Hotspot Mitigation in the StarCAVE

Jordan Rhee, Jurgen Schulze, Thomas A. DeFanti

California Institute for Telecommunications and Information Technology (Calit2), University of California San Diego (UCSD)

Abstract

Rear-projected screens such as those in Digital Light Projection (DLP) televisions suffer from an image quality problem called *hot spotting*, where the image is brightest at a point dependent on the viewing angle. In rear-projected mulit-screen configurations such as the StarCAVE at Calit2, this causes discontinuities in brightness at the edges where screens meet, and thus in the 3D image perceived by the user. In the StarCAVE we know the viewer's position in 3D space and we have programmable graphics hardware, so we can mitigate this effect by performing post-processing in the inverse of the pattern, yielding a homogenous image at the output. Our implementation improves brightness homogeneity by a factor of 4 while decreasing frame rate by only 1-3 fps.

1. Introduction

The StarCAVE at Calit2 is a room-sized immersive virtual reality environment that projects 3D images in real-time. The cave is used for displaying higher order scientific data in real time, such as protein structures and earthquake simulations. The user wears polarized glasses and stands in the center of an array of 15 screens, each driven by two projectors. The user sees images in 3D because the system uses polarizing filters to send a different image to each eye. In order to give a 360° viewing angle, the screens are rear-projected. As with all rear projected screens, the StarCAVE suffers from an image quality problem called *hot spotting*.

The software framework that runs in the cave is called Covise (See Covise). Cave applications are written as Covise plugins, and multiple plugins can run at the same time. Covise abstracts the fact that the application is running in parallel over many machines and multiple OpenGL contexts, and handles things such as stereo perspective calculations and OpenGL context management.

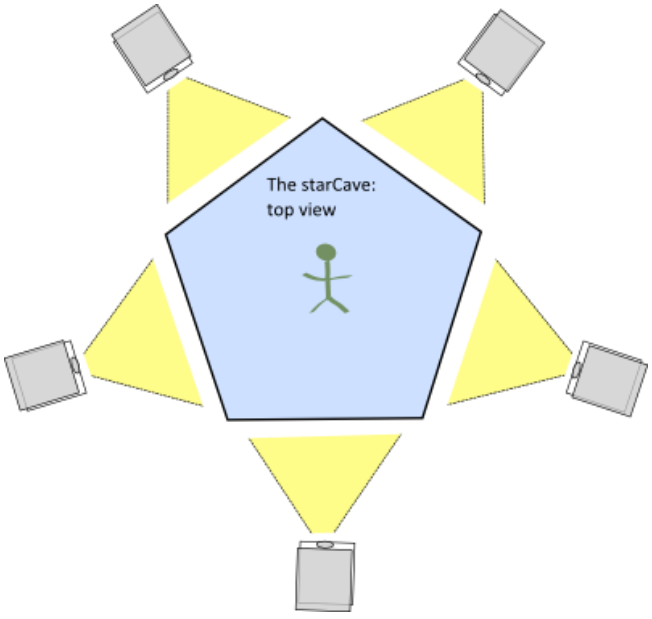


Figure 1. Simplified top view of the StarCAVE. Each wall has three screens, with two projectors driving each screen. The cave has a total of 15 screens and 30 projectors.

Hot Spots. A bright spot appears on each screen in a unique location determined by the viewer's position, the screen's position, and the projector's position. The image is brightest at the hot spot, with brightness decreasing outwards. Because the hot spots are in different locations on each screen, there are discontinuities in brightness at the edges where screens meet, making the effect more noticeable in a tiled display configuration than a single screen configuration.

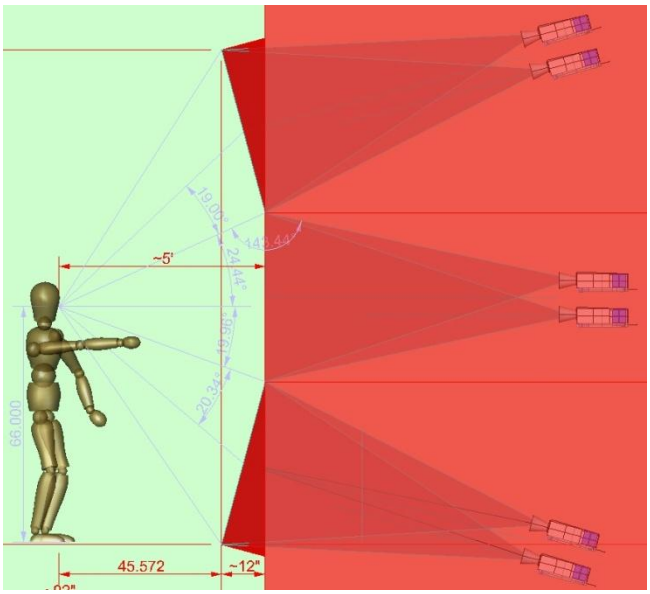


Figure 2. This CAD model of the cave shows a side view of one of the walls. The projectors are located behind the screens, so the position of the hotspot changes depending on where the user is standing in the cave. Image courtesy of Greg Dawe.

The hot spotting problem has been around for a long time, most notably in DLP (Digital Light Projection) televisions, which cope with the problem using Fresnel sheets (Takahashi). For the StarCAVE, using Fresnel sheets was prohibitively expense because of the custom nature of the system, and the required screen size and resolution (DeFanti).

In the cave, we have a critical piece of information not available to the makers of rear projection televisions - the viewer's position in 3D space. This allows us to compensate for the viewing angle-dependent hot spotting effect in a post processing step in software. The idea is to draw an inverse hotspot as the last stage

in the rendering cycle so the image appears homogenous to the viewer when displayed. We implemented this strategy in a Covise plugin, with the result that image quality in the cave is qualitatively improved. Two key requirements for the implementation were to seamlessly integrate with other Covise applications and to not adversely affect performance. Our implementation meets both of these requirements. Frame rate in typical Covise applications is reduced by 1-2 fps, while brightness deviates over a range of 0.1, as opposed to 0.4 without mitigation. This paves the way for acceptance into the Covise codebase and adoption by others.

