A Depth of Field Algorithm for Realtime 3D Graphics in OpenGL

Ola Henriksson

2002-01-31
A Depth of Field Algorithm for Realtime 3D Graphics in OpenGL

Examensarbete utfört i Medieteknik vid Linköpings Tekniska Högskola, Campus Norrköping

Ola Henriksson

Handledare: Per Nordqvist
Examinator: Björn Gudmundsson

Norrköping den 2002-01-31
Title

A Depth of Field Algorithm for Realtime 3D Graphics in OpenGL
Algoritm i OpenGL för att rendera realtids 3D grafik med fokus

Author

Ola Henriksson

Abstract

The company where this thesis was formulated constructs VR applications for the medical environment. The hardware used is ordinary desktops with consumer level graphics cards and haptic devices. In medicine some operations require microscopes or cameras. In order to simulate these in a virtual reality environment for educational purposes, the effect of depth of field or focus have to be considered.

A working algorithm that generates this optical occurrence in realtime, stereo rendered computer graphics is presented in this thesis.

The algorithm is implemented in OpenGL and C++ to later be combined with a VR application simulating eye-surgery which is built with OpenGL Optimizer.

Several different approaches are described in this report. The call for realtime stereo rendering (~60 fps) means taking advantage of the graphics hardware to a great extent. In OpenGL this means using the extensions to a specific graphic chip for better performance, in this case the algorithm is implemented for a GeForce3 card.

To increase the speed of the algorithm much of the workload is moved from the CPU to the GPU (Graphics Processing Unit). By re-defining parts of the ordinary OpenGL pipeline via vertex programs, a distance-from-focus map can be stored in the alpha channel of the final image with little time loss.

This can effectively be used to blend a previously blurred version of the scene with a normal render. Different techniques to quickly blur a rendered image is discussed, to keep the speed up solutions that require moving data from the graphics card is not an option.

Keyword

Depth of field, OpenGL, realtime, stereo, render, computer graphics, vertex program.
Abstract

The company where this thesis was formulated constructs VR applications for the medical environment. The hardware used is ordinary desktops with consumer level graphics cards and haptic devices. In medicine some operations require microscopes or cameras. In order to simulate these in a virtual reality environment for educational purposes, the effect of depth of field or focus have to be considered. A working algorithm that generates this optical phenomenon in realtime, stereo rendered computer graphics is presented in this thesis. The algorithm is implemented in OpenGL and C++ to later be combined with a VR application simulating eye-surgery which is built with OpenGL Optimizer.

Several different approaches are described in this report. The call for real-time stereo rendering (~60 fps) means taking advantage of the graphics hardware to a great extent. In OpenGL this means using the extensions to a specific graphic chip for better performance, in this case the algorithm is implemented for a GeForce3 card. To increase the speed of the algorithm much of the workload is moved from the CPU to the GPU (Graphics Processing Unit). By re-defining parts of the ordinary OpenGL pipeline via vertex programs, a distance-from-focus map can be stored in the alpha channel of the final image with little time loss. This can effectively be used to blend a previously blurred version of the scene with a normal render. Different techniques to quickly blur a rendered image are discussed, to keep the speed up, solutions that require moving data from the graphics card are not an option.
Chapter 1

Introduction

With the development of better and faster computers equipped with the latest in computer graphics technology, we are all used to see stunning three-dimensional computer graphics rendered in real-time. Whether it is a recreational computer game or a VR-application teaching people how to fly airplanes, the computer generated graphics seen on the display device just keep getting more realistic with every year. This realism is achieved with improvements in texturing, higher polygon counts, more accurate lighting etc. all resulting in clear, crisp images being presented to us.

But what if we don’t want that?

In real life (unfortunately) the things we see are not always crisp and clear but rather blurred and bleak.

Depth of field (DoF) is one such instance when the ordinary way of rendering graphics does not suffice. It is an integral part of human vision as well as in “real life” cameras and microscopes where the lens (or system of lenses) determines what parts of the surroundings are in-focus and which are not. For added realism in an artificial scene, the effect of DoF should be considered and only the objects in-focus be rendered as exactly as possible, objects out of focus should be blurred.

Simulating the DoF of human vision in a virtual-reality environment differs from simulation of more static lens-systems such as cameras though. To achieve the first, one would need equipment to track the eye movements of the user to determine where his center of attention is, whereas a camera or microscope is a part of the synthetic scene and all necessary information is known by the application.
This report will only present a method to simulate DoF of a lens system that is part of the artificial scene and how to achieve this in real-time.

The proposed algorithm will be used in a medical VR-application being developed by the company where this thesis was formulated. The final application will be a simulation of an eye operation to remove cataract, where surgeons can practice the complicated procedure on the computer instead of in wet-labs using animal parts. The actual operation includes removing the affected lens using an ultrasonic device, the phaco-emulsificator, to break it apart and extract it, followed by insertion of a synthetic lens. During all this, the surgeon uses a microscope to enhance his view of the small details in the eye, with a foot pedal controlling the zoom and focus of the microscope.

Since it is important for the physician to always be able to see the work area clearly while performing a real procedure, it is imperative that the focus function is included in the training simulator. Not only for visual realism but to train the method of working as well.

There are of course several other uses for an algorithm that simulates focus except for the one described above. In the area of medical VR applications it would also be needed when simulating endoscopy, procedures that involve inserting a camera into the patient's body. Another field of application is in computer games where it could be effectively used in replays or in-between sequences. In cinematography, depth of field is often used to hint to the viewer where he should focus his attention, a simple trick that could be taken advantage of in games as well, apart from the added realism.

This thesis requires that the reader is well-read in three-dimensional computer graphics and familiar with at least the basics of OpenGL. The theory is not very difficult to grasp but the implementation and terminology might be tougher to understand.

The OpenGL programming guide [1] is a very good introduction to OpenGL and can be recommended.
Chapter 2

Problem Analysis

This chapter describes the prerequisites of this project and the problem to be solved. The reader is presented with a brief overview of the possible solutions and what they entail.

The depth of field algorithm is first and foremost to be used in the PHACO application but should, preferably, be general enough to be utilized in other applications as well. The company uses OpenGL Optimizer for their virtual reality environments, a freeware API (Active Programming Interface) developed by Silicon Graphics. Its core is OpenGL (surprise) but with functions to build scene graphs, collision detection and much more. Because of this, the algorithm must be designed using OpenGL and C++ to later be merged with the OpenGL Optimizer code for the simulator.

The final application will display the graphics in stereo mode on two small, separate screens taken from a VR-helmet developed by Ericsson. Each screen will be inserted into a modified microscope and has a resolution of 1280x1024, the 3D-graphics will be rendered in a window of size 1024 x 1024.

The 3D-model of the eye will probably be constructed out of around 15,000 triangles (give or take a couple of thousands).

