Writing Stereoscopic Software for StereoGraphics® Systems Using
Hewlett Packard UNIX Workstations

By Bob Akka StereoGraphics Corporation September 21, 1998
www.stereographics.com

1. Introduction

Hewlett Packard workstations support stereo via two different approaches:

- Full-screen “Starbase stereo”, which uses sync-doubling, and
- Windowed OpenGL stereo buffering.

Most recent HP graphics devices, including the CRX-24, CRX-48, Visualize-48, FX2, FX4, and FX6, support full-screen stereo.

The newest HP graphics devices, the FX4 and FX6, support windowed OpenGL stereo (in addition to supporting full-screen stereo). Although windowed OpenGL stereo is not as universally supported on HP’s installed base, HP plans to support OpenGL stereo on its new graphics workstations from now on. We expect that most developers will prefer to do their stereo development on HP using windowed OpenGL stereo, due to the numerous advantages of windowed application development. There will be a brief section near the end of the document about full-screen Starbase stereo (**9. Full-Screen Starbase Stereo Formatting**), for developers who remain interested in that approach.

This document relates to Hewlett-Packard’s UNIX based systems only. HP now also makes Windows NT based systems that use the FX6 graphics card. If you are interested in developing stereo applications that will run on with HP graphics devices while running Windows NT, please see the document “Writing Stereoscopic Software for StereoGraphics Systems Using Microsoft Windows OpenGL” instead.

This document covers the steps that you need to take to:

- Set up for stereo development,
- Check the graphics hardware for stereo buffering support,
- Enable stereo buffering in a display window,
- Write to separate stereo buffers,
- Do stereo perspective projections, and
- Set up your projections for excellent stereo image quality.

We strongly recommend that you also obtain the HP version of “**OGLPlane**”, a very simple stereo example program that illustrates most of the concepts that are discussed in this document. The “**OGLPlane**” source code is available from StereoGraphics.

2. Setting Up Your Graphics Hardware To Support Stereo

HP graphics devices support stereo only in certain display modes. With the FX4 and FX6 cards, windowed OpenGL stereo is supported in the following two display modes:

- 1280 x 1024, 94 Hz, or
- 1024 x 768, 98 Hz

Note that the 1280 x 1024, 94 Hz, graphics may require a 21" monitor, and may not be compatible with HP's smaller monitors. The 1024 x 768, 98 Hz, mode should work with all HP monitors.

You can change the system's display mode in one of a few ways. One method is to log in as "root" and run "`/opt/graphics/common/bin/setmon`", and then choose the desired display mode from the menu. Another method is to log in as "root" and run "`/usr/sbin/sam`", then select "Display", and then "Monitor Configuration"; next, select the icon representing the FX display device, and choose "Modify Monitor Type" from the "Actions" menu, and finally choose the desired display mode from the menu.

The display mode menus should mention "Stereo in a Window" under the "Description" column of the video modes that will support stereo.

3. Initializing a Window to Enable Stereo

In order to utilize OpenGL's stereo buffering support, a graphics window must be initialized using `glXChooseVisual()`, with an attributes flag array that includes `GLX_STEREO`. Here is some sample code, from the initialization of an application's graphics window:

```
Display *display;
XvisualInfo *visualInfo;
int StereoAttributes[] = {GLX_RGBA, GLX_DOUBLEBUFFER, GLX_RED_SIZE,
      1, GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1, GLX_DEPTH_SIZE, 1,
      GLX_STEREO, None};

display = XtDisplay (WinWidget); /*WinWidget was already init-ed*/
visualInfo = glXChooseVisual (display, DefaultScreen (display),
      stereoAttributes);
if (visualInfo == NULL)  {
  /* Windowed OpenGL Stereo not supported */
}
```

If `glXChooseVisual()` returns a null pointer, that indicates that the user's graphics configuration is not set up for stereo buffering. Most likely, either the user is not using an FX6 or FX4 graphics device, or they are not running in one of the display modes (including the proper sync rate) required for OpenGL stereo buffering. You will want to notify the user, and then either exit, or try to run the application without stereo buffering (you might want to try calling glXChooseVisual() again, with a different attributes flag array that doesn't include GLX_STEREO).

