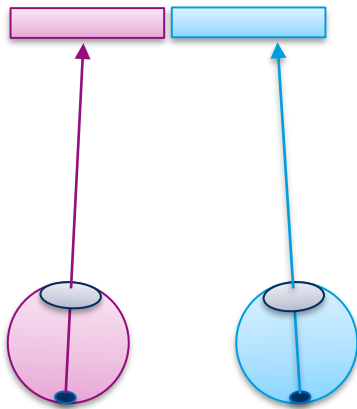
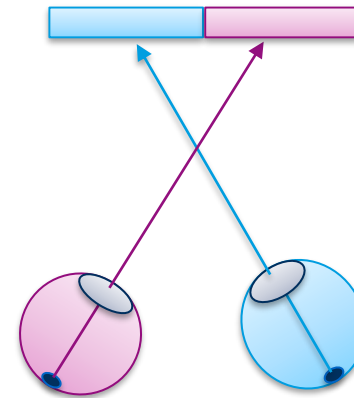


Stereoscopy



Stereo Vision and Rendering



Parallel Eye Viewing

get help [here](#)



Cross Eye Viewing

- ◆ Monoscopic Rendering(Recap)
 - ◆ Rendering Pipeline
 - ◆ Homogeneous Transformations
 - ◆ Viewing
- ◆ Stereo Rendering
 - ◆ Parallax & Depth Resolution
 - ◆ Stereoscopic Viewing & Lighting
 - ◆ Further Aspects
 - ◆ Rendering Process
 - ◆ Holographic Rendering
 - ◆ Immersive Setups and Visualization

Stereo – Rendering

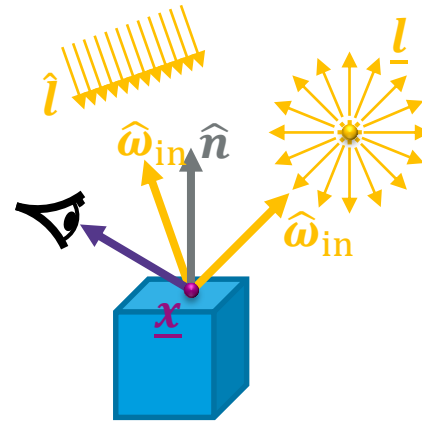
MONOSCOPIC RENDERING

RENDERING PIPELINE

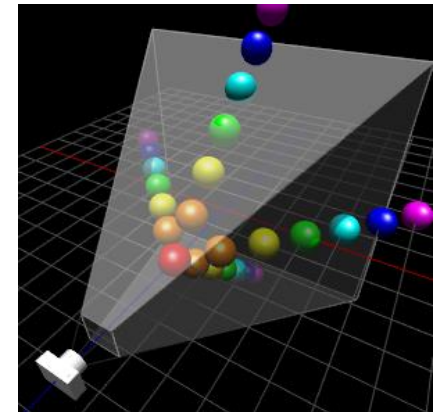
3D Rendering Constitutes



3D Scene Description



Light Sources and
Illumination Models

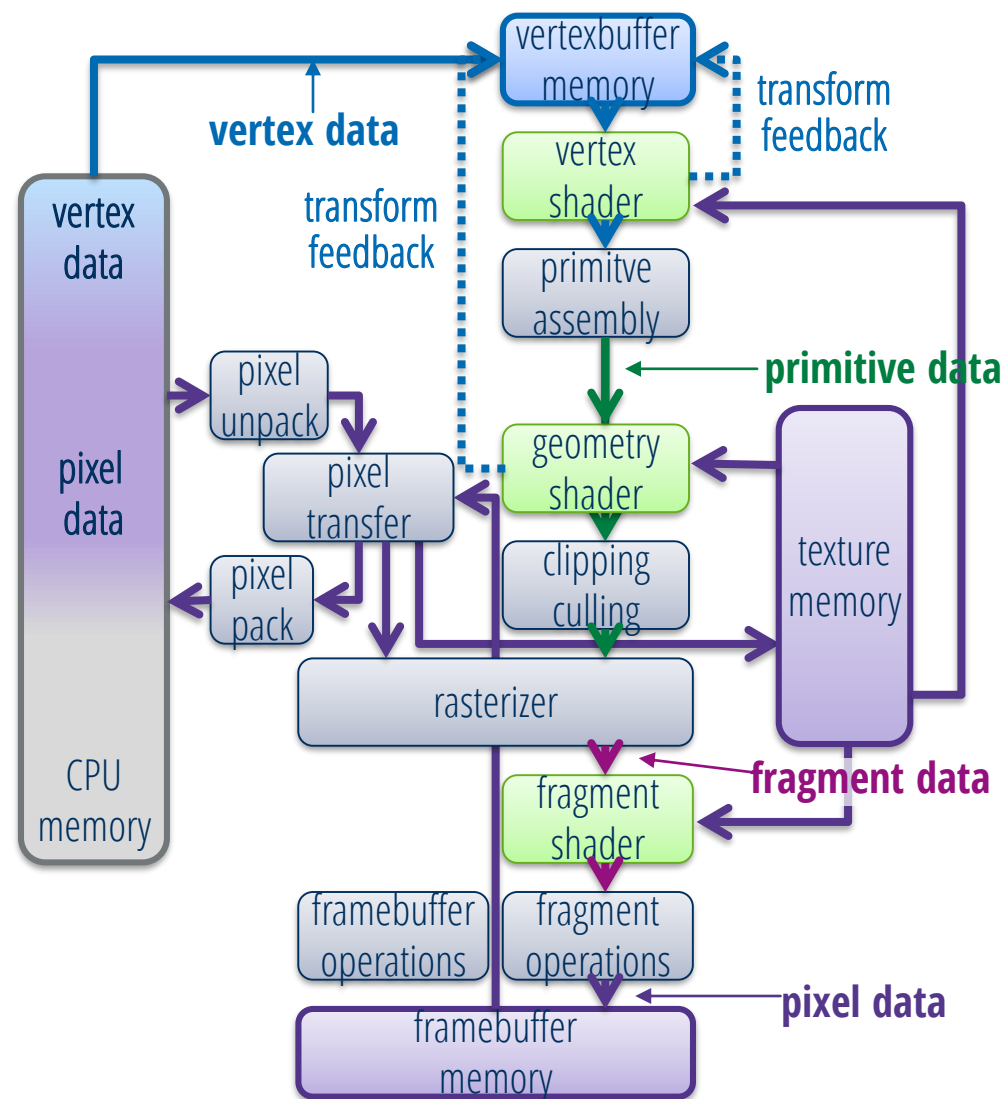


Camera Model and
Placement

Rendering Pipeline

Essentially do the following steps:

1. Create **shader programs**
2. Create **buffer objects** and **textures**, load data into them
3. Create **vertex array objects** that define how to read out vertex shader inputs from buffer objects
4. Set **uniform** variables for current frame
5. Emit **draw** calls to start rendering