It has to be rendered in real-time in stereo mode and at least be able to achieve frame rates averaging around 60 frames per second (fps), 30 fps for each screen.
Chapter 2 - Problem Analysis

The hardware the simulator will be running on is an ordinary desktop, albeit one equipped with a top of the line consumer graphics card. Exactly which model and from which vendor the graphics card will be depends on the computational demands of the final product. Most likely it will be of the Geforce-family, but because of the rapid development in graphics hardware, that might change.

The purpose of this master thesis is to implement and evaluate different methods to achieve a depth-of-field effect using OpenGL. The implementation part of the thesis is to code small sample programs using different techniques for DoF. This includes implementing already existing methods and inventing new ones. The evaluation part of the thesis is simply to determine if it is possible to use the different algorithms in a real-time environment based on the speed and appearance of the 3D graphics.

2.1 Methods

There are several different ways one can approach the problem of simulating depth-of-field. The physically correct method would be to simulate the interaction of incoming light from the scene with a synthetic lens system and achieve a very realistic depth of field. The call for real-time rendering rules out any possibility of simulating real DoF though, it is simply too computationally demanding so the alternative is to find a way to fake it as fast and realistic as possible.

This alternative is common in computer graphics; if the result looks good we can use it, how it is achieved is of less concern. What is needed is a fast method to find and blur the parts of a scene that are out of focus, while keeping the rest intact.

To actually get a framerate of 60 fps the hardware capabilities must be fully taken advantage of to speed up the process.

One method of rendering depth of field is described in the book “OpenGL Programming Guide” [1]. This method includes rendering the scene several times off-screen from slightly different camera locations and blending these images together before finally presenting it to the screen.

The other methods have a common methodology, namely rendering the scene off-screen and converting this image into a texture. The texture can then be blurred using different techniques before it is applied to a plane orthogonal to the view vector and with size equal to the display screen. The blurred image covering the entire scene can then be blended with the original (in-focus rendered) scene in a number of ways. With this approach the problem is divided into two separate issues. First of all, a fast and efficient method to blur a rendered texture needs to be found. Next, the blurred version of the scene must be blended with the original in such a way that only the out-of-focus parts of the final image are blurred.
2.1.1 Blur
The first method that comes to mind regarding blurring an image is probably convolution with an averaging kernel. The result is a very smooth blur without artifacts but the drawback is that like most image processing algorithms, it is not very fast.

A very fast method on the other hand is rendering a low-resolution image and using this as a texture for the screen-covering quad. This approach takes advantage of the linear interpolation capabilities of the graphics card when a small texture is magnified to cover a large surface.

A similar approach takes advantage of the hardware’s interpolation twice, first by rendering a full resolution image and converting it to mipmaps and then choosing one of the downsized images to texture the quad.

Using offset to generate a blurred image is also a possibility. If the texture is applied several times but shifted a pixel (or a few) in different directions before each application and then finally blended equally one gets a blurred image as a result. (Shift it enough times and you have mimicked the convolution approach, but the speed wouldn’t have improved much either)

2.1.2 Blend
The difficulties involved in blending the two versions of the scene is first determining what part are actually out of focus and then applying the appropriate amount of blur. Since depth of field is continuous, the sharp, in-focus rendered scene should preferably gradually change into the blurred image when the distance from the focal-plane increases.

A simple solution is to render several planes parallel to the screen but with increasing distance from the viewer and applying textures with different amounts of blur. However, to avoid noticeable transitions between these different levels of blur quite a number of planes are necessary.

A better solution would be to calculate a transparency value directly related to the distance from focus for each vertex in the model. This can actually be done in hardware on new graphics cards with a great result and almost no time penalty.
Chapter 3

Implementation and Evaluation

This chapter describes the different approaches to the problem that were made during the project. The reader will be presented with the different algorithms that were implemented during the project and their respective pros and cons. The description will be more detailed than in the previous chapter and some short pieces of pseudo-code to further explain.

The work process of this project involved a lot of switching back and forth between actually implementing code and researching about possible ways to improve the algorithms. It was basically not possible to exactly plan the workflow of the project and divide the time between research and implementation.

This is because standard OpenGL is not powerful enough for a working depth of field algorithm. The capabilities of the specific graphics hardware must be fully exploited and that can only be done by taking advantage of the graphics card’s private extensions of OpenGL, which often are vendor specific.

Specifications of these extensions are found on the manufacturer’s web-site, often together with more extensive documentation and/or source code.

At the start of the project the hardware consisted of an ordinary desktop with a Pentium III processor, 512 Mb RAM equipped with a GeForce2 graphics card. The computer stayed the same throughout the entire project with the exception of a replacement of the graphics card to the successor GeForce3. The newer card has some specific properties which greatly improved the final result, more about this later.
3.1 Using the framebuffer

3.1.1 Frustum
The first method to render DoF was taken directly from the book “OpenGL programming guide”. It was apparent beforehand that it was not an alternative but it is always good to have a reference. The algorithm is described very well in the book and the interested can read the details there. As already stated, it involves rendering the entire scene several times from slightly different viewpoints (Figure 1). The rendering target is not the ordinary back-buffer but rather the accumulation-buffer where the average of all renderings is calculated. For a good result at least eight renderings are needed. The result is visually very good, at least if the objects in the scene are of similar size and located in roughly the same space close to focus. If not, the final image can look pretty weird when, instead of a blurred object, there are 8 separate, rather transparent copies of the original.

![Figure 1](image)

The viewpoint is shifted for each render while the plane of focus is kept stationary.

The framerate suffers greatly in this approach to depth of field, it is well below 1 fps. There are a number of reasons for this, first is the multiple renderings which are time consuming. But even with a rather low polycount it is still slow and that is due to the accumulation-buffer. It is most likely not implemented in hardware and should never be used in a real-time application.

This approach to depth of field can be implemented far more efficiently than the book’s description on a GeForce3 card though. By taking advantage of the NV_vertex_program extension (see Appendix A) the algorithm is changed in the following way. Instead of rendering to the accumulation buffer N times, a vertex program can be used to render the scene N times directly to the frame-buffer with blending enabled. Besides the ordinary transform- and lighting-calculations each vertex’ alpha-value is multiplied...
with 1/N. This way, the accumulation-buffer is not utilized. The speed is now only dependent on the complexity of the scene and for the test scene it arrives at around 40 fps.

There is a drawback with this approach though. Since we work with a limited color resolution, blending with a factor of 1/N for each rendering pass reduces the precision of the color-value in each fragment. The result of this is that the final image looks like it has a lower color resolution than it actually has.

### 3.1.2 Convolution

This method, like the former, does not take full advantage of the hardware capabilities either. The OpenGL standard 1.2.1 includes the command `glConvolutionFilter2D(…)` but it is to my knowledge not fully supported in any consumer graphics hardware, only a software implementation of the OpenGL command is used.

This could change in the future with the next generation of graphics cards but I deem it unlikely. Since the OpenGL command is not hardware accelerated anyway (and did not fully cooperate with the drivers that were used at the time) convolution was implemented in the following way.