4. Writing to Separate Stereo Buffers

The glDrawBuffer() OpenGL function allows you to specify which buffer subsequent OpenGL drawings and renderings should be directed to. With normal non-stereo OpenGL double-buffering, you will typically draw to the "back buffer" (glDrawBuffer(GL_BACK);), and then "swap buffers" (wglSwapLayerBuffers(hdc, WGL_SWAP_MAIN_PLANE);) in order to put what you had drawn to the back buffer onto the "front buffer", which represents the visible display.

Once you have initialized a window to support stereo buffering using SetPixelFormat() (see "**3. Initializing a Window to Enable Stereo**"), you can specify your drawing buffer as GL_BACK_LEFT or GL_BACK_RIGHT, which will result in subsequent OpenGL drawings and renderings appearing only in one eye's drawing buffer. After drawing to both the left and right back-buffers, do a single wglSwapLayerBuffers() call to put the back stereo buffers' contents to the front (visible) stereo buffers. Here is an example in which different colored rectangles are drawn to left and right eye buffers:

```
// pCDC already declared as pointer to current CDC

           // Clear both back buffers
glDrawBuffer (GL_BACK);
glClearColor(0.2f, 0.2f, 0.2f, 0.0f);
glClear (GL_COLOR_BUFFER_BIT);

           // Draw left eye view
glDrawBuffer (GL_BACK_LEFT);
glColor3b (0, 0, 127);  // blue
glRectf (-0.8f, -0.8f, 0.2f, 0.2f);

           // Draw right eye view
glDrawBuffer (GL_BACK_RIGHT);
glColor3b (127, 0, 0);  // red
glRectf (-0.2f, -0.2f, 0.8f, 0.8f);

           // Put what was just drawn onto the display
BOOL bSuccess = wglSwapLayerBuffers (pCDC->m_hDC,
     WGL_SWAP_MAIN_PLANE);
```

As the above code example shows, you can still use glDrawBuffer(GL_BACK) to draw to both left and right back-buffers at once. Also note that you can use glDrawBuffer() to access the left and right front-buffers directly (GL_FRONT_LEFT and GL_FRONT_RIGHT).

5. An Introduction to Stereoscopic Perspective Projections

You now know everything you need to know to put different stuff into left-eye and right-eye buffers. Now comes the interesting part: Doing perspective projections that will result in a stereoscopic effect that is both geometrically correct and pleasing to look at.

A good quality stereo image is composed of two stereo pair elements, each of which being a perspective projection whose “center of projection” (let’s use the simpler term, “camera”, instead) is offset laterally relative to the other camera position.

Let’s start with the mathematical representation of a simple non-stereo perspective projection. Assuming that the camera lies on the positive *z*-axis at *(0, 0, d)* (this document uses a “right handed” coordinate system), *d* being the distance from the camera to the *xy*-plane, *(x, y, z)*, projects onto the *xy*-plane at:

(Equation 1)

$$\left(\frac{xd}{d-z} , \frac{yd}{d-z} \right)$$

Thus, for example, if the camera is placed at *(0, 0, 9)*, the arbitrary point *(8, -5, -3)* will project to *(6, -3.75)*. See Figure 1 for a visual representation of perspective projection.