2. Strategy

To achieve the end goal of the viewer perceiving a homogenous image, we compensate for the hot spot effect in software. After the application has rendered its scene to the frame buffer, we modulate the brightness of the image in the inverse profile of the hot spotting effect. The idea is, when the image is displayed, the two modifications cancel each other out and the user sees a homogeneous image. To effectively compensate for the hot spotting effect, we need to characterize it analytically and empirically. Based on this characterization, we will design a correcting function. Finally, we will implement the correcting function as a post processing step in the OpenGL rendering pipeline. We will use a GLSL fragment shader to perform the computationally expensive work of modulating the brightness of each pixel in the frame buffer.

3. Characterization of Hot Spots

The context that most of us are used to seeing a projector is in a movie theater, where the projector is at the back of the theater, in front of the screen. It shines light on the screen, and it bounces off into our eyes. A good movie screen is opaque and dispersive. Now, imagine the projector is no longer behind you in the back of the theater, but in front of you, behind the screen. If the screen is opaque, we see nothing. If the screen is completely transparent like a pane of glass, we don't see an image; we are looking straight into the projector's lens and are blinded. In order for us to see an image, the screen must be translucent - it must pass some light, but unlike a clear pane of glass must disperse some light. The brightness of the image is not uniform, however. Imagine a line from your eyes to the projector's lens, and think of the screen as a plane. The image on the screen appears brightest where the line intersects with the plane. This point is the hot spot. As we walk around the theater, the location of the hotspot on the screen changes because the line from our eyes to the projector intersects with the screen in a different place.

Hot spots occur because the screens are rear-projected, and because the screens are partially dispersive and partially transmissive. For an explanation of why hot spotting occurs, we look to theory of light transmission through diffuse media [Eliyahu]. The intensity of diffuse transmission through random media is given by

$$I = K(\frac{1}{\delta})S_T(\theta)[Re(\Gamma + \delta)\cos\alpha\cos\theta - Re(\Gamma' + \delta')\sin\alpha\sin\theta]$$

where K is an arbitrary constant, and $S_T(\theta)$ is the intensity of the scalar field in transmission

$$S_T(\theta) = \frac{\Delta\cos\theta + \cos^2\theta}{1+\Delta}$$

$$\rho = depolarization\ ratio\ [0, 1.0]$$

$$\alpha = angle\ of\ incidence\ (measured\ from\ surface\ normal)$$

$$\theta = scattering\ angle\ (measured\ from\ surface\ normal)$$

$$\Gamma = \frac{1-\rho}{1+\rho}$$

$$\delta = \rho\left(\frac{1-\rho}{1+\rho}\right)$$

$$\rho' = \rho + \rho(1-\rho)$$

$$\Gamma' = \Gamma - \Gamma(1-\Gamma)$$

$$\delta' = \rho'\Gamma'$$

$$\Delta = 1.0$$

Using this model, we plot the intensity of transmitted light versus the viewing angle for several values of the depolarization ratio ρ in figure 3.

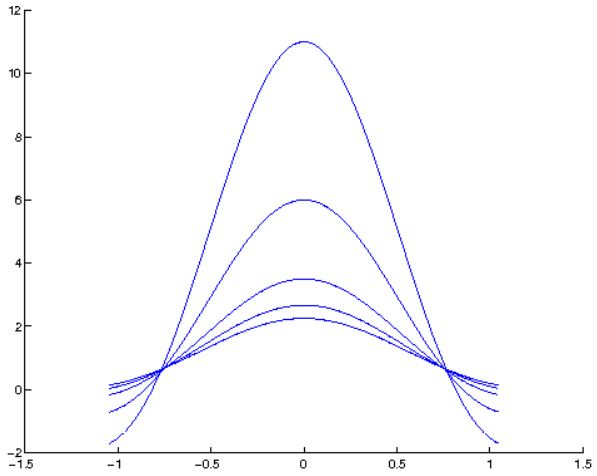


Figure 3. This graph shows the intensity of linearly polarized light transmitted through a diffuse medium for several values of the depolarization ratio. The x axis is angle of incidence in radians, the y axis is intensity. Imagine standing in front of the screen and rotating your head as you look from the left edge of the screen to the right. The intensity is greatest at 0 angle of incidence, when you are looking straight at the center of the screen. As you continue turning your head to the right, the intensity decreases and eventually crosses zero because of the incident light exceeds the critical angle. The shape of the cave is such that it would be physically challenging to look at a screen at an angle greater than the critical angle, so this is not an issue. We note that the analytically predicted angular dependence of transmitted light matches very nicely with the measured data of Figure 4.

We now measure intensity in transmission versus viewing angle in the Cave. We do this by rendering a white background, taking a picture of the screen with a digital camera (figure 4), and plotting the brightness versus viewing angle (figure 6).

Figure 4. The hot spotting effect is most visible when a flat white image is displayed. This is a picture of the screen taken with a digital SLR camera. You can see that the image is substantially darker at the edges, even though the same color (white) is being displayed everywhere.

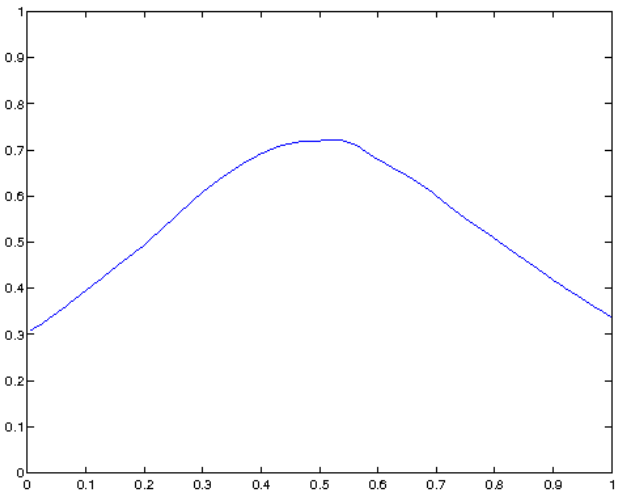


Figure 5. This graph shows brightness versus x-axis for Figure 3 above. This graph was created by holding the y position constant (in the middle of the image), and moving horizontally across the image taking samples of the brightness. Each datapoint is the average brightness of a square region of pixels. This was done to reduce noise due to optical effects inherent in digital photography.

