CSE 190 Discussion 6

PA3: CAVE Simulator

Agenda

- PA3 Recap
- FrameBuffer Usage
- Projection for CAVE Screen
  - Math
  - Implementation
- Viewport Switch
  - Debug Wireframe Mode
- Extra Credit

PA3 Recap

- Project 3 Due Date: May 17th 2pm (This Friday!)
  - If you have scheduling conflicts, let us know
- CAVE Simulator
  - Render scene to different textures
  - Use these to texture 3 different quads to emulate screens
- Other features
  - Switch Viewpoint to controller
  - Freeze/unfreeze the Viewpoint
  - Wireframe debug mode

Framebuffer Usage

Framebuffer Usage

- Framebuffer Recap:
  - A container to hold the stuff you want to draw (attachments)
  - Attachment types:
    - Color -> texture
    - Depth -> renderbuffer
  - Allows up to render scene to a texture
- See last weeks Discussion slides for more details
- Can also look at InitGL() in RiftApp class for an example

Framebuffer Usage

- Initialize framebuffer for each of your quads/screens
  - Generate the framebuffer
  - Initialize the texture and attach to the framebuffer
  - Initialize the renderbuffer and attach to the framebuffer
- Note:
  - Attachment type for texture and renderbuffer

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

Framebuffer Usage

- When drawing (for each eye):
  - Bind to your framebuffer
  - Clear background and color/depth bits
  - Draw stuff that you want to render to your texture
  - Unbind your framebuffer
  - Render your CAVE screen (the quad)

```
// bind our framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

// render scene

// bind the default framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// render quads with the texture
```

Framebuffer Usage

- Note:
  - GLFW has its default framebuffer to draw onto window
  - So whenever you do:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

  - You are binding back to the default frame buffer.

- unbinding your framebuffer:
  - Don’t unbind to the default framebuffer!
  - Instead, unbind to **_fbo**, which is the framebuffer used by the minimal example to render to your HMD

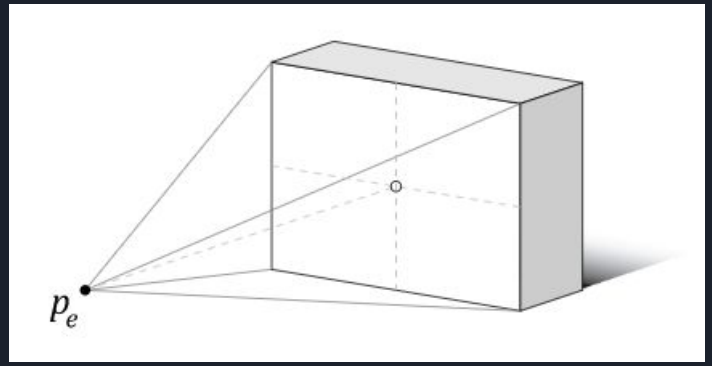
Rendering to the Texture

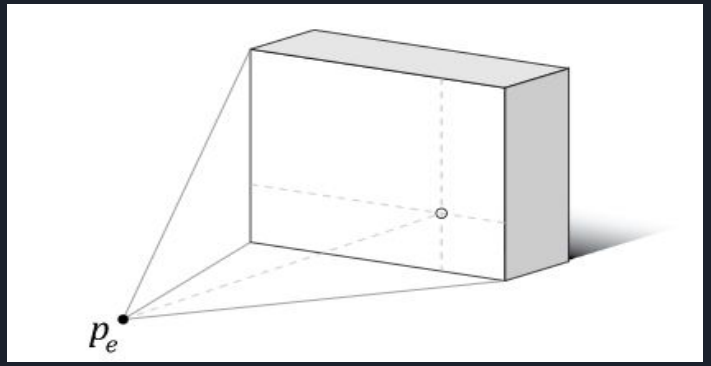
- NOTE:
  - The texture width and height need to match what is used in glViewport() from RiftApp class
- So when you are rendering your scene:
  - Save the glViewport parameters before rendering to FB
  - Set the glViewport to match the texture size
  - Render the scene onto the texture
  - Reset the viewport
  - Render the Cave

Projection Matrix for CAVE Screens

Projection Matrix for CAVE Screens

- Reminder a typical projective matrix assumes we are right in front of the screen
- We need to be able to be off-center