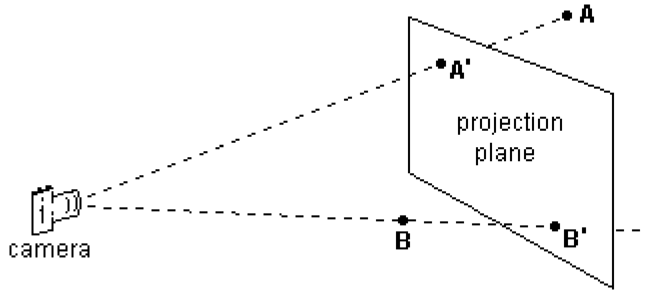

Figure 1: A Perspective Projection of Two Points

Next, we’ll introduce a camera offset to the projection. To do a left-camera perspective projection, we’ll offset the camera to the left by half the overall camera separation, *(c/2)*. Except that, to make the math easier, we’ll offset the entire scene to the right instead of offsetting the camera to the left. So, the left-camera projection of *(x, y, z)* now calculates to:

(Equation 2)

$$\left(\frac{\left(x + \frac{c}{2}\right)d}{d-z} , \frac{yd}{d-z} \right)$$

And, for the right-camera projection, we’ll offset the entire scene to the left; the right-camera projection of *(x, y, z)* now calculates to:

(Equation 3) $$\left(\frac{\left(x - \frac{c}{2}\right)d}{d - z}, \frac{yd}{d - z} \right)$$

Let's go back to the example, with the original camera at *(0, 0, 9)* and the arbitrary point *(8, -5, -3)*. If we use a camera separation of *1*, the left camera projects to *(6.375, -3.75)*, and the right camera projects to *(5.625, -3.75)*.

Notice how, in the above example, the arbitrary point ends up projecting *0.75* units to the right in the left-camera view relative to its projection in the right-camera view. If one projection is superimposed over the other in a stereo viewing system, this will result in the arbitrary point appearing in what is the optics folks call "negative parallax", meaning that it will appear to float in front of the display surface. Conversely, if a scene element appears in the left-camera view to the left of where it appears in the right-camera view, it will appear with "positive parallax", meaning that it will seem to reside somewhere behind the display surface. "Zero parallax" is what happens when the left-camera projection of a point perfectly matches its right-camera projection; a scene element projecting at zero parallax will appear to reside right at the display surface. See Figure 2 for an illustration of how one perceives negative and positive parallax effects.

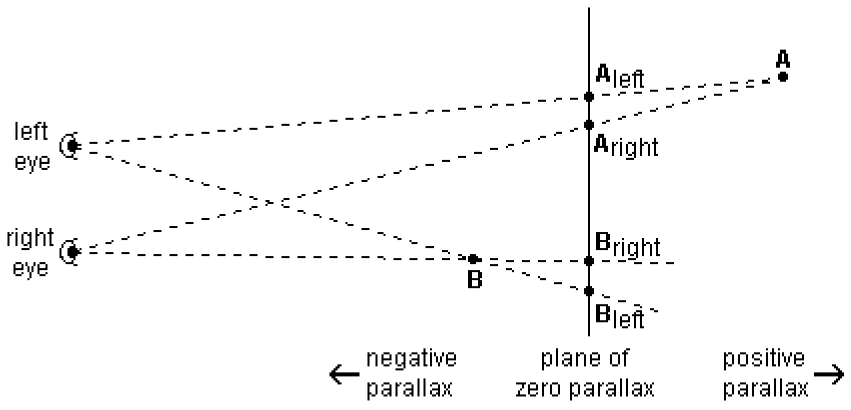

Figure 2: Perception of Parallax Effects

Generally, a pleasing, well balanced stereo image will make use of both negative parallax and positive parallax, and at least some of the 3D scene will project at or close to zero parallax. Unfortunately, if one uses Equations 2 and 3 to calculate the stereo pair projections *all* possible 3D points will project to negative parallax. All negative parallax tends to be uncomfortable to view.