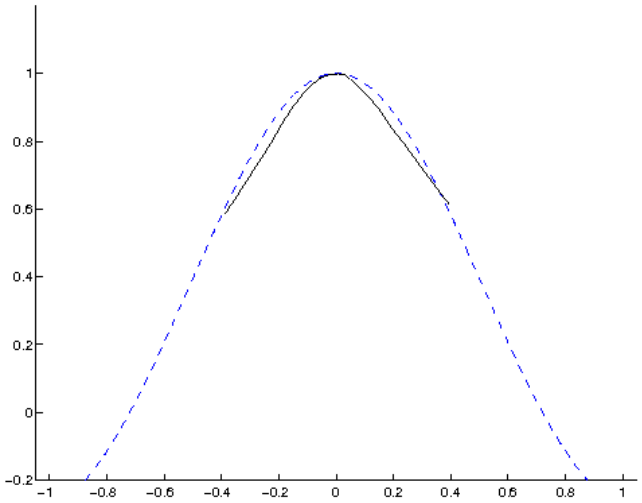
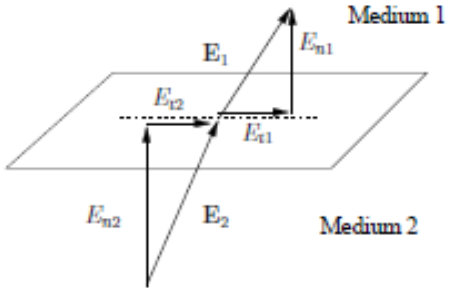


Figure 6. Intensity versus viewing angle - theory and measured data. Y-axis: intensity. X-axis: viewing angle in radians. Theoretical predicted curve (dashed blue). Experimentally measured data (solid black).

Color Dependence. From Maxwell's Equations we know that electromagnetic waves change in direction and magnitude at the boundary between two mediums in a frequency dependent manner.



The tangential and normal components of the electric field on either side of the interface are related by:

$$E_{t1} = E_{t2}$$

$$\epsilon_1(\omega)E_{n1} = \epsilon_2(\omega)E_{n2}$$

The tangential component of the electric field is continuous, and the normal component is discontinuous. The discontinuity is proportional to the ratio of the electric permittivity of the two mediums. Electric permittivity is in general a complex quantity dependent on frequency.

In the cave, color dependence is not detectable with the eye. To determine how prominent the color dependence is, we plot the intensity of red, green, and blue light versus viewing angle in figure 7 below.

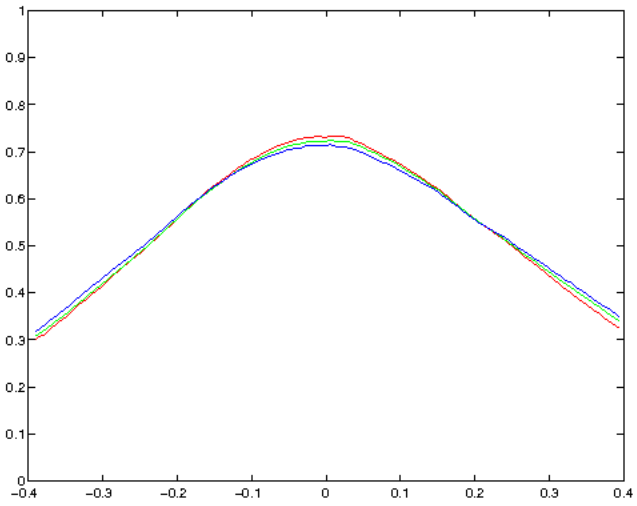


Figure 7. Intensity of red, blue, and green channels versus viewing angle. Y-axis: intensity. X-axis: viewing angle in radians.

Color dependence is present, but so slight that we will not compensate for it. Now that we have characterized the hot spotting effect theoretically and verified it experimentally, we will design our compensation scheme.

4. Design of the Correcting Function

We now must design a function such that when every pixel in the original image is multiplied by this function, the user will perceive an image of homogenous brightness at the output. Obviously, a number times its inverse is 1, so the optimal correcting function is the inverse of the solid black curve in figure 5 above [proof in Appendix A].

There are two constraints on the correcting function. First, the intermediate product of the correcting function and the original image must not exceed 1 or parts of the image will saturate and image quality will degrade. Second, the correcting function must be computable quickly on the GPU.

One simplification we can make is to write the correcting function as a function of linear position across the screen instead of angle of incidence and scattering angle. By the small angle approximation, $\tan(x) \approx x$ for small values of x. Viewing angles for a given screen in the cave range from 20° - 30°. In figure 8 below, we plot brightness versus viewing angle and brightness versus linear position on the same plot. The negligible difference between the two curves validates usages of the small angle approximation.

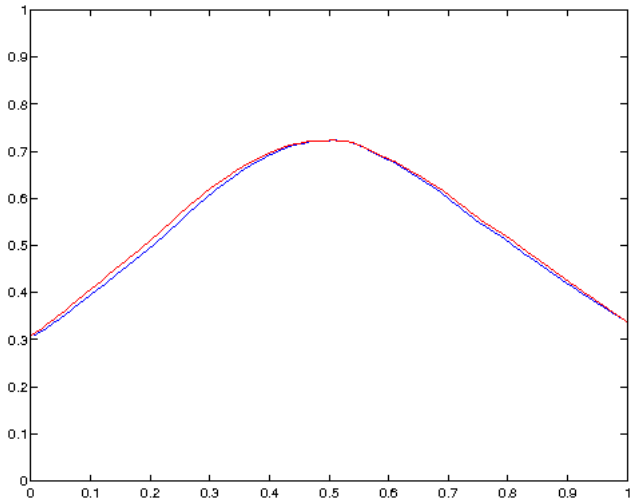


Figure 8. Brightness versus viewing and brightness versus linear position. The x-axis units have been normalized [0, 1] to fit both curves on the same plot.

To compute the correcting function quickly on the GPU, we approximate it as linear. In the figure below, the original image is 1 (constant white background), the hot spot effect is shown in solid blue, and the optimal correcting function is shown in dashed red. We notice however that the optimal correcting function is always greater than 1, and since our input is equal to 1, the product of these two functions will be greater than 1, violating the requirement that the intermediate product of the original image and the correcting function be less than or equal to 1. Therefore we must shift the correcting function downwards so it is always less than or equal to 1.