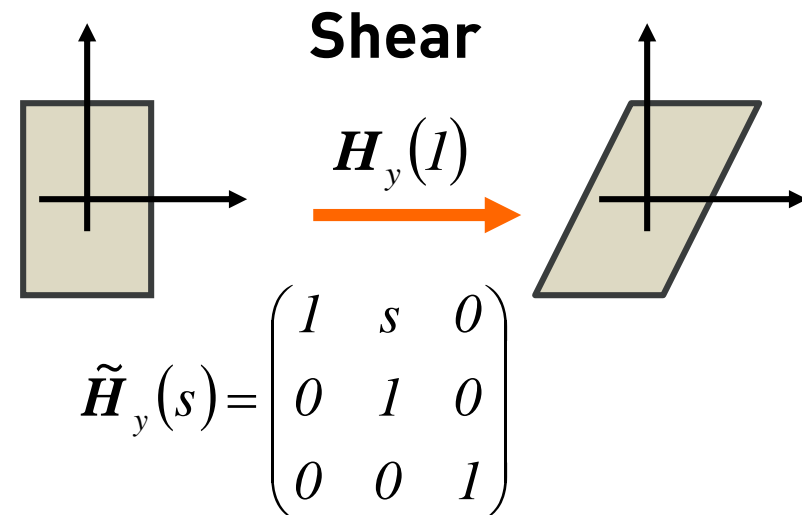
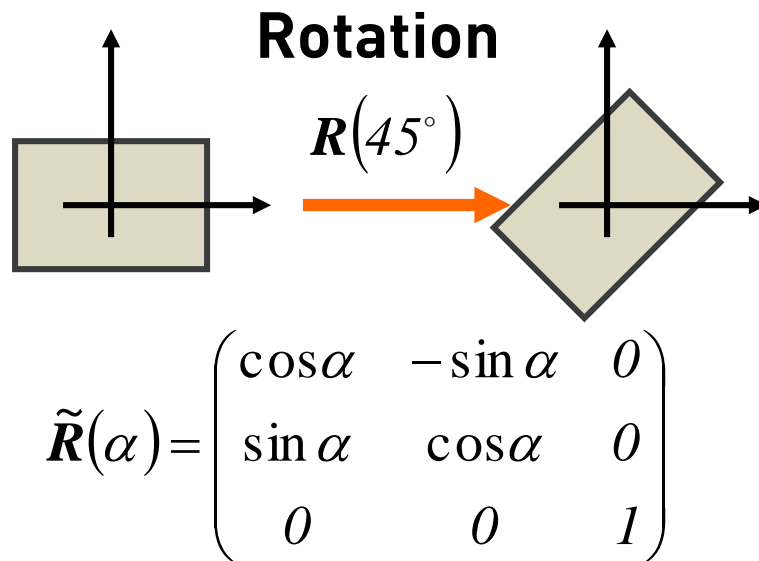
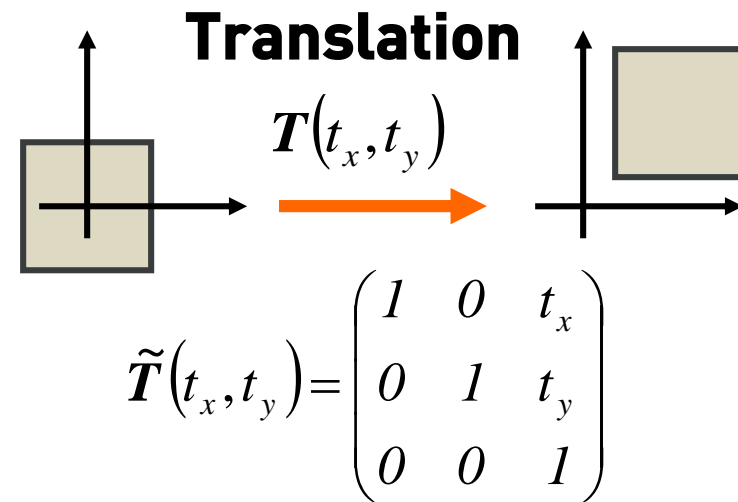
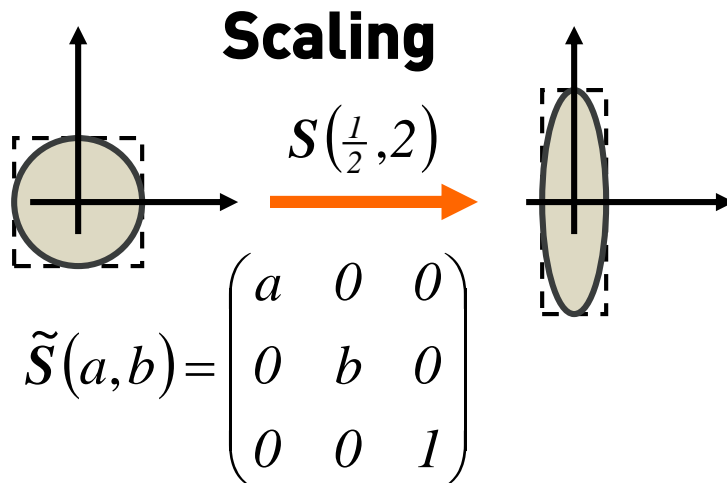


Stereo – Rendering

MONOSCOPIC RENDERING

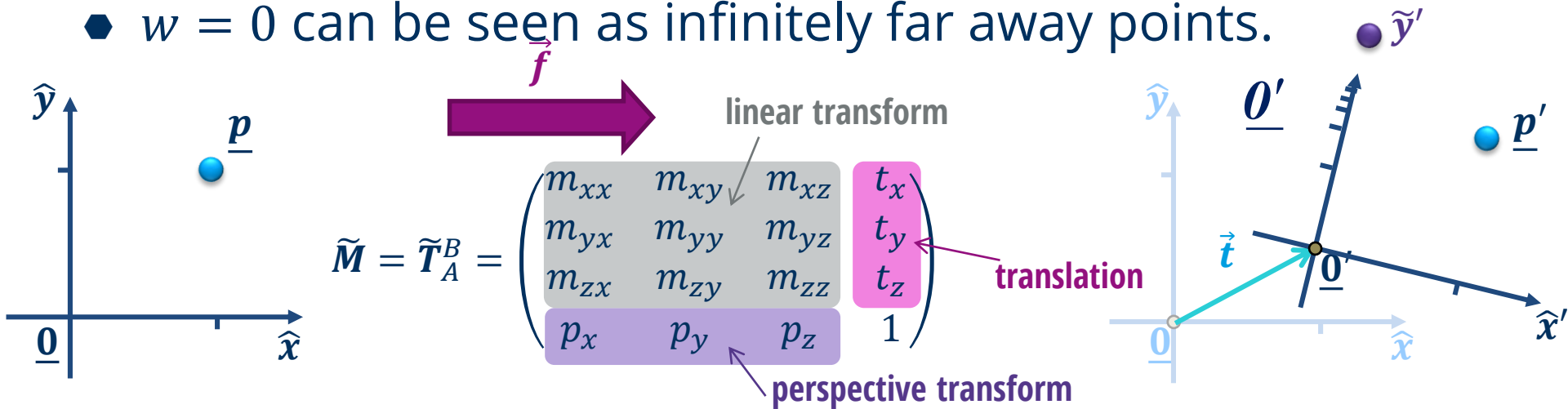
HOMOGENEOUS TRANSFORMATIONS

Homogeneous Matrices to Represent Transformations (2D)



Perspective Transforms

- ◆ generalize affine transforms and allow arbitrary w -values
- ◆ points are allowed in all columns of homogeneous matrix.
- ◆ $w = 0$ can be seen as infinitely far away points.



$$\underline{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \Rightarrow \tilde{p} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

1. homogenization

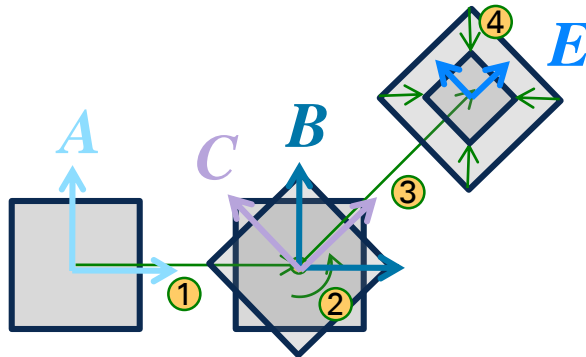
$$\tilde{p}' = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \tilde{M}\tilde{p}$$