We can fix that by simply shifting the projected values leftward for the left-camera projection, and rightward for the right-camera projection. If we shift the projected points by the same amount of the original camera offset, the resulting geometry will place the original projection plane precisely at zero parallax. Scene elements originally placed in front of the projection plane will project to negative parallax, and scene elements originally placed behind the projection plane will project to positive parallax. Here are the new equations for the left eye:

(Equation 4)
$$\left(\frac{\left(x+\frac{c}{2}\right)d}{d-z} - \frac{c}{2},\ \frac{yd}{d-z} \right)$$

...And for the right eye:

(Equation 5)
$$\left(\frac{\left(x-\frac{c}{2}\right)d}{d-z} + \frac{c}{2},\ \frac{yd}{d-z} \right)$$

Returning to our example, *(8, -5, -3)* now projects to *(5.875, -3.75)* for the left camera, and *(6.125, -3.75)* for the right camera, shifting it into positive parallax, and making it appear behind the display surface. This makes sense, since the scene element's original *z*-coordinate of *-3* places it behind the projection plane, whose *z*-coordinate is *0*.

The two projections that we have derived in Equations 4 and 5 above are called "parallel axis asymmetric frustum perspective projections". Note that, even after translating the scene in one *x*-axis direction, and then translating the projected scene in the opposite *x*-axis direction, the projection axes, the camera-target vectors of the two stereo pair cameras, remain parallel. Our final step, in which we shift the projected scene, makes the frustums of the parallel projections asymmetrical, meaning that each camera's final projection shows more of the scene to one side of its axis than the other.

To summarize, the end result is a pair of perspective projections rendered from differently offset camera positions, with frustum asymmetry applied to comfortably balance the overall stereo parallax effect.

It is a common error for programmers to do stereo projections using a "toe-in" camera model. With this model, the cameras are still offset to one side or the other, but the camera-target vectors are not parallel, and converge on a single point. Some developers use this method because it is conceptually simpler and, sometimes, easier to implement than parallel axis asymmetric frustum projections. Also, with this method, a balance between positive parallax and negative parallax is achieved without the need to shift the projections as we do within Equations 4 and 5. Unfortunately, the camera toe-in approach is geometrically incorrect, and leads to some variable vertical misalignment between stereo pair elements, which can make for uncomfortable viewing. We do not recommend the "toe-in" camera model.

6. Implementing Asymmetric Frustum Projections

Now that I've explained the proper way to do stereo projections mathematically, let's talk about how to do these projections in your application. If your application already does a perspective projection, the elements that you need to add, for each of two otherwise identical stereo pair perspective projections are:

- Center of projection (camera) offset, and
- Post-projection shift, or frustum asymmetry

The camera offset is easily accomplished using OpenGL's `glTranslate()` functions (this document will use the double-precision version of the function, `glTranslated()`). In order to effect the pre-projection camera offset, we actually need to call `glTranslated()` just *after* the code that does the perspective projection. As in the mathematical discussion above in "**5. An Introduction to Stereoscopic Perspective Projections**", we will effectively translate the camera by instead translating the entire scene in the opposite direction. Here's the code (where `StereoCameraOffset` equals half the overall camera separation, negative for the left camera and positive for the right camera):

```
   glTranslated (-StereoCameraOffset, 0, 0);
```

There are a few different ways to make each stereo projection's frustum asymmetrical. One imperfect approach would be to simply apply an offset (towards the left for the left-camera rendering, towards the right for the right-camera rendering) when viewporting the regular (symmetric frustum) perspective projections. Though this approach is sometimes easy to implement, and the resulting stereo images are geometrically correct, the disadvantage is that the left and right edges of the two stereo pair element viewports will not line up with each other.

The best way to implement each stereo projection's frustum asymmetry is to integrate the frustum asymmetry into the perspective projection. The OpenGL `glFrustum()` function can be altered to do an asymmetric frustum projection. Here's how:

```
   glFrustum (FrustumLeft + FrustumAsymmetry, FrustumRight +
        FrustumAsymmetry, FrustumBottom, FrustumTop, NearClipDistance,
        FarClipDistance);
```

...Where `FrustumLeft`, `FrustumRight`, `FrustumBottom`, `FrustumTop`, `NearClipDistance`, and `FarClipDistance` are the arguments of the original symmetric frustum projection (since `glFrustum()` assumes that the camera is at the coordinate system origin, `FrustumLeft` should equal `-FrustumRight`, and `FrustumBottom` should equal `-FrustumTop`), and where `FrustumAsymmetry` is positive for the left-camera projection, and negative for the right-camera projection.