1. Render the entire scene.
2. Read pixels from the framebuffer to processor memory
3. Apply convolution filter
4. Draw pixels back to framebuffer

The convolution algorithm was first implemented in such a way as to generate horizontal blur (motion-blur). Using a horizontal kernel of size 1xN with each variable having a value of 1/N and letting this kernel traverse the image row by row does this. This approach can be speeded up by the fact that only two of the values in the kernel changes with each pixel. All that needs to be done is calculate the entire kernel once and for each new pixel subtract the leftmost value of the kernel and add a new in the rightmost space (Figure 2). This way an increase in blur (an increase of N) does not affect performance.

![Figure 2.](image)

When the filter kernel traverses the image only two values change with each step.
After evaluation of this method to solve the blur part of the DoF problem there was no need to implement an algorithm using a NxN kernel (normal blur). It could easily have been done by repeating the process but with a vertical kernel and a traversal of the image column by column. This was not necessary since the first, faster approach was still far too slow to be used in a real-time simulator. It is unaffected by the size of the kernel but not the size of the image. Just like the frustum algorithm, the framerate was under 1 fps and thus not even in consideration. It is not only the actual convolution that is time-consuming though. Reading the entire image from the graphics card into processor memory and back is costly. Commands such as `glReadPixels(...)` does just that and should be avoided.

### 3.2 Blurred textures

The following algorithms have a common methodology, which will be described here. It consists mainly of these steps which are executed each frame:

1. Render the entire scene.
2. Convert the rendered image to a texture.
3. If necessary, render the scene again in full resolution.
4. Apply the texture from (2) on a quad orthogonal to the view-vector and covering the entire screen. Blur the texture.
5. Blend the textured quad with the scene already in the framebuffer.

There are some limitations in the graphics hardware used that make all the steps necessary in most approaches. Both GeForce2 and GeForce3 have a restriction on the size of regular textures, they can not be greater than 512x512 pixels in 32-bits RGBA. This is not entirely true though, there is an extension available (`NV_texture_rectangle`) to both these cards that makes it possible to have larger textures but it contains other limitations such as no automatic mipmap generation.

Because of this limitation, the first render may be in a resolution of 512x512 (or less). Either to an off-screen rendering buffer (a pixelbuffer) or the ordinary framebuffer but with the viewport changed to fit the lower resolution. That is why a second, full resolution, rendering of the scene might be necessary after the result of the first render has been converted to a texture.

There is not a very significant difference in efficiency between using a pixel-buffer and the ordinary framebuffer, pixel-buffer increases performance with a few percent in the test environment. But according to both nVidia (the manufacturer of the GeForce chip) and ATI (Radeon), the use of a pixel-buffer will be faster for render-to-texture algorithms in future drivers.

The differences between the different depth of field algorithms are how the
texture is used to achieve a blurred copy of the scene and how the final blending is achieved.

### 3.2.1 Low-resolution blur

Unlike the previous attempts this method takes advantage of the hardware capabilities to a greater extent. The general idea is to use the linear interpolation of the hardware when applying a small texture to a larger object, to give the appearance of a larger, blurred, texture. The first render in a new frame is consequently made in a low resolution, for instance 128x128 pixels. This image is then copied into texture-memory with the standard OpenGL command `glCopyTexImage(...)` which is actually faster than `glCopyTexSubImage(...)` even though we copy the entire buffer. Unlike the `glReadPixel(...)` and `glDrawPixel(...)` commands the former do not transfer the data via the processor, which is what we want since both the target and the source is on the graphics card.

This texture is applied on a quad covering the entire window, which in effect gives it the size 1024x1024 pixels. To fit the smaller texture onto the surface, the hardware interpolates the intermediate values generating a blurred version of the scene. This approach to blur an image is very fast, over 100 fps on the test scene, but unfortunately not as visually appealing. The blur is not very smooth and especially edges of objects are clearly jagged and angular. There is also a very unpleasant effect when moving objects while keeping the focal-plane in the same position, parts of the texture seem to be alive and creeping.

### 3.2.2 Mipmap textures

Very similar approach to the previous but instead of just one low-resolution rendering, the hardware is used to interpolate several, with the help of automatic mipmap generation. The base texture is supplied in a similar fashion as the previous method, but with a size of 512x512 (the maximum size allowed for an ordinary texture). The graphics hardware automatically interpolates mipmaps from this base texture if the `SGIS_generate_mipmap` extension is used. This extension only works with ordinary textures though, which is why only a texture of size 512x512 can be used. (A 1024x1024 texture would improve the final result, if it could be used.)

![Figure 3. Mipmaps of the original image, each new level of the array has half the height and width as the previous. The last mipmap has been magnified to the original size.](image-url)
Every time the base level of the mipmap-array is updated new mipmaps are generated by this extension, which is done once each frame.

Which mipmap to use when textures are applied to surfaces is normally decided by the graphics card based on how small the surface is in the current camera-view. This decision can be overridden by the EXT_texture_lod_bias extension which gives the programmer control of which mipmap to texture an object with. Instead of linearly magnifying the texture to fit the screen-covering quad, which is the normal course of action, a smaller mipmap is provided via this extension which is then magnified to fit the surface.

The quality of the blur generated with this method is better than the low-resolution approach, less visible artifacts and not quite as jagged and angular edges. The creeping of the texture when moving objects is still apparent.

To automatically generate mipmaps each frame is a little bit more time-consuming than only generating one in a low resolution. This combined with the extra time needed to render the scene in 512x512 pixels instead of only 128x128 makes this method somewhat slower but it is still surprisingly fast. The test scene still has an average framerate of 90 fps making this approach far better than the former (only a low resolution texture).

3.2.3 Offset and blend

This method takes advantage of modern hardware’s capabilities for fast and efficient texturing. The methodology is similar to the previous but the texture from the first rendering is in full resolution, the blur is achieved by applying the same texture several times but with a slight offset with each pass. This approach can be implemented in several ways, the most basic approach is explained first.

First of all the ordinary texture object can not be used to hold the rendered image of the scene, a GL_TEXTURE_RECTANGLE_NV object has to be utilized because of the limit in size of GL_TEXTURE_2D. This texture is applied to several quads, each quad is in the same position covering the entire window. Blending must be enabled while the depth-test must be disabled. For each new quad the texture coordinates are slightly shifted in different directions. With the use of several extensions the steps above can be made more efficient by combining the separate stages into one.

Instead of rendering one quad with each offset texture, all textures are applied with different offsets on one, single quad. To do this, two extensions are used, ARB_multitexture and NV_register_combiners. The multi-texture extension has been included in the OpenGL standard 1.3, but still has to be initialized like all other extensions. It makes it possible to apply several textures on a single object and with greater efficiency than the alternative, two texture objects can be applied with each texturing pass.

To get a blurred result, the same texture is supplied to every multi-texture but the texture coordinates for each differ.

The register combiner extension can be used to define how these textures are to be blended with each other if the normal modes of blending do not suffice. A brief introduction to combiners is found in Appendix A.
For this application, averaging the separate texture inputs with the combiner is enough. The blur generated by this approach looks similar to the blur created by the method proposed in the “OpenGL Programming Guide”. There is only one level of blur though, whereas the book’s algorithm had continuous levels and also took the average of different view-directions. Different levels of blur can be achieved when blending the blurred image with the in-focus rendering of the scene but the result is somewhat lacking. The final result depends on the number of texture applications, which in turn affect performance.