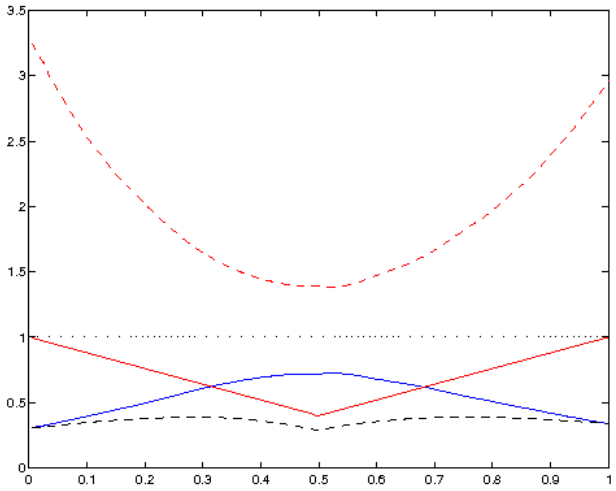


Figure 9. Y-axis: brightness, x-axis is linear position on screen, similar to figure 5. Optimal correction function (dashed red). Linearly approximated correction function (solid red). Original image (horizontal line at y = 1). Perceived image before compensation (solid blue). Predicted perceived image after compensation (dashed black).

The predicted brightness of the image after compensation is shown in dashed black. We notice several things about this curve.

- While not perfectly flat, it is much flatter than the image would be without hotspot mitigation. Testing in the cave shows that the eye cannot perceive this slight nonlinearity, and the image does appear homogenous.
- There is an upper limit on brightness that cannot be exceeded without saturating the image. The edges of the image dictate the brightest part, and the rest of the image must be normalized to these points. The difference between the blue curve and the dashed black curve represents the amount of brightness we're losing. At the center of the image, there is a 53% loss of brightness. This is bad, and we want to avoid it. The constant white background is a worst case scenario though, and we will see later that typical images displayed in the cave are dark enough that we don't have to worry about saturation. In fact, we have found that we can make the edges brighter by a factor of 1.4 ~ 1.8 without any

noticeable saturation, resulting in a brighter image than we started with.

Now we want to see how well our simplified correction function performs in the real world. The results are shown in figure 8 below.

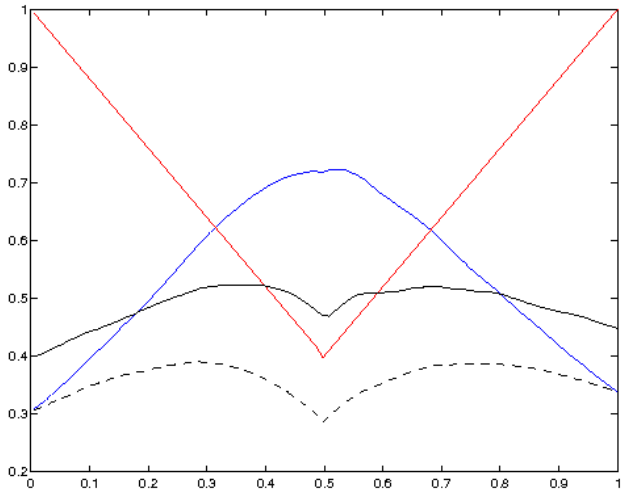


Figure 10. Y-axis: brightness. X-axis: linear position. Linear correction function (solid red). Perceived image before compensation (solid blue). Predicted perceived image after compensation (dashed black). Actual perceived image after compensation (solid black).

The first thing we notice between the theoretical and measured data is vertical displacement. This can be explained by automatic color correction and white balance by the camera. We used a Nikon D3000 digital SLR mounted on a tripod to take these pictures. We set the ISO and exposure manually, but in order to get anything to turn out we had to use automatic white balance. Automatic white balance is applied as a constant over the whole image, so the relative brightness between points remains valid. When the two curves are shifted on top of each other in figure 9, we can see the correlation much better .

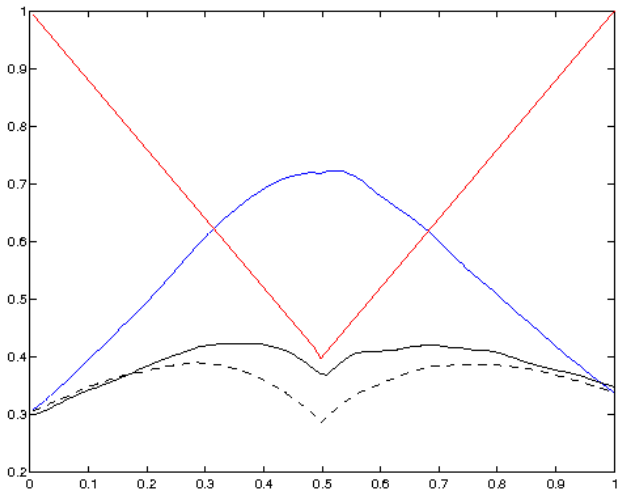


Figure 11. Y-axis: brightness. X-axis: linear position. Linear correction function (solid red). Perceived image before compensation (solid blue). Predicted perceived image after compensation (dashed black). Actual perceived image after compensation with vertical displacement (solid black).

Considering the many approximations involved and the inherent imprecision of taking a picture of the screen with the camera, we would consider this an excellent correlation between theory and the real world. The difference between the real curve and theoretical curve can be accounted for by imprecision in the measuring device - the camera. The pictures were taken in very low light, and the camera simply isn't sensitive enough to detect such subtle changes in brightness.

5. Implementation

Now that we have examined the hotspotting problem and we have a strategy of how to compensate for it, we turn our attention to the implementation. The first task is to compute the hotspot location given the viewer's position, the screen position and orientation, and the projector position. One's first instinct would be to set up a line-plane intersection formula, but there is a more elegant way. The viewer's position is encoded in the OpenGL View matrix and the screen's position and orientation are encoded in the OpenGL projection matrix. The view matrix takes object coordinates to world coordinates and the projection matrix takes world coordinates to screen coordinates.