Projection Matrix for CAVE Screens

- Review of what we did to get the projection matrix

$$P' = PM^T T$$

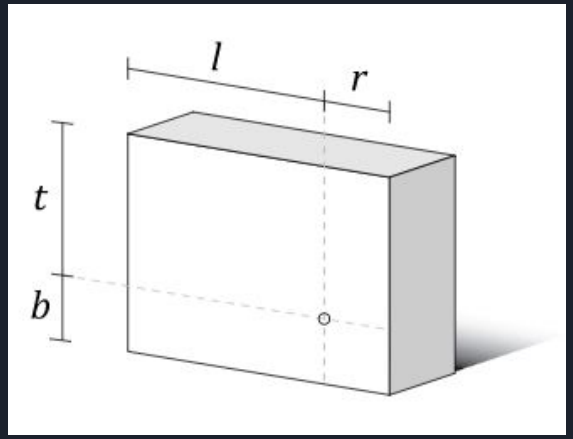
- This gives you the projection matrix (P′) for each screen

Projection Matrix for CAVE Screens - P

- OpenGL gives us a wonderful function:

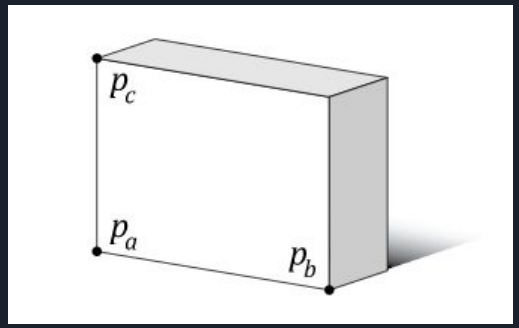
```
glm::mat4 P = glm::frustum(l, r, b, t, near, far);
```

- We need to compute l, r, b, t.
  - Last discussion gives step-by-step explanation of how to compute these frustum parameters from Pa, Pb, & Pc
- Near and far define the near/far clipping plane
  - Depends on how you want to clip user's view



Projection Matrix for CAVE Screens - P

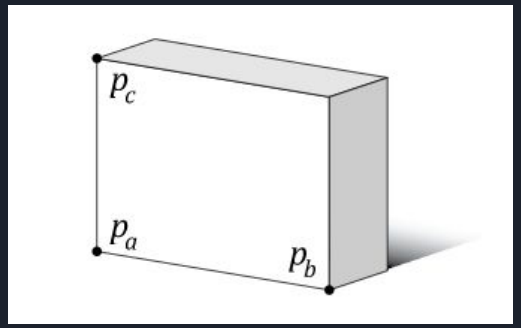
- Since each screen has different translation/rotation, the resultant projection matrix should be different.
- Where does the difference come from?




```
PA glm::vec3(-1.0f, -1.0f, 0.0f);
PB glm::vec3(1.0f, -1.0f, 0.0f);
PC glm::vec3(-1.0f, 1.0f, 0.0f);
```

Projection Matrix for CAVE Screens - P

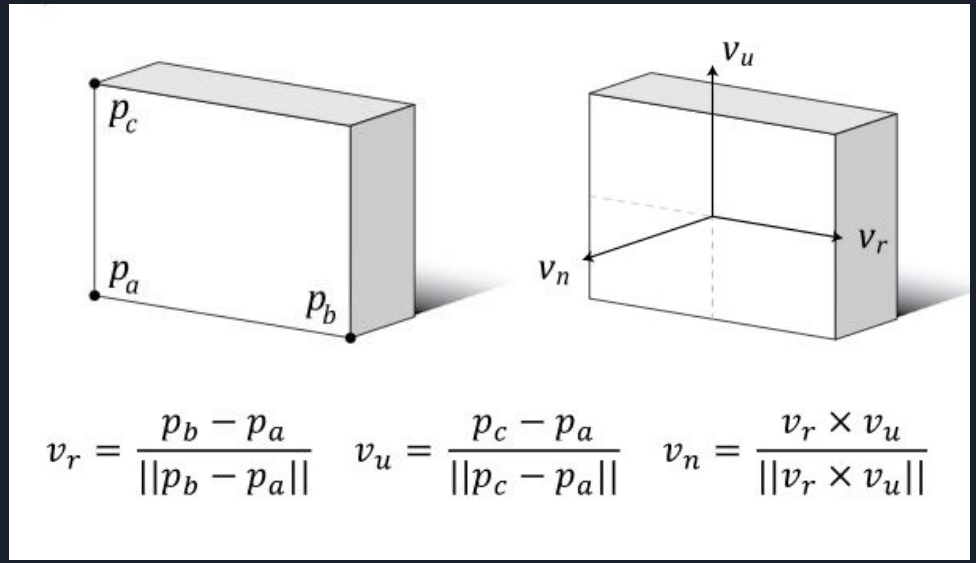
- To get correct Pa, Pb, Pc for each screen:
  - Pa = model_matrix * glm::vec4(PA.x, PA.y, PA.z, 1.0f);
  - Same for Pb and Pc




```
PA glm::vec3(-1.0f, -1.0f, 0.0f);
PB glm::vec3(1.0f, -1.0f, 0.0f);
PC glm::vec3(-1.0f, 1.0f, 0.0f);
```