The graphics card used also affects the methodology, for instance GeForce2 only supports two combiners at a time and two multi-texture objects, while GeForce3 supports eight combiners and four multi-textures.

### 3.3 Blending texture and scene

Combining the out-of-focus image with the in-focus rendering to yield a final result where the level of blur depend on the distance from focus can be done in two ways. One method can be implemented on most graphics hardware while the other requires a GeForce3 or Radeon 8500 card. The first method can be implemented using standard OpenGL alone, but the second needs extensions which only the two cards mentioned support (yet).

#### 3.3.1 Using standard OpenGL

The entire scene should be rendered and available in the framebuffer. The method determines what parts of the scene are in-focus by simply slicing through the view-frustum with two planes, orthogonal to the view-vector, positioned at equal distance on each side of the focal-plane (Figure 4).

The surface behind the focal-plane is rendered normally and textured with the blurred version of the scene effectively blurring everything behind the plane (Figure 5). The surface in front of the focal-plane is rendered with the
depth test set to GL_GREATER, this removes the parts of the scene which are closer to the camera and replaces it with the blurred texture on the plane. Those parts of the scene located in-between the two planes are still visible and in-focus.

If several versions of the scene with different levels of blur have been attained, the final result can be greatly improved if more slice-planes are used and textured with more and more blurred images as the distance from the focal-plane increases. If the planes are rendered back to front, the stencil buffer could be used to keep track of which fragments that have already received blur as we move towards the camera.

But even with several planes, there are often visible borders between differently blurred textures.

There is also the problem of intensity (color) leaking, which is difficult to avoid completely. The color leaks are manifested as blurred edges around objects that are completely in focus. This is because the blurred version of the scene that has been plastered on a plane behind the objects contains blurred versions of the in-focus objects too. The depth-test only removes fragments directly behind the objects, the surrounding fragments are still rendered.

To avoid this, the blurred texture must not contain parts of the scene that will be in focus. This can easily be taken care of in the render-to-texture part by changing the clipping-planes of the frustum. This solution is not perfect though, when choosing not to render all objects the texture might contain areas without any color information (except for the clear color). This in turn can affect the edges of blurred object, giving them too low color intensities.
3.3.2 Via vertex programs
By far the better alternative of the two, this method only works on graphics hardware that supports vertex programs. On a GeForce3, the extension needed is `NV_vertex_program`. Appendix A contains a more thorough description of this extension, if the reader is not familiar with vertex programs it is a recommended introduction.

For the depth-of-field application we need a fast and efficient way to calculate the distance between each vertex and the focal-plane. Using this new technology, these computations can be done very fast in the graphics hardware by overriding the ordinary transform and lighting part of the rendering pipeline.

Besides calculating transforms and lighting for each vertex, the vertex program used in the DoF-algorithms derives a value based on the distance between the vertex and the focal-plane. The value is stored in the alpha part of the vertex’ output color and passed on to the next stage in the OpenGL graphics pipeline. The final image looks exactly like a normal render but the content of the alpha-channel is now based on the distance to focus.

Exactly how this is done can be seen in the program code in Appendix B, but basically a region of complete focus is specified around the focal-plane, if the vertex lies within this region the output alpha is equal to the input alpha. If it lies outside of this region the distance between the vertex and the closest part of the region times a linear coefficient is subtracted from the input-alpha. The value is clamped to [0...1].

This way, the size of the focus-region and how fast the transition from focus to blur is, can be controlled with two floating-point variables.

The vertex program can be enabled during the render-to-texture stage or during the last render of the scene, the one in full resolution. Either way, each pixel in the framebuffer contains an alpha-value that is 1 if the pixel is in-focus and less if it is not.

![Diagram](image_url)

**Figure 6**
A plane textured with the blurred version of the scene is rendered with blend enabled, in front of the geometry. The alpha-value in each pixel determines if the texture or the geometry behind is seen.
With blending enabled and the blending function defined to use source alpha if the texture contains the modified alpha or destination alpha if the framebuffer does, the rendered texture is blurred and applied to a quad at the same position as and aligned to the near-clip plane (Figure 6). The parts of the scene that lie within the specified focus-region are completely visible through the texture, while only the texture is visible for parts that lie completely out of focus. In between the two regions, there is a continuous transition from the rendered scene to the blurred texture.

Like the previous blending method, this approach suffers from color leaking as well. When using a full resolution render to determine the transparency of the blurred texture the intensity leak is very apparent. If the vertex program is enabled during the render to texture stage and a slightly blurred version of the original focus-map is used during blending the effect is not as noticeable.

Because of the great rift between the two approaches in terms of performance and visual quality, the decision was made to use a graphics card that supports the vertex program extension and all further development was made for a Geforce3 graphics card.

The methodology of rendering a texture, blurring it and merging it with the rest of the scene that this report has focused on has a drawback previously unmentioned. How are transparent objects handled? The short answer is, they’re not. The problem is mainly the fact that while a transparent object might be within the focus-region, the object behind it might not and vice versa. With the vertex program we calculate a blur-value for each pixel in the output image (or rather, how much it will blend with the blurred image). The problem is of course that two separate objects are visible in a single pixel if at least one of the objects is transparent. But there is only one alpha-value for that pixel determining the amount of blur. The simple solution for this problem is to ignore it, choose to render all transparent objects after the textured quad has been rendered. This means that all transparent objects will be rendered in-focus, but the objects behind them will be blurred correctly according to their distance from focus. If the transparent objects do not have a significant importance to the scene such as intricate structures, detailed textures or are almost opaque, the fact that they are not blurred is mostly just visible at the edges. If this is not the case it is of course possible to render some of them in the same manner as the other objects, letting their position in the scene be the basis for the blur instead of the opaque object behind it.
This chapter describes the final result. A more in-depth look at the two best approaches to depth of field which, although similar in many ways, are implemented a bit differently.

As the project evolved and the different methods described above was implemented and evaluated it became apparent that they each had different drawbacks as well as different advantages. The kind of three-dimensional scene that was being displayed also influenced the final result, for instance the more complex scene of an eye during surgery with detailed textures was very forgiving to the angularity and jaggedness of mipmap-blurring. This is not the case for a scene containing homogenous areas with soft shading and sharp object boundaries where blurring using mipmaps produces an awful result. For this type of synthetic scene, blurring via offset-textures turned out to yield a far better result.

With this in mind, two algorithms have been implemented which are based on similar techniques but still have some different characteristics. Performance wise they both average at about 60 fps and actual implementation with the VR-application should determine the best alternative of the two.