2. transformation

$$\underline{p}' = \begin{pmatrix} x'/w' \\ y'/w' \\ z'/w' \end{pmatrix}$$

3. persp. divide

- ◆ perspective divide fails for points with $w = 0$.



$$A \xrightarrow{\tilde{T}_A^B} B \xrightarrow{\tilde{R}_B^C} C \xrightarrow{\tilde{T}_C^D} D \xrightarrow{\tilde{S}_D^E} E$$

① ② ③ ④

Sequence of coordinate transformations

$$\tilde{M}_A^E = \tilde{T}_A^B \tilde{R}_B^C \tilde{T}_C^D \tilde{S}_D^E$$

concatenated transformation

Two Interpretations of resulting Transformation Matrix:

Model Transformation

- ◆ Transform shape given in coordinate system A to coordinate system E
- ◆ But result is again in coordinate system A

System transformation

- ◆ Preserve shape given in coordinate system E but re-compute point coordinates such that they are given in coordinate system A

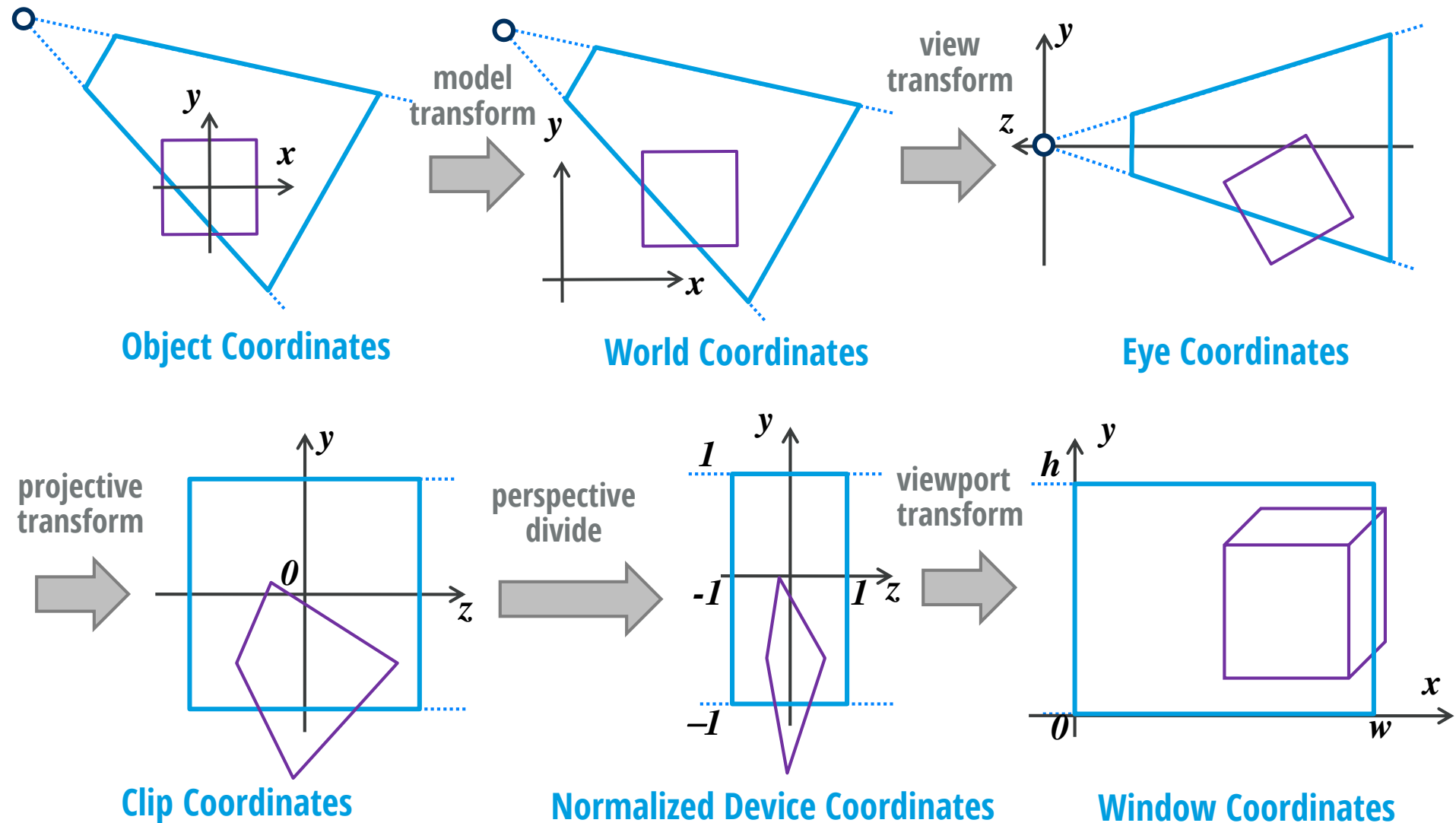
Stereo – Rendering

MONOSCOPIC RENDERING

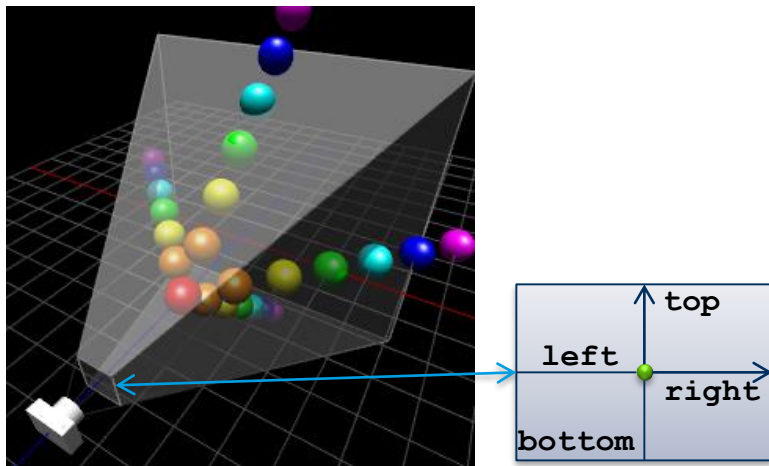
VIEWING

Transformations for Viewing

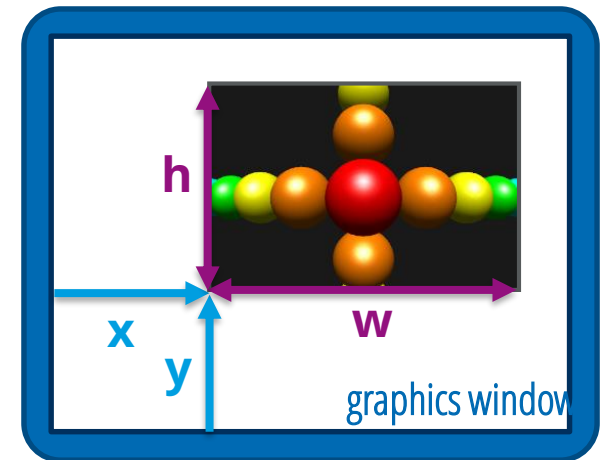
[see data viz on rendering](#)



Projection & Viewport Transform.



```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glFrustum(left, right, bottom, top, znear, zfar);  
ProjectionMatrix = glm::frustum(left, right, bottom, top, znear, zfar);
```



```
glViewport(x,y,w,h);
```