Projection Matrix for CAVE Screens - P

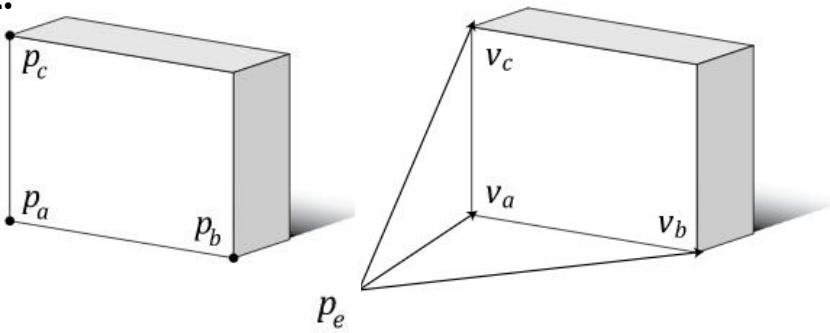
1. Compute basis vectors for screen space



Projection Matrix for CAVE Screens - P

2. Calculate vectors from eye position to the screen corners
3. Calculate distance from eye position to the screen space origin

2.

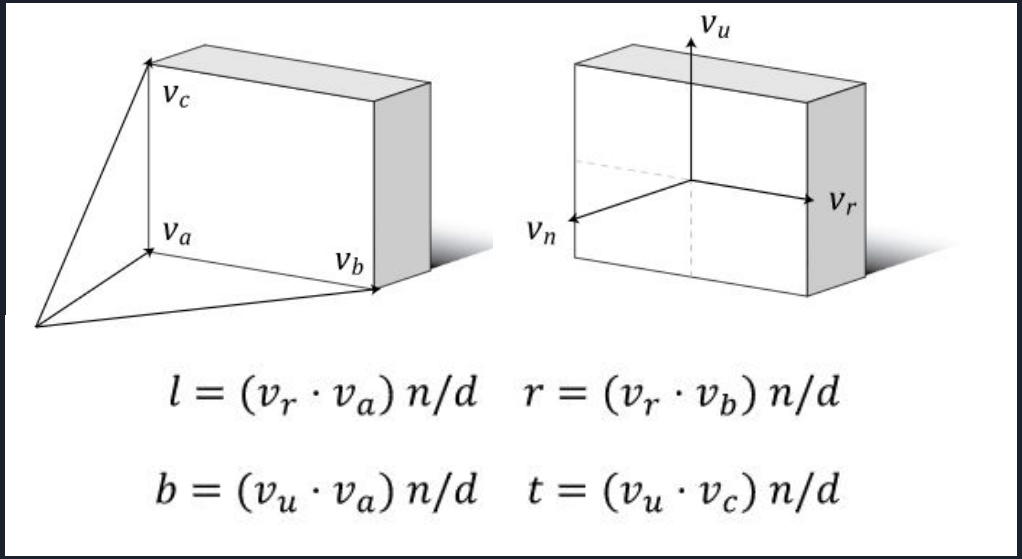


$v_a = p_a - p_e$ $v_b = p_b - p_e$ $v_c = p_c - p_e$

3. $d = -(v_n \cdot v_a)$

Projection Matrix for CAVE Screens - P

4. Calculate the frustum extents at the near plane



Projection Matrix for CAVE Screens - P

- Now that we have our frustum parameters can calculate the P matrix by simply calling:

```
glm::mat4 P = glm::frustum(l, r, b, t, near, far);
```

$$P' = PM^T T$$

Projection Matrix for CAVE Screens - M^T

- Review of the formula for M^T

$$M^T = \begin{bmatrix} v_{rx} & v_{ry} & v_{rz} & 0 \\ v_{ux} & v_{uy} & v_{uz} & 0 \\ v_{nx} & v_{ny} & v_{nz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- We already calculated v_r, v_u, and v_n
- So all we need to do is create a mat4 for M^T and plug those vectors in!

```
glm::mat4 M = glm::mat4(vr, vu, vn, glm::vec4(0, 0, 0, 1));
```