Both algorithms take advantage of several extensions to OpenGL to achieve the necessary speed needed for stereo rendering. The first step in both algorithms is to initialize the necessary extensions. This is only done once before the main loop is called and the same extensions are used:
4.1 Offset on mipmaps

The first algorithm is a combination of mipmapping and offset, and uses a vertex program to calculate the blending. Instead of using a single magnified mipmap to blur the scene, the same mipmap (or rather level-of-detail) is bound to four different multi-textures. These textures each get a different offset and when they are applied to the screen-covering quad their intensities are averaged using a register combiner.

This way the artifacts in a mipmap are smoothed out and the drawback of only having four different offsets is less visible since the textures are already somewhat blurred. The creeping effect when using low-resolution textures that are magnified is also almost gone.

The main loop, drawing each frame, consists of these steps:

1. Switch to pixelbuffer (512x512)
2. Render scene with vertex program enabled
3. Copy to texture (mipmaps are generated automatically)
4. Switch back to ordinary rendering buffer
5. Render scene
6. Enable blending with source alpha
7. Disable the depth-buffer
8. Bind different mipmaps to different multitextures and enable register combiner
9. Render a single quad with the same dimensions as the near-clip plane
10. Disable the combiner and enable the depth-buffer.
11. Render transparent objects
12. Swap buffers

The main difference of this algorithm from previous attempts is found in step 8-9. Here the mipmap-texture is bound to the four multi-textures, each with the same level-of-detail (LOD). An appropriate offset-value is calculated and applied to each of the four textures in opposite directions (a leaning \( x \)).

The amount of blur depends on both the LOD-value and the offset-value. With register combiners enabled, the average of the bound textures’ rgb-values is calculated for each fragment. This part can be done with only two combiners, we need a total of six however. This is because the textures contain the focus-dependent alpha (step 2), which has to be combined in such a way as to minimize the effect of color leakage. Ideally, the object
in focus should not be blurred at all and thereby not leak any color to the surrounding pixels. Since this already has occurred, the output-alpha value for each fragment must not be as high as the average of the four input-alpha values.

This will result in a blurred halo around the object in focus. Instead, both the average and the product of the input values are calculated and the weighted average of the two are used as output.

The performance of this algorithm is very good, with the test scene and a transparent sphere (an addition of ~2500 polygons) it runs at an average of 65 frames per second.

### 4.2 Recursive offset

The second algorithm only uses offset but applies it recursively to yield a very smooth blur. Instead of applying offset at the very last part of the algorithm, a similar technique with textured quads is utilized already in the pixelbuffer. By repeating the process of texturing a quad (with offset) and generating a new texture, several, differently blurred textures are acquired.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Switch to pixelbuffer (512x512)</td>
</tr>
<tr>
<td>2</td>
<td>Render scene with vertex program enabled</td>
</tr>
<tr>
<td>3</td>
<td>Copy to texture</td>
</tr>
<tr>
<td>4</td>
<td>Disable the depth-test</td>
</tr>
<tr>
<td>5</td>
<td>Bind the previous texture to four multitextures and enable register combiners (set one)</td>
</tr>
<tr>
<td>6</td>
<td>Render a single screen-covering quad</td>
</tr>
<tr>
<td>7</td>
<td>Copy to texture</td>
</tr>
<tr>
<td>8</td>
<td>Repeat step 5-7 several times</td>
</tr>
<tr>
<td>9</td>
<td>Switch back to ordinary rendering buffer</td>
</tr>
<tr>
<td>10</td>
<td>Render scene</td>
</tr>
<tr>
<td>11</td>
<td>Enable blending with source alpha</td>
</tr>
<tr>
<td>12</td>
<td>Disable the depth-buffer</td>
</tr>
<tr>
<td>13</td>
<td>Bind the first and last texture to different multitextures and enable register combiner</td>
</tr>
<tr>
<td>14</td>
<td>Render a single screen-covering quad</td>
</tr>
<tr>
<td>15</td>
<td>Disable the combiner and enable the depth-buffer.</td>
</tr>
<tr>
<td>16</td>
<td>Render transparent objects</td>
</tr>
<tr>
<td>17</td>
<td>Swap buffers</td>
</tr>
</tbody>
</table>

The main loop, drawing each frame, consists of these steps:

In step four the depth-test is disabled removing the need to clear the buffers between each quad. The texture is then applied with an offset in the shape of an \( x \) and the average of the four textures, both the rgb- and alpha-part, is calculated with the first set of combiners.
Unlike the previous approach, we need two different sets of combiners. The first is used in the pixelbuffer and its task is simply to blur the four inputs equally. The resulting image is copied into a new texture-object and the process is repeated but with this new texture (Figure 6). For a good result the shape of the four offsets should be alternated, the second time around it has the form of a +.

![Figure 6](image)

The image at the top is the original texture generated from the content of the pixelbuffer. This is followed by the first offset application (middle) in the shape of an x and a second in the shape of a + (bottom).

One of the advantages with this approach is the scalability of the actual blurring, the more times the previous steps are repeated the smoother blur is achieved.

To blend the blurred texture with the in-focus part of the scene another alternative method is introduced. This is where the second combiner set comes into play. As before, it is important to minimize the effect of color-intensity leaking from in-focus objects to their surroundings.

The theory used in this method is that if an object is in focus its alpha-value in each pixel is zero and the pixels surrounding it has a value of one. When this image is blurred in the pbuffer, the alpha values are blurred as well and that gives a hint of how much leakage has occurred (Figure 7).

In the final combiner the alpha value of the blurred texture is used to interpolate between the color of the blurred texture and the texture without blur.
This method with two recursions (three textures in total) performs as the previous with a slightly smoother blur, around 63 fps. With three recursions however the framerate drops to around 54 but the visual quality improves as well.

After tests with different scenes this method gives a slightly better result than the former and is most likely the best choice. It is however a bit more demanding on the texturing fillrate than the first alternative so both should be tested in the VR-environment before the final decision.

Much of the code for this approach, the proposed depth-of-field algorithm, can be found in Appendix B.

Figure 7
The alpha channel of the images from figure 6. Black is a part of the image that is in focus and white is completely out of focus. The bottom image is used to interpolate the colors from the previous figure in the last combiner set. If alpha is 1.0 (white) in a pixel the color is fetched from the most blurred texture. The lower the alpha value gets, the more color from the original texture is blended.
Chapter 5

Summary

This chapter summarizes the report and discusses the aspects of the result that are problematic and what could be improved.

The final depth-of-field algorithm fulfills the call for real-time stereo rendering which must be considered the most important in a virtual reality application. It is not in any way a real DoF, only an appearance of this optical phenomenon. The transition from focus to not-focus is without visible seams but not correct in a physical point of view, a more correct approach would include calculating the circle-of-confusion for each pixel and blur it accordingly. The mathematics involved in depth of field- and focus calculations are described in [2] and [5].

Because of the methodology in the approach to depth of field and focus presented in this report, these calculations are difficult to apply. With a texture approach, the texture is the limit for the amount of blur applied to the scene. The offset can not be allowed to increase above a certain threshold. The amount of blur each pixel receives is also hard to determine exactly. To mimic the human vision much better precision in these areas would be needed (we all “know” how human vision should look). Simulating the effect of a camera or microscope in a synthetic environment is another matter, in this case the user has no natural reference to evaluate the correctness.