If your application, rather than using `glFrustum()`, does its perspective projections using a transformation matrix of your own creation, you can still integrate frustum asymmetry into the perspective projection matrix. Just multiply the existing projection matrix by a transformation matrix that represents a lateral translation (leftward for the left-camera projection, rightward for the right-camera projection).

A few miscellaneous details to mention:

- `glFrustum()` assumes that the center of projection (camera) is at the coordinate system origin (unlike the equations used in **5. An Introduction to Stereoscopic Perspective Projections**, where the camera was along the positive *z*-axis). Thus, you may need to apply translation transformations to your scene in order to place the camera at the origin.

- `glFrustum()` also assumes that the projection axis (camera-target vector) is along the coordinate system *z*-axis, in the negative *z* direction. If that is not already the case, you will need to either apply transformations to your scene, or adapt your projection geometry. Note that the stereo camera offset should always be in a direction that is perpendicular to the original camera-target vector.

- Finally, `glFrustum()` does projections such that the projection's "up-vector" (the 3D direction that will appear as "up" in the final rendering) is along the coordinate system's positive *y*-axis. If you are working with a projection geometry that renders uses some kind of roll angle, remember that the stereo camera offset should always be in a direction that (in addition to being perpendicular to the original camera-target vector) is perpendicular to the projection's up-vector. Also, the projection's frustum asymmetry should always be along an axis that is perpendicular to the projection's up-vector.

7. Appropriate Camera Offset and Frustum Asymmetry

In the above discussion about implementing asymmetric frustum projections, two important questions were left unanswered:

- How much camera-offset should one use, to get a stereo effect that is strong enough to offer an effective sense of depth, yet not so strong that viewing is uncomfortable?
- How much frustum asymmetry should be applied, in order to yield a pleasing balance between positive and negative parallax?

Both questions get into the realm of a lot of evolving research, and a certain amount of controversy. For example, StereoGraphics' own documentation used to advise "don't exceed parallax values of more than *1.5°*". We know now that a person's relative acceptance and tolerance of different parallax magnitudes, on a computer screen or a projection display, is far more consistent when parallax is expressed as a percentage of viewport width, than when parallax is computed as an angular measure based on retinal disparity. Which is actually quite convenient for the software developer, since it eliminates the need to guess at variables like the user's display size and seating position.

Thus, we now advise setting up perspective projections such that the negative parallax and positive parallax effects (take off the stereo glasses to measure on-screen parallax) each fall within about *3%* or so of the image's overall viewport width. Since the viewport width represents a projection plane in the 3D scene, a good starting point for offsetting each stereo camera is about *3%* of the 3D scene's horizontal range. The 3D scene's horizontal range should be measured along the plane where the camera's projection frustum intersects the center of interest in the scene (which is roughly where we will probably want the plane of zero parallax to be).

So, let's say that we are rendering a mechanical part whose bounding box has a width of *100* units, and that mechanical part nearly fills the screen. You would probably want to offset each camera laterally by about *3* units. Here's the updated camera translation source code:

```
   double StereoCameraOffset = Xrange * 0.035 * UserOffsetAdjustment;
```

```
if (WhichEyeProjection == LEFT_EYE_PROJECTION)
   StereoCameraOffset = -StereoCameraOffset;
glTranslated (-StereoCameraOffset, 0, 0);
```

...Where `Xrange` equals the horizontal range of the scene along the desired plane of zero parallax. One way to derive `Xrange` is to get the difference between `glFrustum()`'s first two arguments, which represent the projection frustum's horizontal range at the near clipping plane, and multiply that difference by the ratio of the distance to the desired plane of zero parallax to the near clipping plane distance. Another way to calculate `Xrange` would be to multiply the distance to the desired plane of zero parallax by two times the tangent of half the horizontal field of view angle.