Projection Matrix for CAVE Screens - T

- Review of the formula for T

$$T = \begin{bmatrix} 1 & 0 & 0 & -p_{ex} \\ 0 & 1 & 0 & -p_{ey} \\ 0 & 0 & 1 & -p_{ez} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Reminder:
  - p_e = eye position
  - T = translation matrix by $-p_e$

```
glm::mat4 T = glm::translate(glm::mat4(1.0f), -p_e);
```

Projection Matrix for CAVE Screens

- Now take a look at the formula again

$$P' = PM^T T$$

- Note:
  - P' is the actual projection that we want to return, NOT P
- What's the next step when I get the projection?
  - Draw your scene to your framebuffer R
  - Render them onto the texture of your screen

Viewport Switch

Viewport Switch

- Currently your View position is coming from the Position and Orientation of your HMD
- Need to be able to switch the view position to your right controller
  - This is simulating being a spectator in a CAVE with another person wearing the head tracker
  - Your controller is acting as that person's head

Viewport Switch

- Note:
  - When your rotate your head, the scene on the screens should not rotate
  - So when you rotate your controller in this mode, the scene should not rotate
  - You still have two “eyes” on your controller in this mode

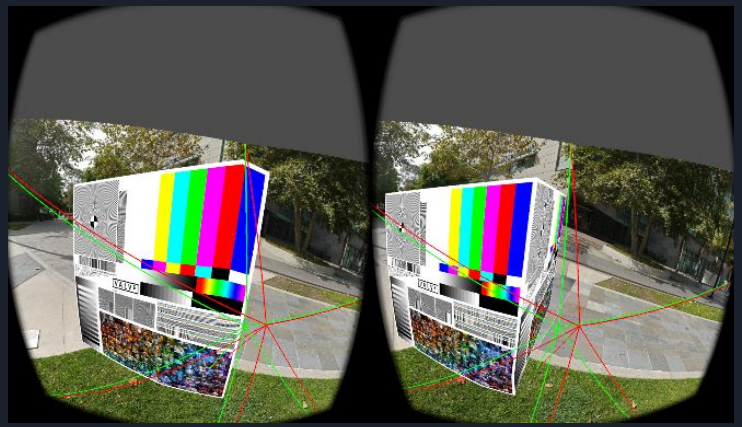
Viewport Switch

- Although rotation is not reflected in the scene, you are still expected to see some changes while rotating controller:
  - Controller’s forward is perpendicular to your head forward
    - The image becomes mono
  - Controller’s forward is in reverse direction
    - The image is inverted stereo

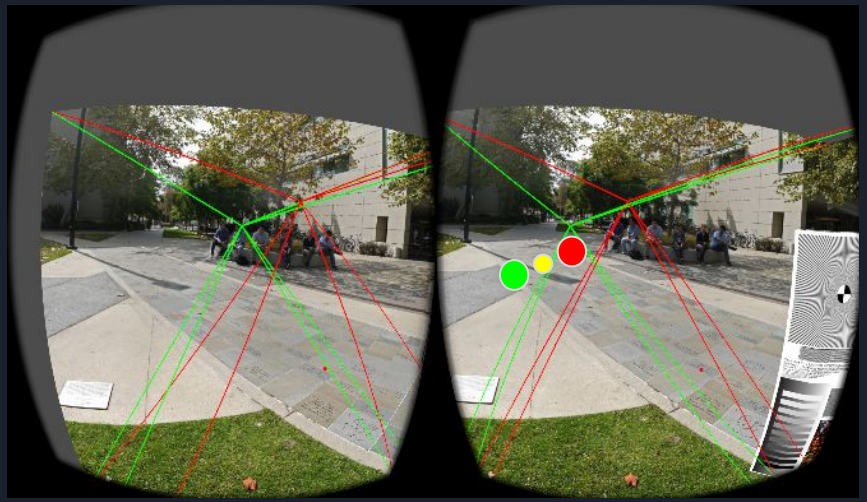
Viewport Switch- Debug Wireframe Mode

Viewport Switch- Debug Wireframe Mode

- The point of this is to:
  - Visualize the two “eye positions” of your controller
  - Visualize the six pyramids
    - Green pyramids for left eye, red for right eye



Viewport Switch- Debug Wireframe Mode



Green Dot: Left Eye Position (On the controller)

Red Dot: Right Eye Position (On the controller)