Persepctive Frustum Transform

- glFrustum takes six parameters:
 - l, r ... left and right on near clipping plane
 - t, b ... top and bottom on near clipping plane
 - n, f ... depth of near and far clipping planes
- For transformation to clip space we need the theorem of intersecting lines:

$$y_p = y_{eye} \cdot \frac{n}{-z_{eye}}$$

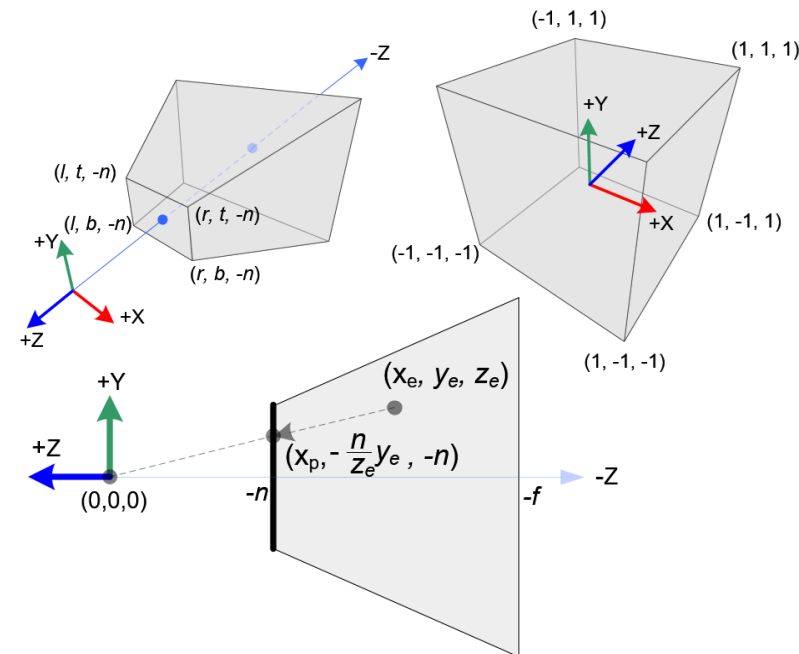
& renormalization to $y_{clip} \in [-1,1] \Leftarrow [b, t]$:

$$y_{clip} = \frac{2}{t-b} y_p - \frac{t+b}{t-b}$$

- Plugging both together gives:

$$y_{clip} = \left(\frac{2n}{t-b} y_{eye} + \frac{t+b}{t-b} z_{eye} \right) / (-z_{eye})$$

- w_{clip} is chosen to be $-z_{eye}$
- similarly for x-component
- z-entries follow from $z_{clip} \in [-1,1] \Leftarrow [n, f]$



http://www.songho.ca/opengl/gl_projectionmatrix.html

$$P_{frustum}(l, r, b, t, n, f) = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- Clip coordinates are computed via

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = \mathbf{P} \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

- Normalized device coordinates are then

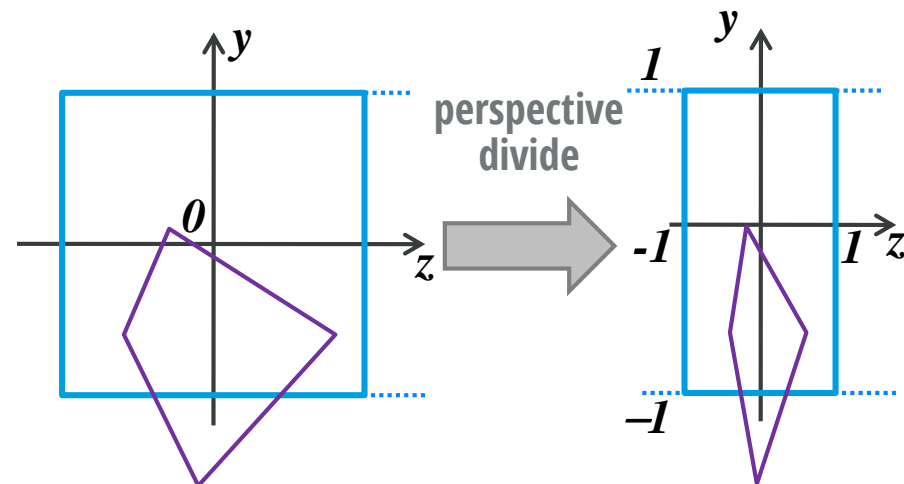
$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \frac{1}{w_{clip}} \begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \end{pmatrix}$$

- Finally we can compute window coordinates with the view port transformation and renormalizing z to [0,1]:

$$\begin{pmatrix} x_{window} \\ y_{window} \\ z_{window} \end{pmatrix} = \begin{pmatrix} \frac{w}{2}(x_{NDC} + 1) + x \\ \frac{h}{2}(y_{NDC} + 1) + y \\ \frac{1}{2}(z_{NDC} + 1) \end{pmatrix}$$

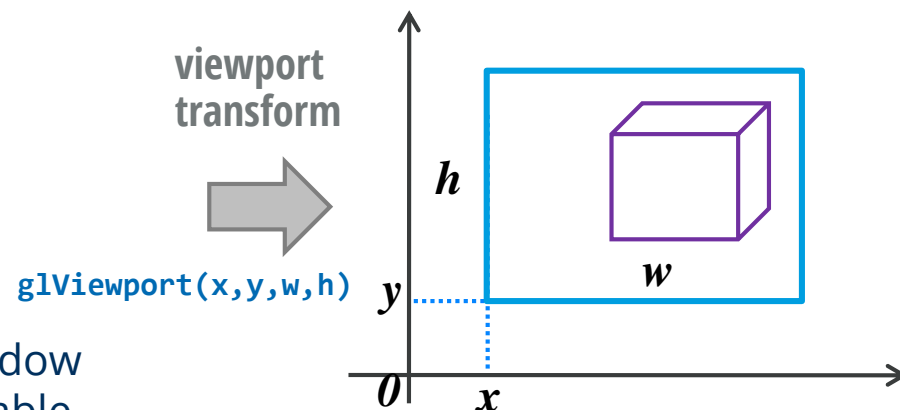
- In a fragment shader one can access window coordinates through the predefined variable