To calculate the position of the hotspot in screen coordinates, we perform a series of matrix transformations on the projector coordinates:

$$hotspot = projMatrix * viewMatrix * projCoords$$

where projMatrix and viewMatrix are 4x4 matrices and projCoords is a 1x4 vector in homogeneous coordinates. The result is a 4-component vector where the first two components are the x, y position of the hot spot in screen coordinates. The z and w components can be ignored. Now that we know how to compensate for the hotspot and how to calculate its position, we must apply the compensation to the image.

5.1 Alternative Implementations

We researched and implemented several techniques of applying the correction function to the original image before we arrived at the final implementation. We will now discuss the pros and cons of these implementations.

Always-on Shader. The idea behind this implementation was to enable a shader and leave it on, so that it processed all fragments coming from all plugins. The potential upside of this technique is that it would be very fast and simple. By intercepting each pixel on its way to the framebuffer, there would be no need for an additional post processing step. This approach has several critical downsides, however. First, when you enable a shader, it replaces the fixed-functionality shader. Since the fixed-functionality shader is responsible for built-in OpenGL functionality including texturing, lighting, and fog, one would have to implement all of these features in the custom shader. We wrote a simple shader that operated on the incoming `gl_FragColor`, but this approach was highly inadequate because it did not take care of texture mapping. Elementary shapes with solid colors turned out fine, but, for example, it made the Covise menu unintelligible because it did not do texture mapping. Since in OpenGL there is exactly one shader enabled at any given time, this would preclude plugins from using their own shaders - clearly an unworkable requirement. Also, this approach is not modular, and is fragile because it could break if other plugins modify the state of the rendering pipeline.

Blending. The strategy of this implementation is to draw the hotspot pattern on a screen-aligned rectangle, then blend this rectangle with the current framebuffer using `GL_FUNC_REVERSE_SUBTRACT`. With this approach you can subtract (or add) a different value to every pixel in the framebuffer. The benefits of this technique are there is no copying involved, it uses addition and subtraction which is faster than multiplication, it does not interfere with other plugins, and it is implemented entirely in OSG, which eases complexity by leveraging OSG's state management facilities. The critical downside of this technique is that it uses addition and subtraction. Consider a pixel in the framebuffer (R_s, G_s, B_s). We want to reduce (or increase) the brightness of this pixel. So, we subtract a constant from all three components. $(R_s - c, G_s - c, B_s - c)$. We have reduced the brightness of the pixel, but we have changed its hue, distorting the color. This technique has a tendency to saturate the image earlier than scaling. Visual results were very poor, so this implementation had to be scrapped.

Two other techniques we researched, but did not test, were using the **accumulation** buffer, and using **multitexturing**. The accumulation buffer suffers from the same problem as blending in that it does not support multiplication. Multitexturing lets you combine textures in many different ways, including modulation (multiplication). The logical steps in a multitexturing approach would be as follows:

1. Copy the framebuffer to a 2D texture
2. Draw the hotspot pattern to an auxillary buffer, then copy it to a 2D texture. This step could be performed only once during initialization.
3. Map the texture from 1) to a screen aligned rectangle

4. Map the texture from 2) to the same rectangle, specifying the `GL_MODULATE` function as the texture combiner.

The benefit of using this approach would be that you could use built-in OpenGL functionality without having to write a shader. The downside is that you lose the generality and flexibility of the shader. This technique is likely to have worse performance than the custom shader technique because the computational work of multiplying each pixel in the first texture by each pixel in the second is still being performed by the fragment shader, but with the additional overhead that comes with the generality of the built-in fragment shader.

5.2 Final Implementation

Since we want our post processing code to be the last thing in the rendering cycle, we do it right before the front and back buffers are swapped. In Covise, this corresponds to the `CoVRPlugin::preSwapBuffers()` callback. The logical steps in post-processing are:

1. Copy the back color buffer to a 2D texture.
2. Enable the fragment shader with the following data as uniforms
    a. The texture containing the scene created in 1). (`tex`)
    b. The screen coordinates of the hotspot (`hotspot`)
    c. The distance from the hotspot to the point on the screen farthest from the hotspot (`max_dist`). The farthest point from the hotspot will be the darkest point on the screen, so this is the point we wish to normalize against.
3. Draw a screen-aligned rectangle with the texture mapped to it.

The rest of the work goes on in the fragment shader. The fragment shader receives as input the rasterized scene, the hotspot location in screen coordinates, and the distance against which to normalize. The logical steps in the shader are:

1. Compute distance from the current fragment coordinate to the hotspot.
2. Compute the correction factor by the ratio of this distance to the longest distance. When the fragment coordinate is equal to the hotspot location, the ratio is 0. When the fragment coordinate is the farthest point from the hotspot, the ratio is 1.
3. Get the RGB pixel values for the current fragment coordinate by doing a lookup in the texture. Since the rectangle being shaded is screen-aligned, the texture coordinate is the current pixel position.
4. Scale the RGB pixel values by the correction factor calculated in 2)

The largest bottleneck in this implementation is copying the screen buffer to the texture. Each pixel has to travel through the rendering pipeline twice. A possible way to avoid the copy operation would be to use Nvidia's non-standardized Framebuffer Object extension. The scene could be rendered directly to a texture, eliminating the need to copy the frame buffer to a texture.

6. Visual Results

The hotspot mitigation plugin produces a noticeable improvement in brightness homogeneity in the cave. The largest improvement comes from better matching of brightness at the edges where screens meet. Since the algorithm compensates for the fact that the hot spot location is dependent on the position of the user in the cave, the image remains homogenous as the user walks around. Most images displayed in the cave are somewhat darker than the worst-case white background, so we can actually increase the brightness of the image while mitigating the hotspot effect at the same time. With bright images there is the danger of saturation, but this can easily be fixed by reducing the gain, which is configurable at runtime from the Covise UI. The user can adjust the

gain/attenuation while standing in the cave until the image looks homogenous. See Appendix A for visual results.

Figure 12. Implementation diagram.