Yellow Dot: Controller's Position

P.S. The dots are not required to be rendered

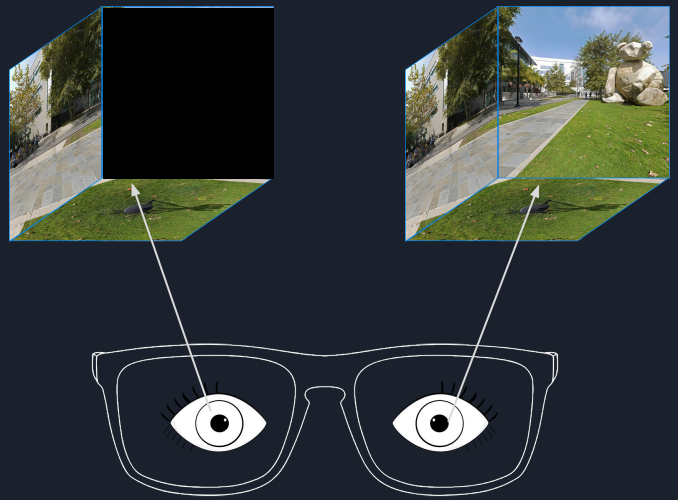
Viewport Switch- Debug Wireframe Mode

- Note:
  - You need to draw 6 pyramids to both eyes
    - NOT 3 pyramids for each eye
    - Meaning you should see all 6 pyramids in both eyes
  - Use GL_LINES and GL_TRIANGLES to draw lines/surface
    - If drawing surface, you might want to adjust the alpha value
  - If you do not hold the trigger (The Viewport Is Not Switching), you should be seeing red/green lines going out from your eyes.
- These should be rendered in Rift space
  - Meaning they are not rendered to your framebuffer

Extra Credit

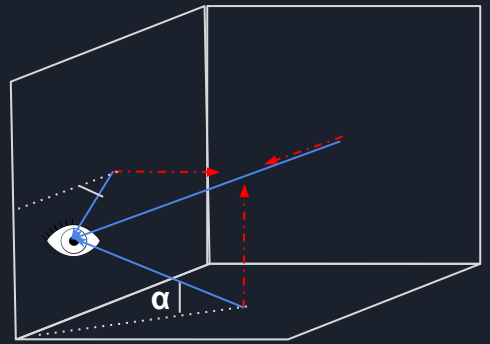
Extra Credit - Simulate One Screen Failing

- Press button to render one random black square screen (just one eye, not an entire wall - assuming using passive stereo)



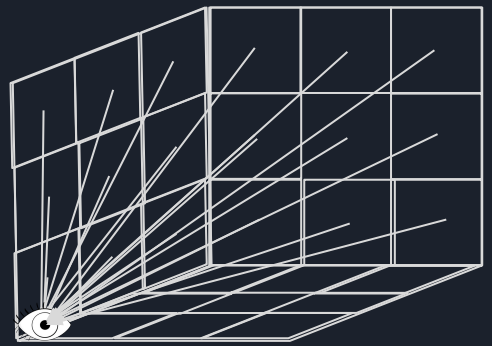
Extra Credit - Brightness falloff of LCD

- Reduce the brightness of the entire quad/screen based on the angle between you and the screen (0 - 90 degrees)
  - This will make the entire screen darker or brighter
  - 5 points