`glFragCoord ... vec4(xwindow, ywindow, zwindow, 1/wclip)`



Clip Coordinates

Normalized Device Crds



`glViewport(x,y,w,h)`

Window Coordinates

Stereo – Rendering

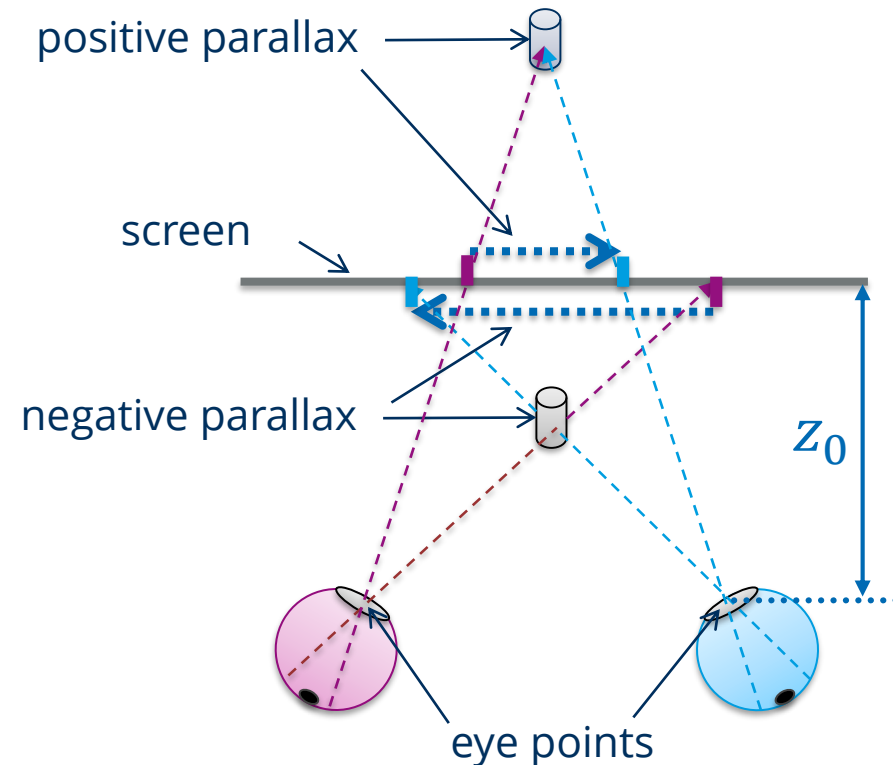
STEREO RENDERING

PARALLAX & DEPTH

RESOLUTION

Stereo Rendering - Parallax

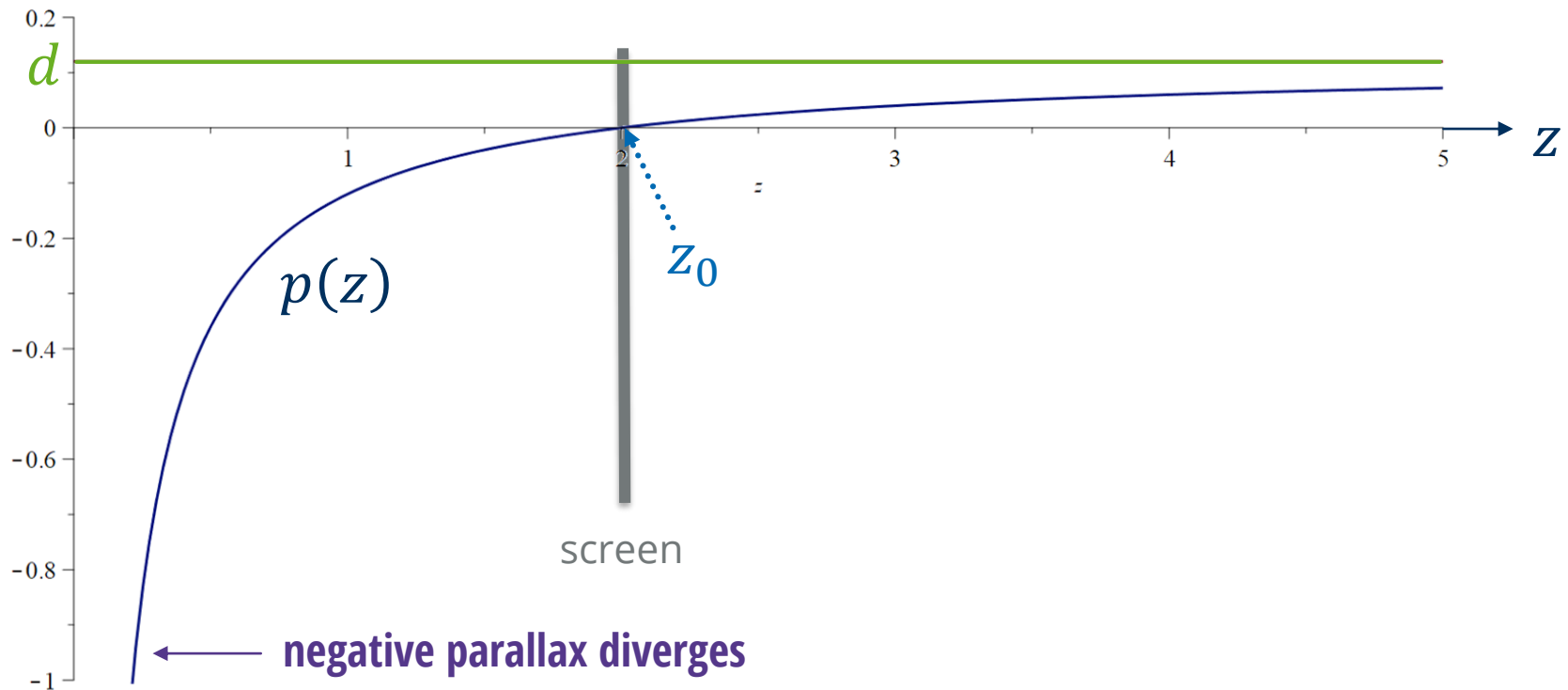
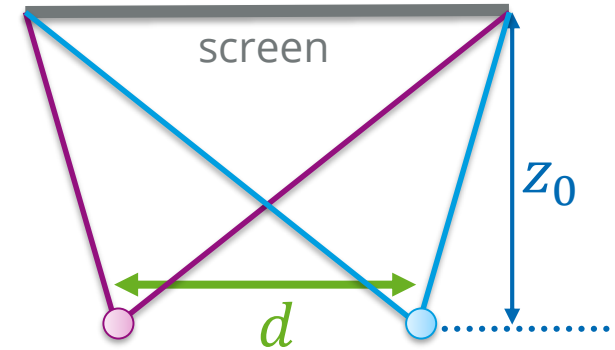
- ◆ On a screen, each 3D point projects along a straight line through the eye points onto two horizontally offset points
- ◆ The distance between the projection points is called **parallax**
 - ◆ negative for points in front of the screen and positive for points behind
 - ◆ zero for points on the screen



Stereo Rendering - Parallax

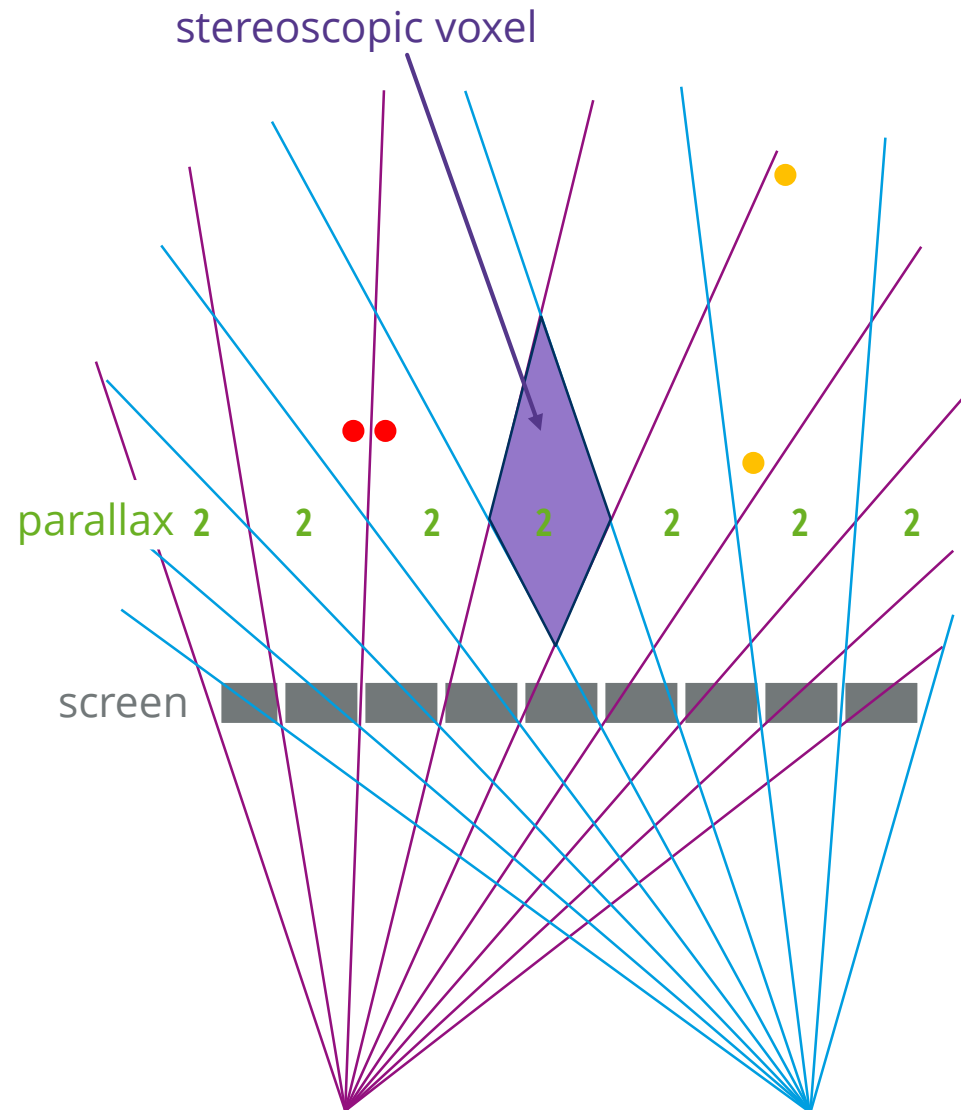
- Parallax p depends on depth z of object, screen depth z_0 (distance of screen wrt eye points) and eye separation d :