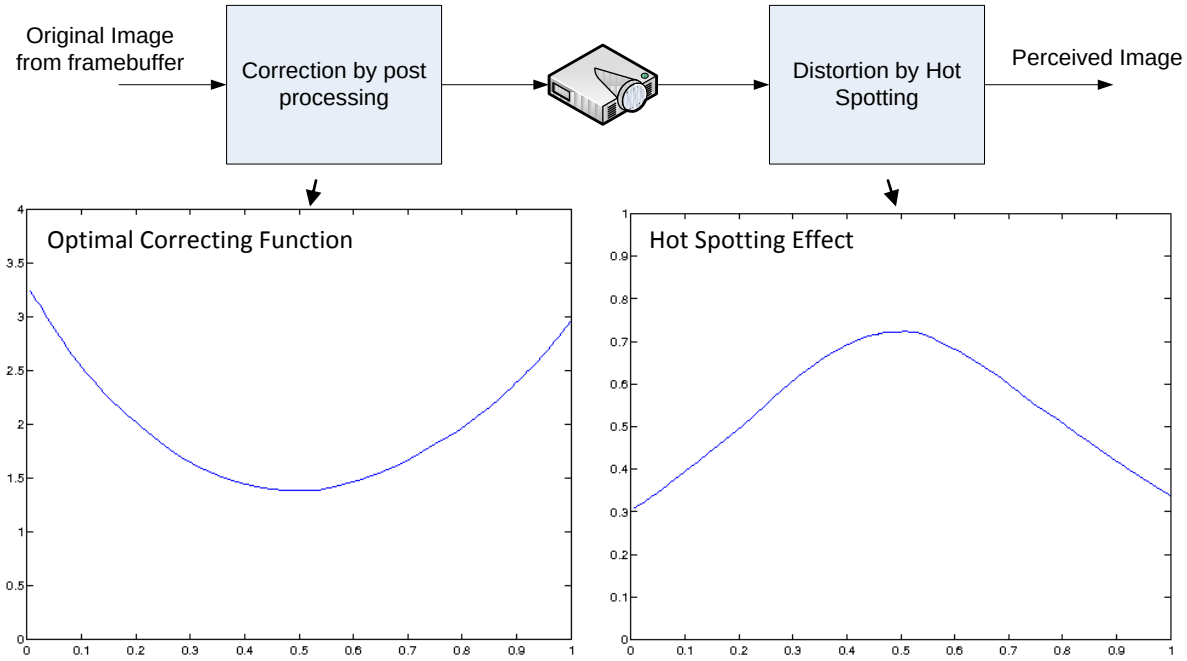
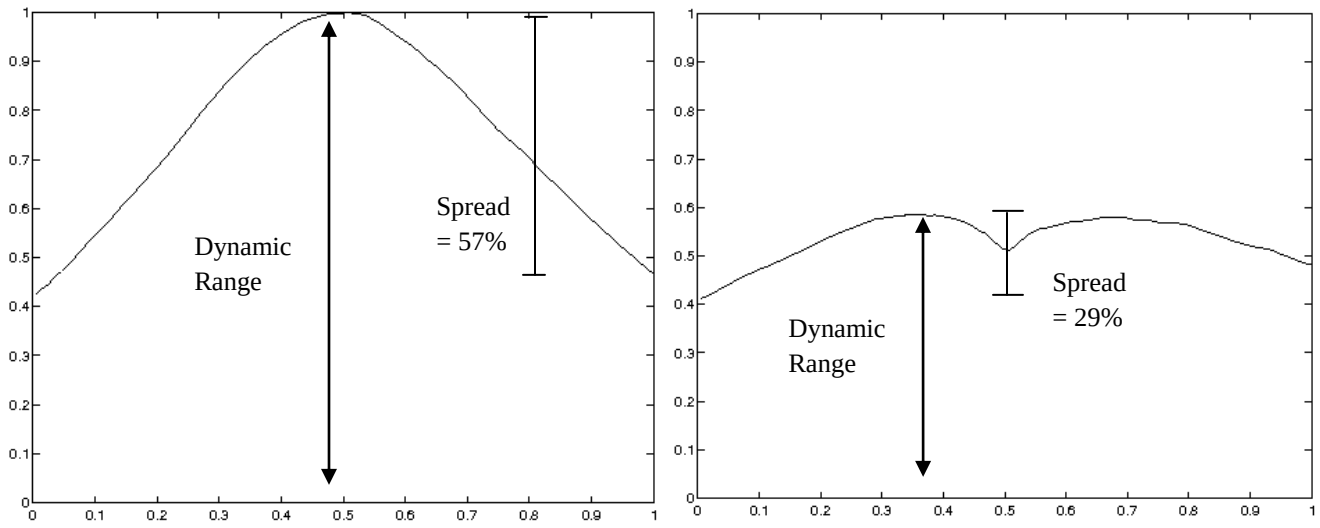


Figure 13. Quantification of results. We will use spread as a percentage of dynamic range to quantify the homogeneity. Ideal is 0%. Left: before correction: 57%. Right: after correction: 29%.



7. Performance

Performance is an important requirement. The cave is used for computationally intensive scientific visualization, so we don't want to tax the CPU and GPU any more than we have to. The measurable performance benchmark is frames per second. We hypothesize the algorithm takes constant time per frame. We measure the frame rate of

several applications with and without compensation, and calculate how much time the algorithm adds to the rendering of each frame.

Table 1. Frame rate of typical cave applications with and without hot spot mitigation.

Plugin	compensation off (fps)	time per frame	compensation on (fps)	time per frame	difference (ms)
None	148.4	6.74	125	8.00	1.2615
PanoView360	44.7	22.37	43.4	23.04	0.6701
Calit2 Model	32.5	30.77	31.6	31.65	0.8763
PDB Viewer (hemoglobin)	40.2	24.88	37.6	26.60	1.7201
				Average (ms)	**1.132**

The algorithm adds about 1ms to each frame. This translates into a reduction of about 1-3 fps in typical cave applications. This is an acceptable hit as long as the frame rate stays above 30 fps. Below 30 fps, the image looks choppy.

8. Related Work

US Patent "Graphics System having a super-sampled sample buffer with hot spot correction" outlines an architecture for a hardware graphics pipeline and its potential applications. One of the applications described is hot spot correction. Intensity scaling values would be loaded into a buffer and multiplied per-pixel against the frame buffer to do brightness normalization, and could be updated as the user moves around. This is similar to the multitexturing approach described in section 5.1 Alternative Implementations. The authors describe their idea but do not present an implementation or results. There are a number of standalone hardware devices that apply a ramp function to the edges of images for blending in tiled display walls [Inova]. Nvidia holds a patent for Per-Pixel Output Luminosity Compensation [USPatent7336277], where the brightness of the image could be modulated per-pixel by texture blending. The patent mentions using the technique to correct for keystone distortion and edge smoothing, but does not provide a mechanism to update the compensation in real time as the user moves around.