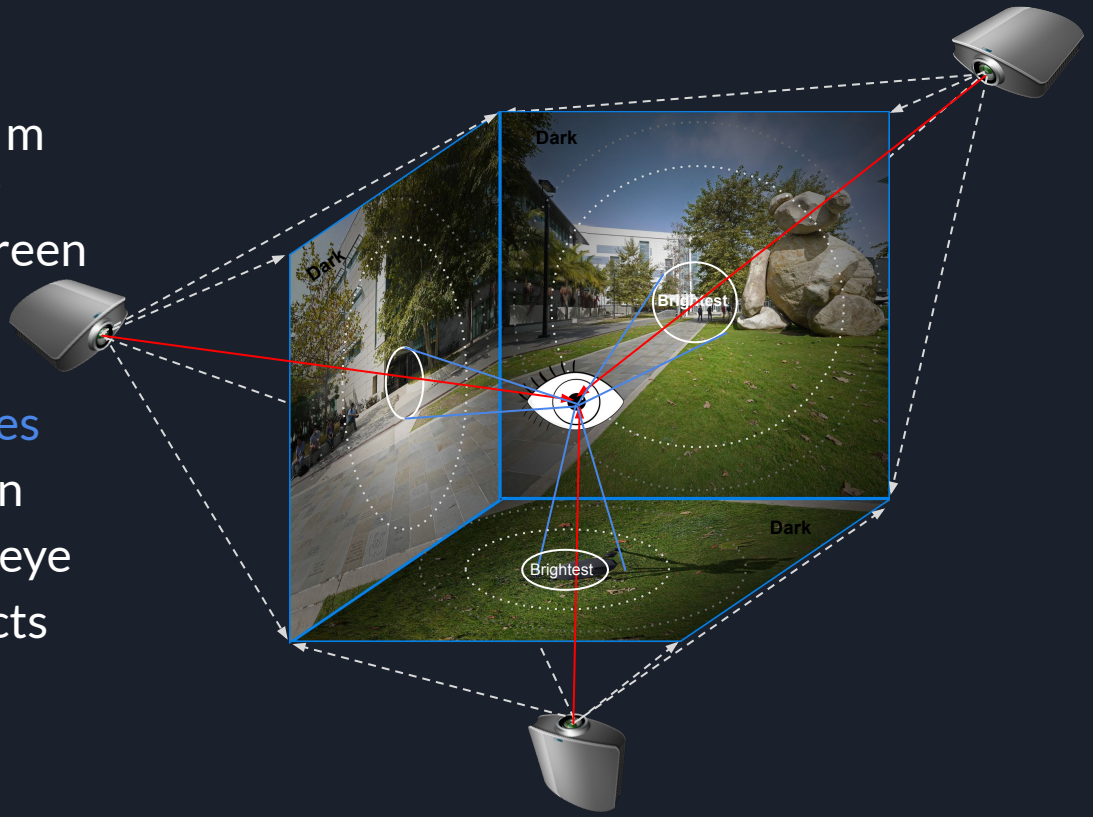
Extra Credit - Brightness falloff of LCD

- Reduce the brightness of each individual pixel based on the angle between you and that pixel (0 - 90 degrees)
  - Done through the shader
  - Brightest at the pixels that are looking directly to the eye
    - ie. normal to the frame
  - 5 points



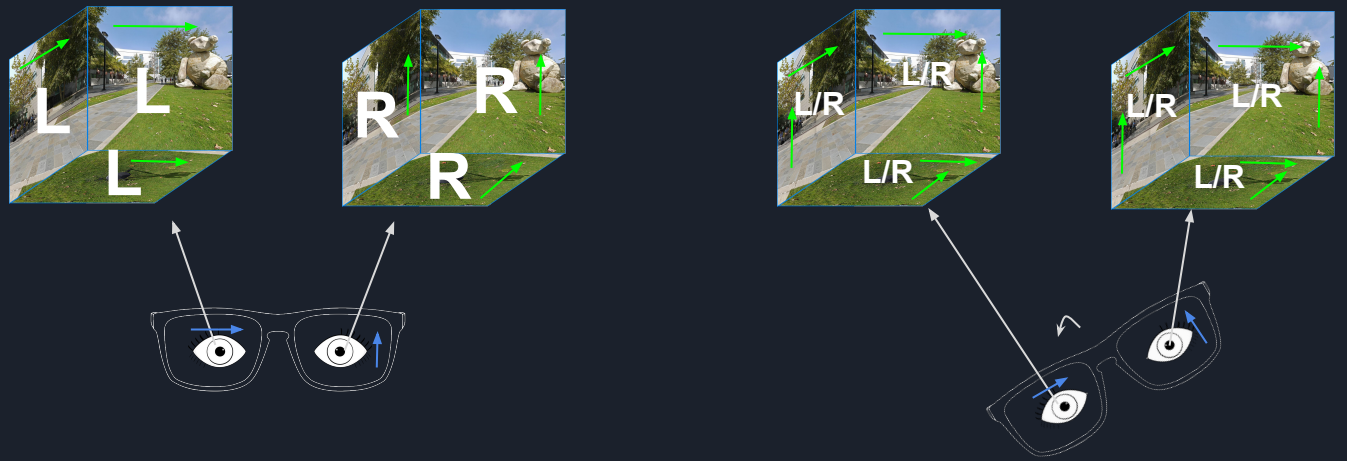
Extra Credit - Vignetting Projected Screens

- Imaginary projectors 2.4 m behind screen projecting onto the center of the screen
- Use shader
- Note:
  - Brightest point moves around depending on where the line from eye to projector intersects the screen



Extra Credit - Linear polarization effect

Polarized direction for glasses:
Polarized direction for cave screens:



QUESTIONS?