$$p(z) = d \cdot (1 - z_0/z)$$



Stereo Rendering – Depth Alias

- ◆ The screen resolution also influences depth resolution
- ◆ The figure shows rhombic areas of constant parallax
- ◆ The red dots show equi-depth points that have different parallax
- ◆ The orange points show equi-parallax points at different distance
- ◆ Multi-sampling for antialiasing helps

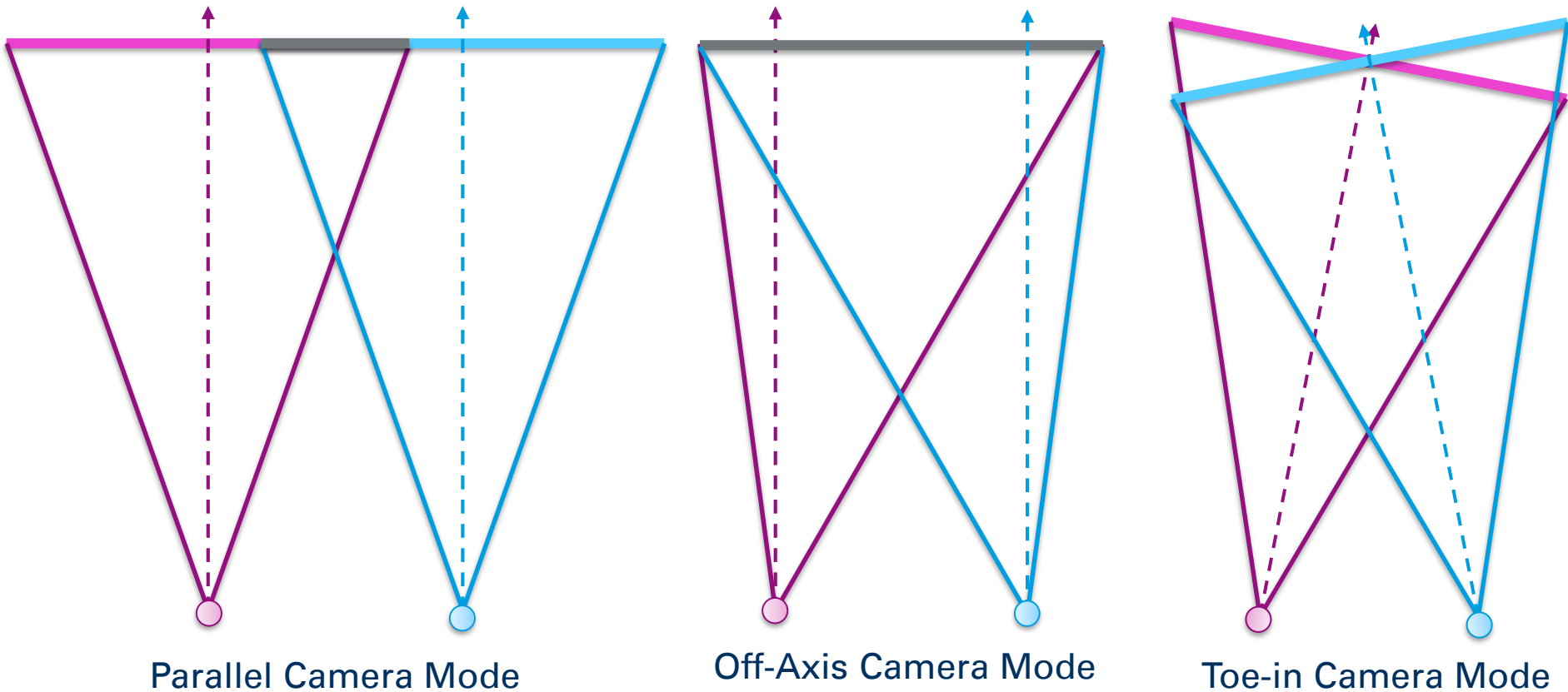


Stereo – Rendering

STEREO RENDERING

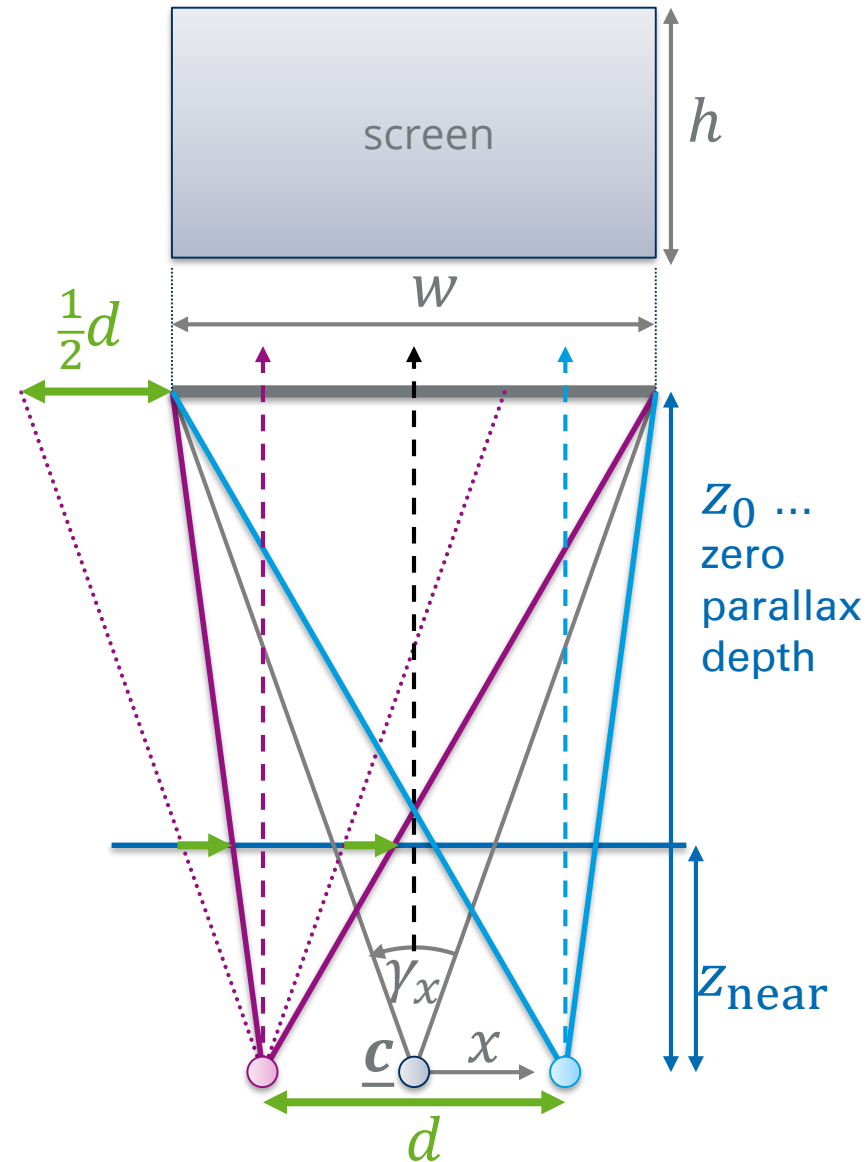
STEREOSCOPIC VIEWING & LIGHTING

Different Camera Modes



- ◆ For rendering on display wall or screen only the off-axis camera mode gives correct results

- ◆ parameters of central stereo projection
 - ◆ \underline{c} ... cyclopic eye position
 - ◆ d ... eye distance | separation
 - ◆ $a = w/h$... aspect ratio
 - ◆ γ_x ... horizontal field of view
 - ◆ z_{near} ... near clipping Distanz