In "LAM: Luminance Attenuation Map for Photometric Uniformity in Projection Based Displays," Aditi Majumder implements hot spot mitigation by scaling the brightness of the image by an attenuation map generated by taking a picture of the screen. However, it does not provide a mechanism to update the map in real time based on the position of the user. The main contribution of this paper is to use head tracking to dynamically compute the correction factors in a GLSL shader, which is portable over any hardware that supports OpenGL. The technique is applied as a post-processing step in the OpenGL pipeline and does not require modifications to existing applications. This paper provides a concrete implementation on commercially available hardware with good performance and presents results for a virtual reality environment.

9. Conclusions and Further Work

The StarCAVE presents a unique opportunity to combat the common problem of hot spotting because we know the position of the user at all times and we have high performance, programmable graphics

hardware. Our implementation produces noticeably smoother images, and is being used daily by researchers in the Cave.

The greatest area for improvement is in the correcting function. Right now it is implemented as a simple linear falloff, but from the empirical data and analytical models of light transmission we see the intensity profile is much smoother. A possible technique for basing the correcting function off the empirical data would be to load the empirical data into a 1D texture, then define an appropriate function to map distance values to indices in the texture, and use the texture as a lookup table. Also, we see from the analytical model of light transmission that the intensity of transmitted light depends on the angle of incidence and the scattering angle. Therefore, instead of the correction factor being a function of distance from the hotspot to the current pixel, it should be a function of the angle of incidence from the projector to the current pixel, and from the current pixel to the viewer's position. We believe it is possible to implement both of these improvements without affecting performance too much. Using two variables (angle of incidence and angle of scattering) to determine the correction factor would require a two dimensional lookup table, and one would have to strike a balance between accuracy and texture memory consumption. Future implementations could use the Nvidia Framebuffer Objects extension to render the scene directly to a texture, eliminating the performance-limiting copy-to-texture step.

10. Acknowledgements

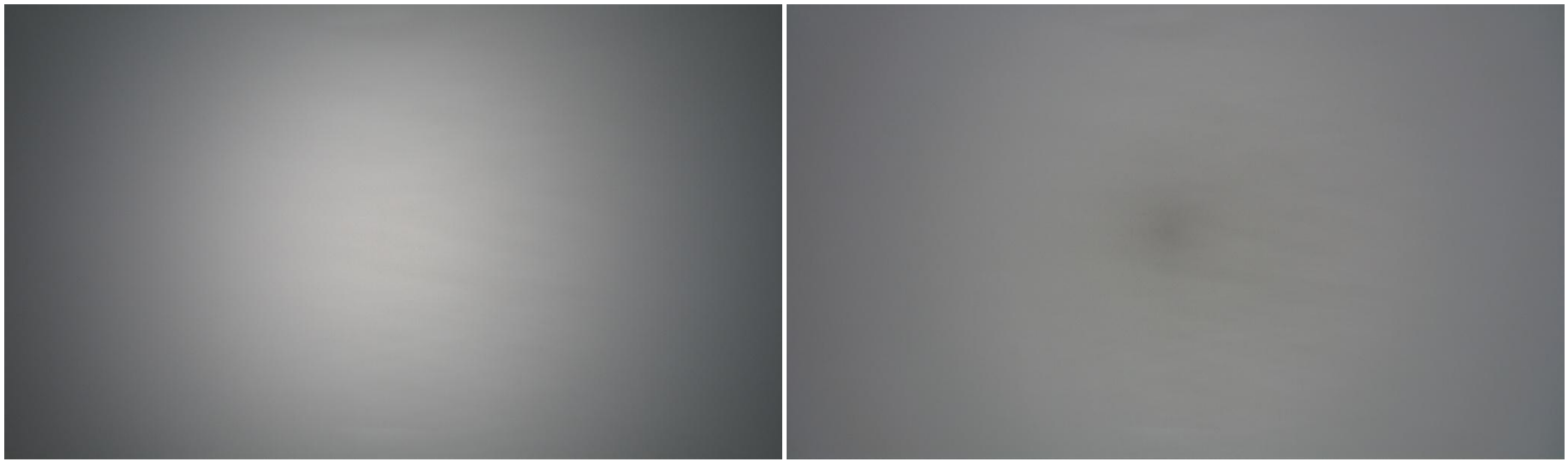
I would like to thank Jurgen Schulze for advising me on the project, Tom DeFanti and Kaust for funding the project, Andrew Prudeholme for devising the hot spot position calculation method, Greg Dawe for figure 2 and CAD models of the cave, and Calit2 for providing me the opportunity to do this project. This publication is based in part on work supported by Award No. US 2008-107, made by King Abdullah University of Science and Technology (KAUST).