Using mipmaps to blur the parts of the scene that are out of focus is very fast but lacks somewhat in the visual quality. Blending different mipmaps via
multi-textures and combiners improves the results greatly but there are still artifacts. The problem is noticeable when objects move around in the scene while the focus is kept stationary, the colors and textures can seem to live a life of their own which might be a bit disturbing. It can be controlled by the amount of blur applied and how large the region between focus and completely out-of-focus is.

This problem is not apparent when offset is used several times to blur the texture already in the pixel-buffer, without the use of mipmaps. This property combined with the smoother blur that is generated gives this approach the advantage. The scalability of this algorithm is also good in the sense that it would be easy to improve the quality of the blur by simply repeating the offset an extra time.

The color leaking that occurs is minimized but not eliminated, this results in some optical errors that might be detected around the edges of the objects. Preferably, all objects should be blurred independently of one another instead of together in the rendered texture. If an object is partly hidden from view by another object, the texture contains no color information for the occluded parts causing the error. [4] presents an approach for visibility sorting and compositing as a way to avoid this issue.

The final algorithm is made specifically for the GeForce3 chip, and only works on computers equipped with such a graphics card. Other cards that support vertex program can of course be used but the code would have to be modified because of the extensions. Unfortunately the OpenGL standard is far behind the technology in today's graphics hardware which is why there are numerous extensions developed by the manufacturers of the graphic chips.

Furthermore, the use of vertex programs demands very specific solutions since this program is executed for each vertex and the number of commands should be kept to an absolute minimum. For example, the vertex program in Appendix B is written for a static camera position, at the origin, pointing in the negative z-direction. A general solution for a movable camera is naturally possible but would decrease performance unnecessarily for this application.

The rendering of transparent objects is a problem, which has not been solved adequately. Transparent objects are either rendered without the correct amount of blur or the objects behind them are. For this application it was deemed a minor problem and ignored, but there is a possible solution for applications with lesser demand for speed.

It would be a simple process to separate the first render-to-texture to one with only opaque objects and converting this to a blurred texture, followed by a second render, with the transparent objects. This is also converted to a blurred texture.

All these textures contain a distance-map in the alpha-channel. To get the depth-of-field effect, the second render is first done for the opaque objects and the texture of the corresponding scene is applied to a quad and blended. When this is done the transparent objects are rendered and a second
quad with the transparent-object-texture is added. By enabling and disabling the depth-buffer, using color masks and maybe separating the clearbuffer-commands it should be possible to manage this. The framerate would probably drop significantly, even though the texture of the transparent object does not need a high quality blur.

Actual testing of the algorithm in an environment similar to the final VR-application has not been done, mainly because the display equipment need two separate graphic inputs which no GeForce3 card can provide, as of yet. Testing is of course important to make sure that the user has no problem with the blurred graphic that is being displayed inches before his eyes.

Combining the developed focus algorithm with OpenGL Optimizer remains to be done. A few attempts have been made but the timeframe of this project did not allow dwelling on this problem for too long. Switching between pure OpenGL and Optimizer code is a walk in the park, initializing and utilizing the GeForce3 specific extensions is also easily managed. Creating a pixelbuffer and making it the current rendering context is done pretty much the same way as in OpenGL, the same way goes for vertex programs and register combiners.

The problem is how the scene is rendered. In Optimizer this is achieved with the command \texttt{csDrawAction(...) which tells Optimizer to start traversing the scene graph. What happens next is hidden from the programmer and this is what is causing trouble. To use the vertex program to calculate the new alpha value based on distance from focus we need to execute the vertex state program between each object. This since the material properties and half-angle vector needs to be re-calculated between each object for the vertex program to calculate the correct lighting. In OpenGL this is not a problem since the programmer has full control over how the scene is rendered. This is now hidden from view when Optimizer is used to draw the content of the scene graph. One solution is of course to not use the Optimizer draw command but rather make a scene-graph traversal method of your own. While this is certainly possible it might be an unneccessary overkill. It might be possible to simply override some properties of the csDrawAction command but the lack of source code does not help.
Due to the nature of this project, a lot of the information regarding both theory and implementation has been found on the internet. The most important documents and websites are found in this chapter.

Books:


Articles:


**Documents from http://developer.nVidia.com**


[7] Chris Wynn, OpenGL Vertex Programming on Future-Generation GPUs (VertexPrograms.pdf)

[8] Mark Harris, Dynamic Texturing (DynamicTexturing.pdf)


[10] Chris Wynn, Using P-Buffers for Off-Screen Rendering in OpenGL (PixelBuffers.pdf)


[12] Sébastien Dominé and John Spitzer, Texture Shaders (TextureShaders.pdf)


**Documents from http://www.opengl.org**


**Documents from http://www.sgi.com**

Appendix A
nVidia specific extensions

Appendix A gives an introduction to two nVidia specific extensions to the OpenGL standard. For more information on the subject, the reader can visit the nVidia developer site at http://www.nVidia.com

A.1 Vertex Programs

Programmable graphics hardware is a new technology, introduced by nVidia’s GeForce3 chip to the consumer market. The vertex program extension to the OpenGL standard enables the programmer to redefine the section of the graphics pipeline that handle transform and lighting of vertices. In ordinary case, every vertex is sent through the pipeline and each is transformed to view-coordinates and the specified lighting is applied. A vertex program is a small assembler-like program that is executed for each vertex instead of the ordinary transform and lighting calculations (Figure 8). Since the program completely replaces the original computations, these calculations have to be coded by hand and included in the vertex program if they are needed. Since the programming is very low-level and affects the hardware, new functionality can be implemented at a very low cost. There are many areas of application, for instance custom lighting, custom blending and skinning, physics, texture transformations and much more.

There are two types of vertex programs, one that executes per-vertex, simply called a vertex program and one that only executes per-begin/end block called a vertex state program. The difference will (hopefully) become apparent.
Input to the vertex program is an unlit vertex in world-coordinates and output is a transformed vertex. The program cannot create new vertices or delete them. Besides coordinates each vertex might also have additional information such as colors, normal, weight, fog and texture coordinates, if they have been defined in the OpenGL application. These are stored in the 16 input registers, each register contains four float-values. In fact, all register that are available to a vertex program consists of four float-values.

Output from the vertex program is stored in 15 output registers, at least a transformed position and a primary color is needed but secondary color, point size, fog and texture coordinates etc can be output as well. The input registers are read-only and the output registers are write-only. There are two more kinds of registers, 12 temporary registers that are read/write-able and 96 constant registers that are read-only.

The temporary registers are used to store intermediate values in the calculations performed by the vertex program and are initialized to zeroes for each new vertex. It is consequently not possible to affect another vertex than the one currently in processing.