 - ◆ z_0 ... screen depth / zero parallax depth
- ◆ view frusti for left / right eye are
 - ◆ translated by $\mp \frac{1}{2}d$ and
 - ◆ sheared by $\pm \frac{1}{2}d/z_0$



- two ways to construct projection matrix from cyclopic eye:
(subscript –/+ for left/right eye)

- with gluPerspective plus shear and translate:

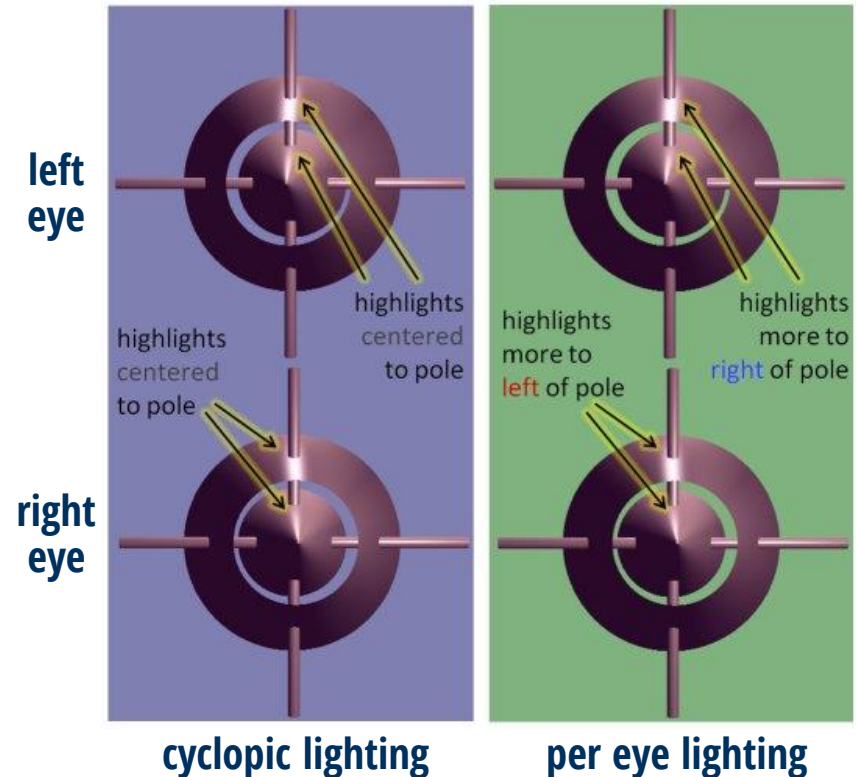
$$\tilde{\mathbf{P}}_{\pm} = \tilde{\mathbf{P}}_{\text{persp.}} \left(\gamma_x, \frac{w}{h}, n, f \right) \cdot \tilde{\mathbf{S}}_{\pm} \cdot \tilde{\mathbf{T}}_{\pm}$$

$$\tilde{\mathbf{S}}_{\pm} = \begin{pmatrix} 1 & 0 & \pm \frac{1}{2}d/z_0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

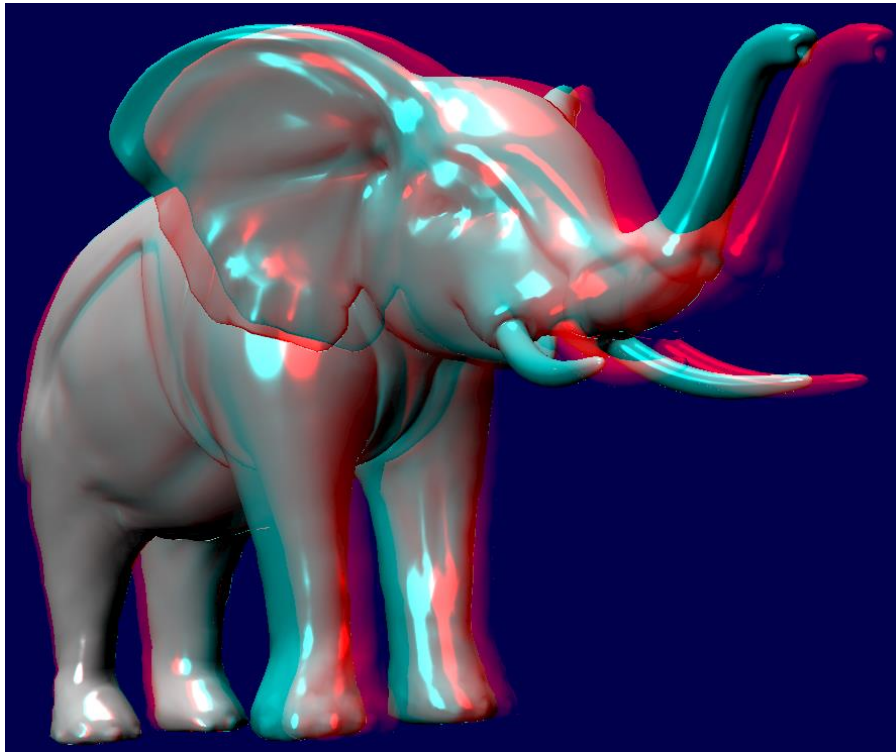
$$\tilde{\mathbf{T}}_{\pm} = \begin{pmatrix} 1 & 0 & 0 & \mp \frac{1}{2}d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- with glFrustum and translate:

