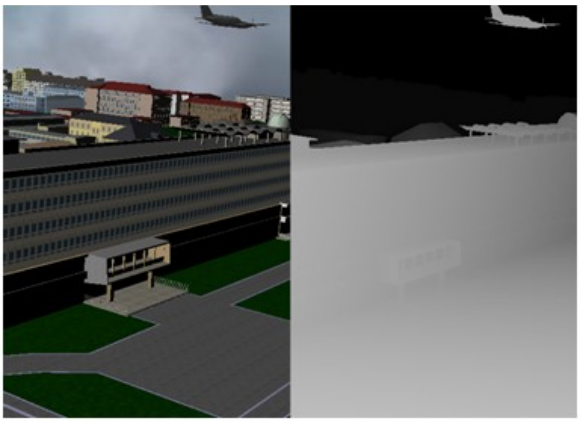

Figure 4. Projection using base image and depth map.

are some techniques we can use to achieve the scene depth map. They include using the OpenGL render to texture feature, the programmable shaders or using the fog effect. Each of these has its advantages and disadvantages.

With the OpenGL render to texture feature, we do not have any control of the output depth map. We cannot change the parameters of the depth map.

When using programmable shaders, modern graphic accelerators have to be used. This method is also technically harder to implement.

In this work, the fog method was used. The fog effect is one of the simplest effects that can be used on the graphic cards. It is supported even by the OpenGL or DirectX drivers and works on almost every modern graphics card. Its advantages are in the simplicity of its implementation, the speed of the effect and in the fact that the parameters of fog can be adjusted. The fog method alone is not very usable.

It uses the scene color data and is usually denser far from the camera.

To get the depth map using the fog technique, everything except the fog and geometry has to be disabled before rendering pass. All materials, lights or other features (skybox etc.) have to be disabled. The scene needs to be rendered only with the geometry and fog. Framebuffer clear color has to be set to black.

When all of that is disabled and the scene is rendered only with fog, we can see that the fog gets denser far from the camera. The area near the camera is black. To create the proper depth map, we have to invert the colors of this rendered image. We can do this by two ways - using some technique to manually invert each pixel of the rendered depth map or using the simple trick. The trick lies in the fact that we can set the fog start at the far distance, and the depth end in the near camera distance (see Fig. 5). Using this, the fog will get more dense near the camera and less far from the camera.

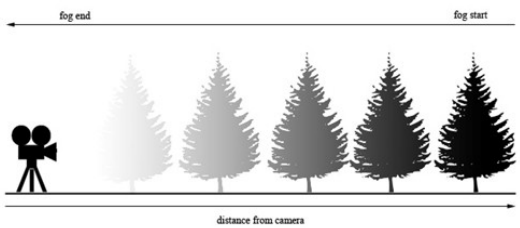

Figure 5. Using the fog effect to render the depth.

V. IMPLEMENTING THE DYNAMICS INTO THE SCENE

To increase the feel of being in the virtual scene, there need to be some moving objects. There can be cars moving around, or airplanes flying above the terrain and many others types of objects. When the user sees that the objects are moving, has some of the stereoscopic projections enabled and listens to the sounds that are in the scene, he can have the great experience using the application.

In this work, the first step to be done is implementing the Python language interpreter. When this part is done, our own scripts can be run. There need to be some calls from the script to our application code. An example of these is loading models, playing sounds, defining the animation paths and so on.

In the scripting subsystem, our own scripting module needs to be implemented. This module can then be directly called from the script and it joins the script code with our application code. When the script calls some script function, it is decoded by the interpreter and calls the native C++ function in our application. When the scripting subsystem is fully implemented, the problem how the objects will be moved on the screen must be solved. There are two ways of doing it - in the script by updating the object position each frame, or indirectly by defining and assigning the animation path to the object.

The first method of animating objects is done automatically when the scripting subsystem is fully implemented. One has the script, which he/she can run, so in this script one can make his/her own update functions that will be called each frame. In them, one can move the objects manually.

The better method that needs to be done is using the animation paths. These paths are the curves that are defined by control points. The point on these curves is computed by some known algorithm, such as Bezier, Catmull-Rom etc.

In the script, inserting the control points manually into it creates the animation path. After the last control point is inserted, the animation path can be assigned to some object in the virtual scene. The object that has the animation path assigned will move on it automatically.