References

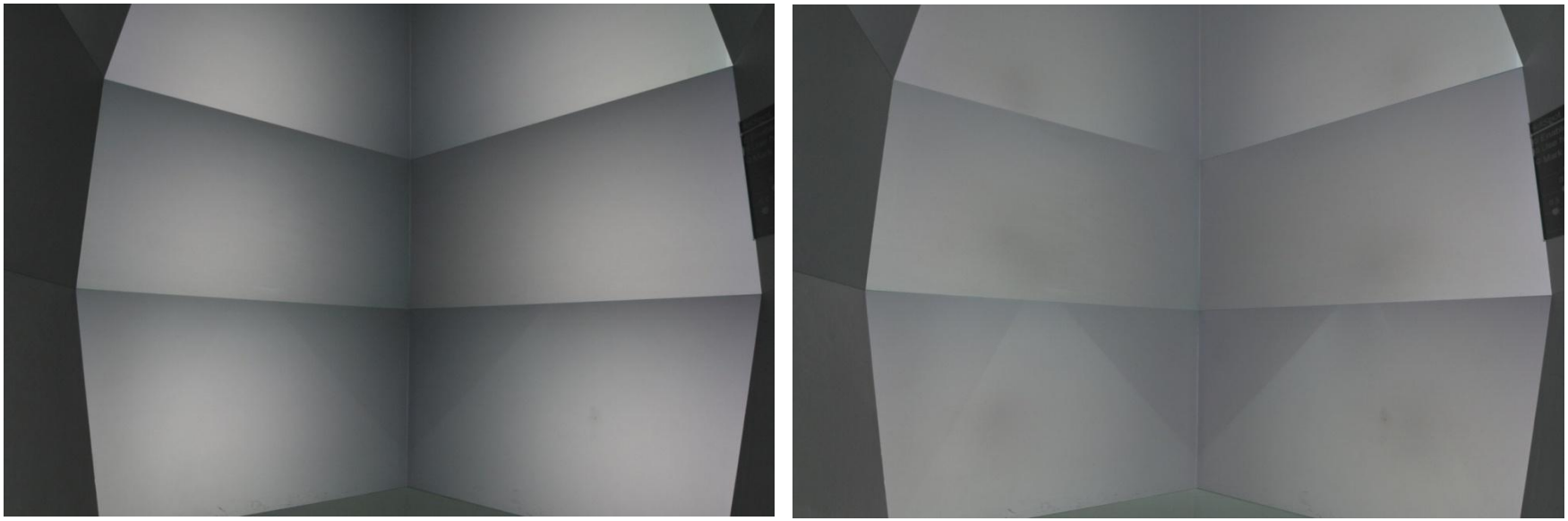
1. A. Theodorou, "Image post-processing with shaders" Retrieved April 3, 2009, from http://encelo.netsons.org/blog/2008/03/13/image-post-processing-with-shaders/
2. Anonymous, "Real-Time Fog using Post-processing in OpenGL," Retrieved April 3, 2009, from http://cs.gmu.edu/~jchen/cs662/fog.pdf
3. Covise. http://www.hlrs.de/organization/av/vis/covise/
4. D. Eliyahu, M. Rosenbluh, I. Freund, "Angular intensity and polarization dependence of diffuse transmission through random media," J. Opt. Soc. Am. A 10, 477-491 (1993)
5. D. Shreiner et al, (2008). *OpenGL Programming Guide* (6th ed.). New York: Addison-Wesley.
6. Immersive Visualization Lab wiki. http://ivl.calit2.net/wiki/index.php/COVISE_and_OpenCOVER_support
7. S. Green, "The OpenGL Framebuffer Object Extension," Retrieved April 3, 2009, from http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf
8. T. DeFanti et al, "The StarCAVE, a third-generation CAVE and virtual reality OptIPortal," Future Generation Computer Systems, Volume 25, Issue 2, February 2009, Pages 169-178, ISSN 0167-739X, DOI: 10.1016/j.future.2008.07.015. (http://www.sciencedirect.com/science/article/B6V06-4T7F5J5-1/2/03c791cd37872aae20833196e41ec097)
9. Takahashi, K. (2000). "Fresnel lens sheet for rear projection screen." U.S. Patent No. 6052226.
10. Stone, M. C. 2001. Color and Brightness Appearance Issues in Tiled Displays. IEEE Computer Graphics Applications 21, 5 (Sep. 2001), 58-66.

Appendix A. Visual Results

Original image (left), distorted image (center), corrected image (right). While subtle, the improvement is most noticeable when standing in the cave and enabling/disabling the hotspot plugin.



Distorted image (left), corrected image (right). The effect is easiest to see when displaying a solid white background.



StarCAVE without hot spot mitigation (left). Notice the discontinuities at the seams. StarCAVE with mitigation (right).

Appendix B. Calculating the Optimal Correcting Function

To put things on a sound mathematical footing, let's first examine the inputs, outputs, and functions involved.

$$let\ f(x,y) = \ original\ image, \qquad g(x,y) = \ distorting\ function$$

$$let\ h(x,y) = \ perceived\ distorted\ image, \qquad j(x,y) = \ correcting\ function$$

$$let\ k(x,y) = \ perceived\ corrected\ image$$

We know the original image $f(x,y)$ because it's the thing we're trying to render, and we know the perceived distorted image $g(x,y)$ because we can take a picture of the screen with a camera. We want to find $j(x,y)$, the correcting function. We can write the perceived distorted image as a product of the original image and the distorting function.

$$h(x,y) = \ f(x,y)g(x,y)$$

We can write the perceived corrected image as:

$$k(x,y) = f(x,y)j(x,y)g(x,y)$$

We want to find the correcting function $j(x,y)$ such that the perceived corrected image is equal to the original image, $k(x,y) = \ f(x,y)$. Substituting and solving yields

$$j(x,y) = \frac{1}{g(x,y)}$$

It makes sense that the correcting function is the inverse of the distorting function. Now we have to find the distorting function $g(x,y)$. To do this, we set the input $f(x,y) = 1$ and the expression for $g(x,y)$ becomes

$$g(x,y) = \frac{h(x,y)}{f(x,y)} = \frac{h(x,y)}{1} = h(x,y)$$

So to find the correcting function $j(x,y)$, we set the input $f(x,y)$ equal to 1 (by rendering a white background) and take a picture of the screen. To extract the 1-dimensional brightness profile, we hold the y-coordinate constant and sweep over the x values. The correcting function is the inverse of this curve.

$$j(x,y) = \frac{1}{h(x,y)}$$

Appendix C. Fragment Shader Code Listing

```
//max bound for the alpha value
uniform float max_alpha;
//minimum bound for the alpha value
uniform float min_alpha;
//texture that contains the framebuffer
uniform sampler2D tex;
//location of the hotspot in unit coordinates
uniform vec2 hotspot;
//longest distance from hotspot to other point
uniform float max_dist;
//constant scale factor to do inter-screen normalization
```

```
uniform vec4 rgba_scale;

void main()
{
	float dist = distance(gl_FragCoord, hotspot);
	float ratio = dist / max_dist;
	float j = ratio * (max_alpha - min_alpha) + min_alpha;
	vec4 texel = texture2D(tex, gl_TexCoord[0].st);

	//we don't want to modify the alpha channel across the hotspot normalization
	vec4 new_frag_color = texel * j;
	new_frag_color.a = texel.a;
	gl_FragColor = new_frag_color * rgba_scale;
}
```