Also note that a "`UserOffsetAdjustment`" factor was thrown into the equation. It is a good idea to allow users to adjust the strength of the stereo effect to suit their preference. `UserOffsetAdjustment`'s default value should be `1.0`, and the user should be able to adjust its value downward towards `0.0` (which would result in no stereo effect at all), or upward (`2.0` is usually OK as an upper limit). If your user interface provides some kind of keypad stereo adjustment, `UserOffsetAdjustment` should ideally be adjusted based on multiplication (i.e.: `UserOffsetAdjustment *= 1.1`) rather than addition and subtraction.

Next, we need to quantify the amount of each stereo projection's frustum asymmetry. Recall that in "**6. Implementing Asymmetric Frustum Projections**", we added a factor called "`FrustumAsymmetry`" (positive for the left-camera projection, negative for the right-camera projection) to each of `glFrustum()`'s first two arguments:

```
glFrustum (FrustumLeft + FrustumAsymmetry, FrustumRight +
      FrustumAsymmetry, FrustumBottom, FrustumTop, NearClipDistance,
      FarClipDistance);
```

Further recall, from Equations 4 and 5, that the amount of frustum adjustment, when measured in the projection plane, should equal the amount of the original camera offset (though, in the opposite direction), in order for the frustum adjustment to place the projection plane at zero parallax. Thus, `FrustumAsymmetry` should equal `-StereoCameraOffset`.

However, `glFrustum()`'s interprets its first two parameters as *x*-axis boundaries, as measured on the near clipping plane. Yet, `FrustumAsymmetry` is being calculated based on the desired amount of frustum asymmetry in what is to become the plane of zero parallax. So, `FrustumAsymmetry` needs to be adjusted by the ratio of the near clipping distance to the desired zero-parallax distance. Here's the resulting source code:

```
double FrustumAsymmetry = -StereoCameraOffset *
      UserBalanceAdjustment;
double n_over_d = NearClipDistance / ZeroParallaxDistance;
FrustumAsymmetry *= n_over_d;
glFrustum (FrustumLeft + FrustumAsymmetry, FrustumRight +
      FrustumAsymmetry, FrustumBottom, FrustumTop, NearClipDistance,
      FarClipDistance);
```

Notice that we have introduced yet another user adjustment factor, "`UserBalanceAdjustment`". As with the stereo camera offset, it's a good idea to let the user adjust the parallax balance to their liking. `UserBalanceAdjustment`'s default value should also be `1.0`, and the user should be able to adjust its value downward towards `0.0` (which would result in no frustum asymmetry, hence all negative parallax), or upward (`2.0` is once again a good upper limit). And, like the camera offset adjustment factor, `UserBalanceAdjustment` should be adjusted multiplicatively (or, perhaps, via a dialog box) rather than by addition or subtraction.

Also note that the above code is set up such that `UserOffsetAdjustment` affects both the amount of camera offset and the amount of frustum asymmetry. This is appropriate because, if the amount of frustum asymmetry is not adjusted proportionally whenever the camera offset is changed, camera offset changes will have the side-effect of changing the scene's parallax balance.

Other stereo camera adjustment issues:

- The horizontal frustum range (`Xrange`) is directly proportional to the tangent of half the horizontal field of view angle. So, if the field of view angle is changed, the stereo camera offset and frustum asymmetry both need to be recalculated.

- The horizontal frustum range (`Xrange`) is also directly proportional to the distance from the camera to the desired plane of zero parallax. So, if you "dolly" the camera to get closer to an object, and want the plane of zero parallax to remain the same relative to that object's position, both the stereo camera offset and the frustum asymmetry factors will need to be recalculated. In fact, stereo camera offset and frustum asymmetry factors should be recalculated any time the distance from the camera to the desired plane of zero parallax changes.