During animation implementation, another problem occurs. This problem is related to the orientation of the object moving on the animation path. For preserving the orientation, or allowing its change respectively, additional calculations are needed.

The implemented system uses three angles called *yaw*, *pitch* and *roll* for rotation representation. Values of these angles are used to set the proper values for the rotation quaternion *rot=[x y z w]* used in the visualization algorithm in the core of the implementation (the interconnection to the library used in it). Value calculation equations follow.

$$
\begin{aligned}
c_1 &= \cos yaw \\
s_1 &= \sin yaw \\
c_2 &= \cos pitch \\
s_2 &= \sin pitch \\
c_3 &= \cos roll \\
s_3 &= \sin roll
\end{aligned} \tag{1}
$$

The values from (1) are used to calculate the quaternion elements *w, x, y,* and *z* as follows:

$$w = \frac{\sqrt{1 + c_1c_2 + c_1c_3 + c_2c_3 - s_1s_2s_3}}{2} \tag{2}$$

$$x = \frac{c_2s_3 + c_1s_3 + s_1s_2c_3}{4w} \tag{3}$$

$$y = \frac{s_1c_2 + s_1c_3 + c_1s_2s_3}{4w} \tag{4}$$

$$z = \frac{-s_1s_3 + c_1s_2c_3 + s_2}{4w} \tag{5}$$

Finally, the *rot* quaternion values are placed in each control point of the animation curve to preserve the orientation of the moving object. A little speed enhancement is achieved by the modification of the equations presented above by calculating the value *4w* present in (3)-(5) also separately in advance.

The final implementation of the calculations is as follows:

```
1.  c1 = cos( yaw );
2.  s1 = sin( yaw );
3.  c2 = cos( pitch );
4.  s2 = sin( pitch );
5.  c3 = cos( roll );
6.  s3 = sin( roll );
7.  w = sqrt(1.0+ c1*c2+ c1*c3- s1*s2*s3+ c2*c3)/2.0;
8.  w4 = (4.0 * w);
9.  x = (c2 * s3 + c1 * s3 + s1 * s2 * c3) / w4 ;
10. y = (s1 * c2 + s1 * c3 + c1 * s2 * s3) / w4 ;
11. z = (-s1 * s3 + c1 * s2 * c3 +s2) / w4 ;
```

VI. Conclusions

In this work, the three-dimensional visualization core has been implemented. There was shown how to implement some stereoscopic projections using existing components, such as anaglyphic, horizontally divided and the one with the depth map.

All of these, when used with the proper device, can bring the great user experience during viewing the virtual scene. The dynamic parts of the scene can be implemented using the scripting language with defining animation paths in the script file.

However, there are many features that can be implemented later. One of these could be editing the dynamic parts of the virtual scene in the real time.

Next, the more graphical effects can be implemented, including the full scene shadow mapping, per-pixel lighting, bloom rendering etc.

Acknowledgment

This work was supported by VEGA grant project No. 1/0646/09: “Tasks solution for large graphical data processing in the environment of parallel, distributed and network computer systems.”

References

[1] P. Martz, “OpenSceneGraph Quick Start Guide.” In: Skew Matrix Software, 2007.

[2] B. Kuehne and P. Martz, “OpenSceneGraph Reference Manual v.2.2.” In: Skew Matrix Software, 2007.

[3] M. Poth and T. Szakall, “Spatial and frequency domain comparison of interpolation techniques in digital image processing,” in *Proc. of CINTI 2009*, Budapest, Hungary, Nov 12-14, 2009.

[4] B. Koljic, J. Simon, and T. Szakall, “Determining distance and shape of an object by 2D image edge detection and distance measuring sensors.” In *Proc. of SISY 2008*, Subotica, Serbia, 2008.

[5] Cs. Szabó, B. Sobota, and R. Kis, “Terrain LoD in real-time 3D map visualization.” In *Abstracts of the 8th Joint Conf. on Math. And Comp. Sci.*, Komarno, Slovakia, Jul 14-17, 2010.

[6] Cs. Szabó, S. Korecko, and B. Sobota, “Processing 3D scanner data for virtual reality.” In *Proc. of the 10th Intern. Conf. on Intel. Syst. Design and Application*, Cairo, Egypt, Nov 29-Dec 1, 2010.

[7] B. Sobota, M. Rovnak, and Cs. Szabó, “3D scanner data processing.” In *J. of Information, Control and Management Syst.*, vol. 7, no. 2, 2009.