The constant registers on the other hand are only write-able from outside the vertex program, either by sending parameters to it from the application or via a vertex state program. From the main program it is simply to add the command `glProgramParameter4fNV(...)`, specify which register 0-95 and supply four float-values. Another important command is `glTrackMatrixNV(...)` which automatically tracks the specified matrix (such as GL_MODELVIEW, GL_MODELVIEW_PROJECTION) and stores it in four constant registers.

With a vertex state program however, it is possible to perform calculations that are only made once per object and store the results in the constant registers instead of doing the same computation over and over for every vertex. This program must be executed before the vertices start to wander down the rendering pipeline though.

The syntax is similar to assembler but with an extended instruction set. There are 17 instructions in total and they operate on complete registers.
(four values). They range from the simple \texttt{ADD} (add), \texttt{MUL} (multiply), \texttt{MOV} (move), \texttt{DP4} (dot product), \texttt{MAD} (multiply and add) to more elaborate like \texttt{LIT} (supplying a few specified lighting parameters calculates the lighting coefficients).

A vertex program is defined as arrays of Glubytes or strings (can be read from a separate file) and handled similar to texture objects with commands such as \texttt{glGenProgramNV(...)}, \texttt{glLoadProgramNV(...)} and \texttt{glBindProgramNV(...)}.

To use the vertex program instead of the built-in T&L, the program must first be bound and then enabled. Until vertex programs are disabled again all vertices sent down the render-pipeline will be affected by it.

For more theory on the subject of programmable hardware the reader can turn to [3] and the nVidia web-site.

\section{A.2 Register Combiners}

If register combiners are enabled, the current combiner setup defines how textures are blended when they are applied to the geometry. The GeForce2-chip supports two general combiners and one final combiner while GeForce3 supports eight general and one final.

Each general combiner has four rgb-inputs and four alpha-inputs, the calculations are separate for color and transparency. Each combiner can output three rgb-vectors and three alpha-values, which can be used by the next combiner in line.

The inputs are labeled \(A\), \(B\), \(C\) and \(D\), the calculations include dot product (\(A \times B\) and/or \(C \times D\)), multiplication (\(A \times B\) and/or \(C \times D\)) or muxsum (\(A \times B + C \times D\)).

Whether one or all eight combiners are enabled the final combiner is always active and calculates the final output.

Input is one alpha value that is outputted directly and four rgb-vectors that are combined as \(A \times B + (1-A) \times C + D\).

Input to a combiner can be the object color, specular color, texture 1-4 color, two spare colors, two constant colors for each combiner, fog color or zero. Some of these registers are write-able and can store intermediate values between combiners, others are only readable.

The command \texttt{glCombinerParameteriNV(...)} specifies how many general combiners that are used. Using \texttt{glCombinerInputNV(...)} with parameters specifies the input (\(A\),\(B\),\(C\) or \(D\)), which registry to get the data from, which combiner (0-7), how the data is mapped and whether it is rgb or alpha. The corresponding command for the final combiner is \texttt{glFinalCombinerInputNV(...)}.
To determine the calculations that will be applied to the input data `g/CombinerOutputNV(...)` is used. With this command the output registries are specified too along with optional scale or bias factors.
Appendix B

OpenGL and C++

Appendix B contains some of the OpenGL code for the depth-of-field algorithm, but not all. It is commented so no further explanation will be presented in the appendix.

```c
/* This function is called once, before the main display loop is entered.
 */
void glinit(void)
{
    setupLighting();
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glClearColor(0.0, 0.0, 0.0, 0.0);

    // Initialize multiformat
    if(initMultiTex()){
        fprintf(stderr, "Error: initMultiTex\(\)\n\n\n\nexit(-1);
    }

    // Initialize vertex program (only done once) then set a few parameters
    if(!initVertexProgram()){
        fprintf(stderr, "Error: initVertexProgram\(\)\n\n\n\nexit(-1);
    }

    vProg = loadProgram("focusVP.vp", VERTEX_PROG);
    vProg2 = loadProgram("texcoordVP.vp", VERTEX_PROG);
    vState = loadProgram("focusState.vsp", VERTEX_STATE);

    // Initialize the register combiner and define it
    if(!initRegCombiner()){
        fprintf(stderr, "Error: initRegCombiner\(\)\n\n\n\nexit(-1);
    }

    // Initialize the pixelbuffer and define its contents
    if(!initPBuffer(pbuf_x, pbuf_y)){
        fprintf(stderr, "Error: initPBuffer\(\)\n\n\n\nexit(-1);
    }
```
// Get visible window handle and link with pbuffer to share textures
if(!wglShareLists(rendr, pbuffer)) printf("wglShareLists failed\n");
wglMakeCurrent(hpbDC, pbuffer);

// Track the concatenation of the modelview and projection matrix in registers c[0]-c[3].
gTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 0, GL_MODELVIEW_PROJECTION_NV, GL_IDENTITY_NV);

// Track the modelview matrix in registers c[4]-c[7].
gTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MODELVIEW, GL_IDENTITY_NV);

// Set the light direction and color in c[12] and c[13..15]
gProgramParameter4fvNV(GL_VERTEX_PROGRAM_NV, 12, direction);
gProgramParameter4fvNV(GL_VERTEX_PROGRAM_NV, 13, lmodel_ambient);
gProgramParameter4fvNV(GL_VERTEX_PROGRAM_NV, 14, diffuse);
gProgramParameter4fvNV(GL_VERTEX_PROGRAM_NV, 15, specular);

// Set the view vector in c[16]
gProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 16, 0.0, 0.0, -1.0, 0.0);

// Set the view vector in c[16]
gProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 16, 0.0, 0.0, -1.0, 0.0);

wglMakeCurrent(hDC, rendr);

/* The display routine (with focus enabled) first renders the scene into the pixelbuffer, which has a resolution
of 512 x 512. This render is with the vertex program enabled because we want the texture to have an alpha value
depending on the distance from focus...
The limited resolution is because the next step is to take the content of the pixelbuffer and convert it into a texture,
that is taken care of by the genTexture() routine.
Then a screen-covering quad is rendered multi-textured with this texture, but with a vertex program enabled which offsets the
texture coordinates. The contents of the pbuffer is again converted to a texture and a new quad is rendered this
time with the blurred texture, yielding an even more blurred texture.
Then the ordinary buffer is set to active and the entire scene is rendered again in full resolution to give the in-focus
part of the scene.
After this, blending is enabled and a plane aligned with the xy-axis is rendered without writing to the depthbuffer,
this plane is textured with the first and the last texture from the first pbuffer renderings.
Finally all transparent objects are rendered... they will not be blurred though.
*/

void display(void)
{
    // Set pbuffer to be active
    wglMakeCurrent(hpbDC, pbuffer);
    setupRegCombiner(set0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glPushMatrix();

    // Variables for the VP, determining where focus is and how large the region is
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 30, -1.0f, focalPlaneZ, fRegion, focalCoeff);

    // Render the scene--------------------
    renderScene(1);
    glPopMatrix();

    // Generate first texture
    genTexture(&blurTex1);