- Depending on the application's rendering architecture, the projection frustum may be calculated using different units from those used in the 3D scene. In such unusual cases, it may be best to keep the frustum asymmetry factor constant and independent of the camera-offset amount (though both `UserOffsetAdjustment` and `UserBalanceAdjustment` should still affect the frustum asymmetry).

8. Other Stereo Aesthetic Issues

The last few sections have discussed stereoscopic perspective projections. Some have wondered if it is possible to do stereoscopy with orthographic projections. Though it is technically possible to do stereo orthographic projections using the camera toe-in approach, the results are so uncomfortable that we strongly advise against it. Thus, developers should design their applications such that available stereo display options will always be used in combination with perspective projection.

Similarly, for the best quality stereo, the stereo pair perspective projections should use a moderately wide field of view angle (note that a narrow "telephoto" field of view angle results in a projection geometry that is quite similar to that of an orthographic projection). For the best results, we recommend a horizontal field of view angle of $50°$ or more.

One significant issue relating to negative parallax is a phenomenon that occurs when a scene element at negative parallax is clipped by an edge of the rendering window. The problem is that a scene element that appears to be floating somewhere in front of the display surface is occluded by an edge that is at the display surface. This results in a somewhat disturbing optical contradiction in which the occluding edge is farther away than the scene element that it occludes.

Some people firmly believe that this negative parallax edge-clipping effect must be avoided at all costs, even if that means not ever using any negative parallax at all (to do this, they place the zero parallax setting at the near clipping plane). We have found that approach to be too extreme. Eliminating negative parallax tends to result in either uncomfortable amounts of positive parallax, a flattening of the stereo effect within a narrow portion of the positive parallax range, or both. Using some balance of positive parallax and negative parallax results in stereo images that are more comfortable to view, even if that approach inevitably results in some amount of negative parallax edge-clipping.

How you should deal with the negative parallax edge-clipping issue depends on the nature of your application. In a mechanical CAD application where the scene elements of interest will usually not be clipped by the window edge, it is usually safe to place the zero-parallax plane at or just in front of the object's center. However, with a "flythrough" type of application, it is often best to set the zero-parallax distance such that most (though not all) of the scene projects to positive parallax.

Finally, it should be mentioned that very high contrast values in a stereo image will often result in "ghosting", in which some of one eye's view appears as a "ghost" in the other eye's view. This tends to be caused by CRT phosphor persistence (ghosting can also be an issue when using projection displays). Green tends to be the worst offender. The workaround to this problem is to avoid extreme amounts of contrast if at all possible. Use a gray background instead of a black background (even dark gray is better than black), and if possible, try to avoid using bright white, green, and cyan scene elements against a dark background.

9. Full-Screen Starbase Stereo Formatting

Most developers will prefer to skip this section, and stick to doing stereo using the newer windowed OpenGL stereo method. However, if you are interested in supporting the widest variety of installed HP graphics devices, and don't mind the design constraints imposed by full-screen stereoscopic graphics, read on.

In addition to supporting the OpenGL stereo buffering standard on FX4 and FX6 display devices, Hewlett-Packard also supports full-screen "Starbase" stereo, on current models, and many other recent systems, including the CRX-24, CRX-48, Visualize-48, FX2, FX4, and FX6.

As with windowed OpenGL stereo, full-screen Starbase stereo requires that your system be configured in a compatible display mode. With most systems, 1280 x 1024, 72 Hz, is the display mode that supports Starbase stereo. Some graphics devices, such as the FX6, support Starbase stereo in several other modes as well. To change your system's display mode to support Starbase stereo, log in as “root” and run “`/opt/graphics/common/bin/setmon`”, and then choose the 1280 x 1024, 72 Hz, from the menu.

HP’s Starbase stereo implementation uses circuitry which, when in stereo display mode, doubles the frequency of the vertical sync signal. This causes the display to refresh at twice its normal rate, which has the effect of vertically stretching the screen display. At this doubled display frequency, whatever was previously on the top half of the display, and whatever was previously on the bottom half of the display, each take up the full display on alternate refreshes.

Thus, to format your graphics display for Starbase stereo, you need to draw left-eye information to the top half of the screen (the upper 1280 x 512), and draw right-eye information to the bottom half of the screen (the lower 1280 x 512). (Unlike the Above-Below stereo implementations on some other platforms, HP takes care of “blank interval” issues internally, so your left-eye and right-eye stereo viewports should each be drawn to fill exactly half of the regular display.) Since HP’s sync-doubling circuitry will stretch everything vertically, each eye’s rendering should have a *1:2* pixel aspect ratio such that everything originally appears vertically squashed.

Starbase stereo mode is turned on and off using the `gescape()` system call, using code such as the following:

```
gescape_arg arg1, arg2;

if (TurnStereoOn == TRUE)  /* activate stereo sync-doubling */
     arg1.i[0] = TRUE;
else                /* set flag to restore non-stereo display */
     arg1.i[0] = FALSE;

gescape (FilDes, STEREO, &arg1, &arg2); /* FilDes is the Starbase
                         client file descriptor handle, type int */
```

Remember: Sync-doubling affects the entire display, including window borders, menu bars, other application windows, etc. For this reason, a Starbase stereo application window needs to fill the *entire* display, blocking out all other application windows including the Windows Taskbar. Additionally, a Starbase stereo application window should neither have a border, nor standard title bars, menu bars, or status bars. Interface design using Starbase stereo formatting can be a challenge.

Finally, note that Sections **5** through **8**, relating to stereoscopic projections, apply to Starbase stereo formatting, as well as windowed OpenGL stereo buffering.

10. Points to Remember

- With windowed OpenGL stereo buffering, the necessary elements are:
  - Check for stereo display configuration availability, and enable stereo buffering support when initializing each graphics window.

- Render two asymmetric frustum perspective projections, one to each of the two stereo back-buffers. Then swap buffers to put the stereo pair on the display.
- Offset each stereo camera using a lateral translation (`glTranslate()`). Stereo projection axes should remain parallel to the original projection axis.
- Balance the stereo parallax by making the perspective projection frustums asymmetrical. This can be done by changing the first two arguments of `glFrustum()`, or by altering the transformation matrix that does the projection.

- Use enough stereo camera separation to yield a pleasing stereo effect, but not so much as to be uncomfortable. In general, on-screen parallax should be limited to about *3%* or so of viewport width, for both negative and positive parallax effects. For the best stereo effect, use a wide-angle perspective projection, with a field of view angle of *50°* or more.

- Parallax effects should be balanced such that there is some negative parallax, and some positive parallax. To do this, apply the right amount of projection frustum asymmetry to put the plane of zero parallax near the center of interest in any given scene. (For some applications such as "flythrough" applications, it may be desirable to use more positive parallax than negative parallax, in order to reduce the amount of negative parallax that is clipped by the window edges.)

- Stereo projection settings should not be fixed quantities. Recalculate the camera separation and frustum asymmetry any time that the field of view angle or the camera-to-target distance changes. Stereo projection setting calculations should also include user-adjustable factors.

- Full-screen Starbase stereo formatting is another approach to doing stereo display on HP UNIX systems. Though Starbase stereo will work on more older HP systems, windowed OpenGL stereo tends to be more convenient for both the application developer and the user.

11. Resources

As previously mentioned, many of the concepts discussed in this document are illustrated in the HP-UNIX version of the example program, "**OGLPlane**", which is available from StereoGraphics. Another simpler example program, "**RedBlue**" (again, you'll want the HP version), illustrates stereo OpenGL buffering without doing any stereo projections (it simply draws colored rectangles to each stereo buffer). Other example programs may additionally be available at our website, http://www.stereographics.com, and in the "Developers" directory of our ftp site, ftp.stereographics.com. StereoGraphics developer support can also be reached at support@crystaleye.com, or 415-459-4500.