    // texture offsets
    float texel = 1.0f / 512.0f;
    texel = OFFamount*texel;
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 31, texel, -texel, 0.0, 0.0);
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 32, -texel, -texel, 0.0, 0.0);
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 33, -texel, texel, 0.0, 0.0);
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 34, texel, texel, 0.0, 0.0);

    // Blur by offset
    blurPlane(&blurTex1);

    // Generate second texture
    genTexture(&blurTex2);

    // new texture offsets
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 31, texel, 0.0, 0.0, 0.0);
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 32, -texel, 0.0, 0.0, 0.0);
}
void blurPlane(GLuint* id)
{
    // Bind texture to multitextures 0-3
    glActiveTextureARB(GL_TEXTURE0_ARB);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, *id);
    glActiveTextureARB(GL_TEXTURE1_ARB);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, *id);
    glActiveTextureARB(GL_TEXTURE2_ARB);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, *id);
    glActiveTextureARB(GL_TEXTURE3_ARB);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, *id);

    // Enable the texcoord vertex program.
    glUseProgramNV(GL_VERTEX_PROGRAM_NV, vProg2);
    glEnable(GL_VERTEX_PROGRAM_NV);

    // Switch back to the ordinary buffer
    wglMakeCurrent(hDC, rendr);
    setupRegCombiner(set1);
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDisable(GL_DEPTH_TEST);
    renderPlanes();
    glEnable(GL_DEPTH_TEST);
    glPopMatrix();

    // Render the scene---------------------
    renderScene(0);
    glPopMatrix();

    // Blend with alpha-value from the blurred texture
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glUseProgramNV(GL_VERTEX_PROGRAM_NV, vProg2);
    glEnable(GL_VERTEX_PROGRAM_NV);

    // Enable the texcoord vertex program.
    glUseProgramNV(GL_VERTEX_PROGRAM_NV, vProg2);
    glEnable(GL_VERTEX_PROGRAM_NV);

    // Switch back to the ordinary buffer
    wglMakeCurrent(hDC, rendr);
    setupRegCombiner(set1);
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDisable(GL_DEPTH_TEST);
    renderPlanes();
    glEnable(GL_DEPTH_TEST);
    glPopMatrix();

    // Render transparent objects
    renderTransparent();
    glDisable(GL_BLEND);
    swapBuffers();
}

/* This function renders a screen-covering quad and textures it. The texture referred by the pointer "id" is bound to
four multi-textures which each is offset by the vertex program. This results in a blurred result.
By calling this function repeatedly from the display function more and more blur is achieved.
*/
void blurPlane(GLuint* id)
{
// Enable the combiner

gEnable(GL_REGISTER_COMBINERS_NV);

// Render the quad with ordinary texture coordinates, the vertexprog deals with multiTex. coord.

glBegin(GL_QUADS);

gTexCoord2f(0.0, 0.0); glNormal3f(0.0, 0.0, 1.0); glVertex3f(-xClip, -yClip, zNear);
gTexCoord2f(1.0, 0.0); glNormal3f(0.0, 0.0, 1.0); glVertex3f(xClip, -yClip, zNear);
gTexCoord2f(1.0, 1.0); glNormal3f(0.0, 0.0, 1.0); glVertex3f(xClip, yClip, zNear);
gTexCoord2f(0.0, 1.0); glNormal3f(0.0, 0.0, 1.0); glVertex3f(-xClip, yClip, zNear);
gEnd();


gDisable(GL_REGISTER_COMBINERS_NV);
gDisable(GL_VERTEX_PROGRAM_NV);
gDisable(GL_TEXTURE_2D);
gActiveTextureARB(GL_TEXTURE2_ARB);
gDisable(GL_TEXTURE_2D);
gActiveTextureARB(GL_TEXTURE1_ARB);
gDisable(GL_TEXTURE_2D);
gActiveTextureARB(GL_TEXTURE0_ARB);
gDisable(GL_TEXTURE_2D);

}

/* Renders a plane with two textures of the original scene (rendered in the pixelbuffer).
The first texture is not blurred and the second is the one with the most blur.
These textures are "combined" by the register combiner (the exact combination formula is described in the
reg.comb section).
This is to achieve a good blend with the in-focus part of the scene.
*/

void renderPlanes(void)
{

// Bind texture to multitextures 0-3

gActiveTextureARB(GL_TEXTURE0_ARB);
gEnable(GL_TEXTURE_2D);


// Enable the combiner

gEnable(GL_REGISTER_COMBINERS_NV);

// Render the quad with ordinary texture coordinates, the vertexprog deals with multiTex. coord.

gBegin(GL_QUADS);

// Bind texture to multitextures 0-3

gActiveTextureARB(GL_TEXTURE0_ARB);
gEnable(GL_TEXTURE_2D);

}
This is the vertex program calculating transforms, lighting and the distance from focus.

!!VP1.0

// Vertex program for specular and diffuse lighting and calculation of alphavalue dependent on distance from focus.
// c[0]...c[3] contains the concatenation of the modelview and projection matrices.
// c[12] contains the (object space) light direction.
// c[13] contains the ambient light color
// c[14] contains the diffuse light color
// c[15] contains the specular light color
// c[16] contains the (object space) view-vector
// c[17] contains specular powers and(0, power, 0, 0)
// c[18]...c[20] contains material color: ambient, diffuse and specular
// c[21]...c[23] contains the premultiplied colors (that is incoming light * reflection properties).
// c[24] contains the unit half angle vector H.
// c[30] contains the z-value for the focal-plane, the focus region and a linear coefficient (-1, z-focus, focus_region, coeff);

// Calculate homogeneous clip space position of the vertex
DP4  v[HPOS].x, c[0], v[OPOS];
DP4  v[HPOS].y, c[1], v[OPOS];
DP4  v[HPOS].z, c[2], v[OPOS];
DP4  v[HPOS].w, c[3], v[OPOS];

// Output texture coordinates without transformation
MOV  v[TEX0], v[TEX0];

// Determine if vertex is back-facing
DP3  R4, v[NRML], -c[16];
SGE  R5, R4.x, c[17].x;

// Calculate and output the color of the vertex after lighting has been applied
DP3  R0.x, c[12], v[NRML];
DP3  R0.y, c[24], v[NRML];
MOV  R0.w, c[17].y;
LIT  R1, R0;
MUL  R2, R1.x, c[21];
MAD  R3, R1.y, c[22], R2;
MAD  R4, R1.z, c[23], R3;

// Calculate and output the alpha value of the vertex,
// z(f) is always positive, z(v) is always negative.
DP4  R6, c[6], v[OPOS];
ADD  R6, R6, c[17].z;

// Need negative absolute value of the distance vector.
MIN  R6.w, R6.w, -R6.w;
// Set a region which is in focus by adding c[30].z to the (neg) distance vector, making sure it still is not a positive value. Multiply this with a coefficient, c[30].w
ADD  R6.w, R6.w, c[30].z;
MIN  R6.w, R6.w, c[17].x;
MUL  R4.w, c[30].w, -R6.w;
MUL  c[COL0], R4, R5;

END