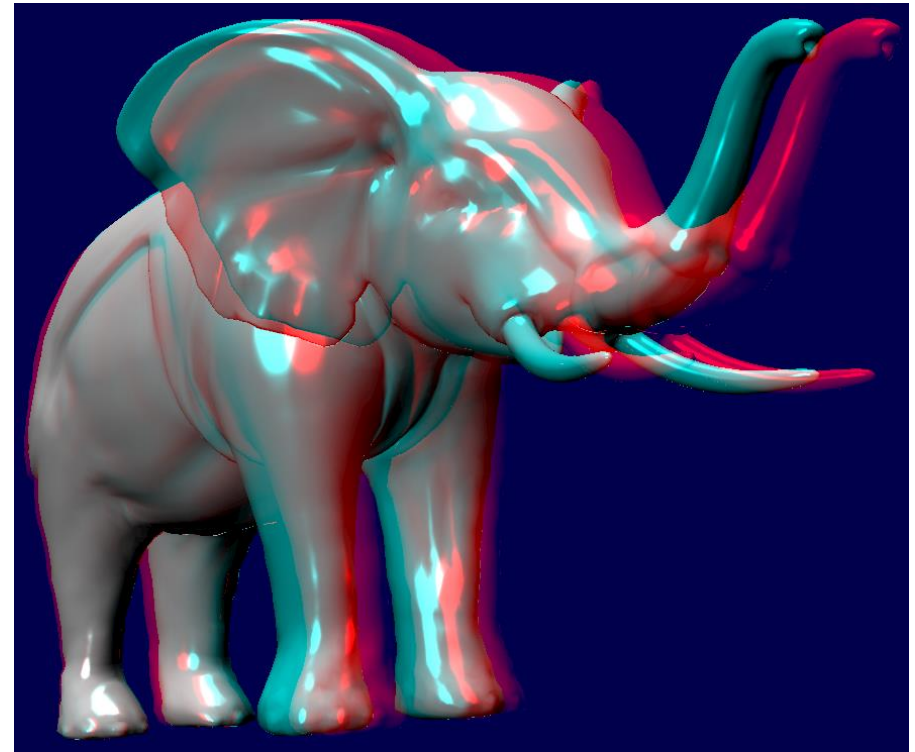
$$\tilde{\mathbf{P}}_{\pm} = \tilde{\mathbf{P}}_{\text{frustum},\pm}(l, r, b, t, n, f) \cdot \tilde{\mathbf{T}}_{\pm}$$



- eye location influences specular highlights.
- For cyclopic eye location highlights are at same scene position
- Otherwise they can appear at different depth than surface



cyclopic lighting



per eye lighting

In this example highlights with per eye lighting have very different shape around the eye of the elephant and lead to significant popout of highlights based on dead-eye effect

- Let matrix \tilde{M} represent transformation from model to cyclopic eye coordinates.

- The model view projection matrix for the two eyes computes to

$$\widetilde{MVP}_{\pm} = \tilde{P}_{\text{frustum},\pm} \cdot \tilde{T}_{\pm} \cdot \tilde{M}$$

- Depending on lighting approach this is split differently into model view and projection matrix:

	projection	model view
lighting in cyclopic eye: (convenient)	$\tilde{P}_{\text{frustum},\pm} \cdot \tilde{T}_{\pm}$	\tilde{M}
lighting in left/right eye: (physical)	$\tilde{P}_{\text{frustum},\pm}$	$\tilde{T}_{\pm} \cdot \tilde{M}$

- The non physical approach with the cyclopic eye defining the direction to the viewer is more convenient as specular lighting is located at the same surface location.

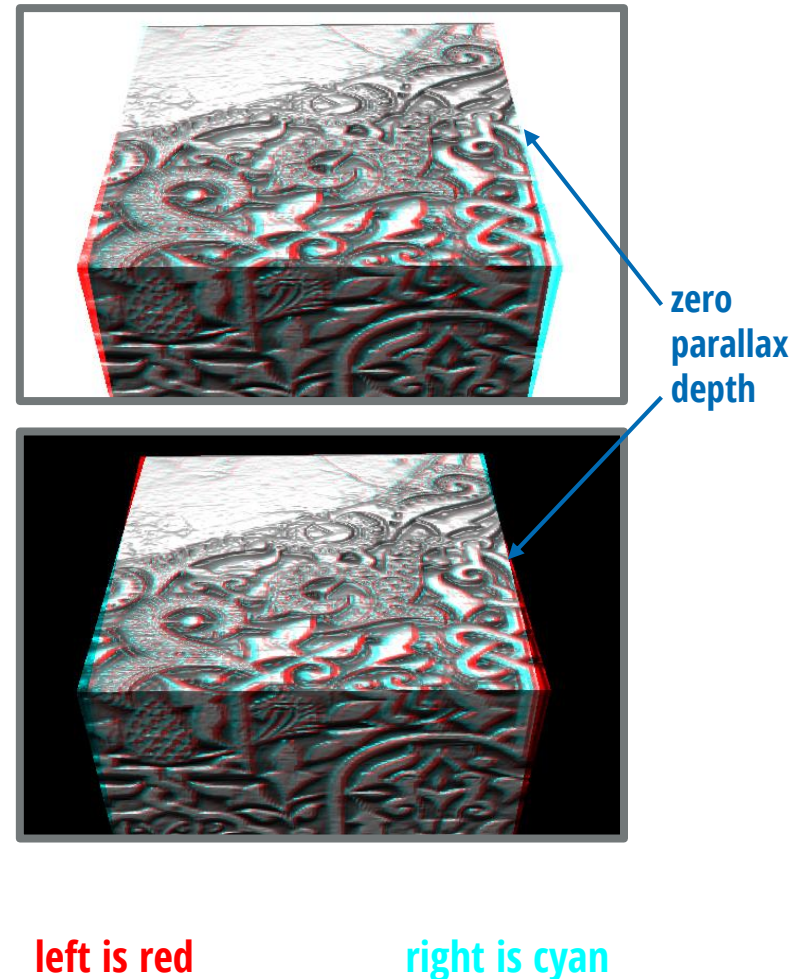
Stereo

STEREO RENDERING

FURTHER ASPECTS

Stereo Rendering – Validation 1

- the HVS can fuse stereo images even if left and right are swapped
- therefore it is important to know how to validate stereo images
- to perceive left or right image close the other eye
- where images coincide, there is parallax zero depth that is perceived on the plane of the screen
- to front of parallax zero depth, the left eye image should be to the right of the right eye image (crossed)
- careful with anaglyph: colored shadows depend on background. Black background is easier to interpret.



Stereo Rendering – Validation 2

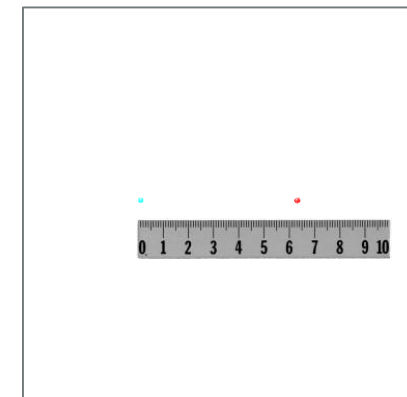
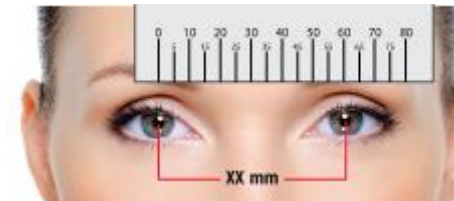
- Pupillary distance (PD) or interpupillary distance (IPD) is the distance measured in millimeters between the centers of the pupils of the eyes

IPD values (mm) from 2012 Army Survey

Gender	Sample size	Mean	Standard deviation	Minimum	Maximum	Percentile				
						1st	5th	50th	95th	99th
Female	1986	61.7	3.6	51.0	74.5	53.5	55.5	62.0	67.5	70.5
Male	4082	64.0	3.4	53.0	77.0	56.0	58.5	64.0	70.0	72.5

taken from [wikipedia](#)

- IPD can be measured with ruler
- The correct value for the eye distance d depends on size of screen
- One can adjust d with an object that is very far apart using a ruler again

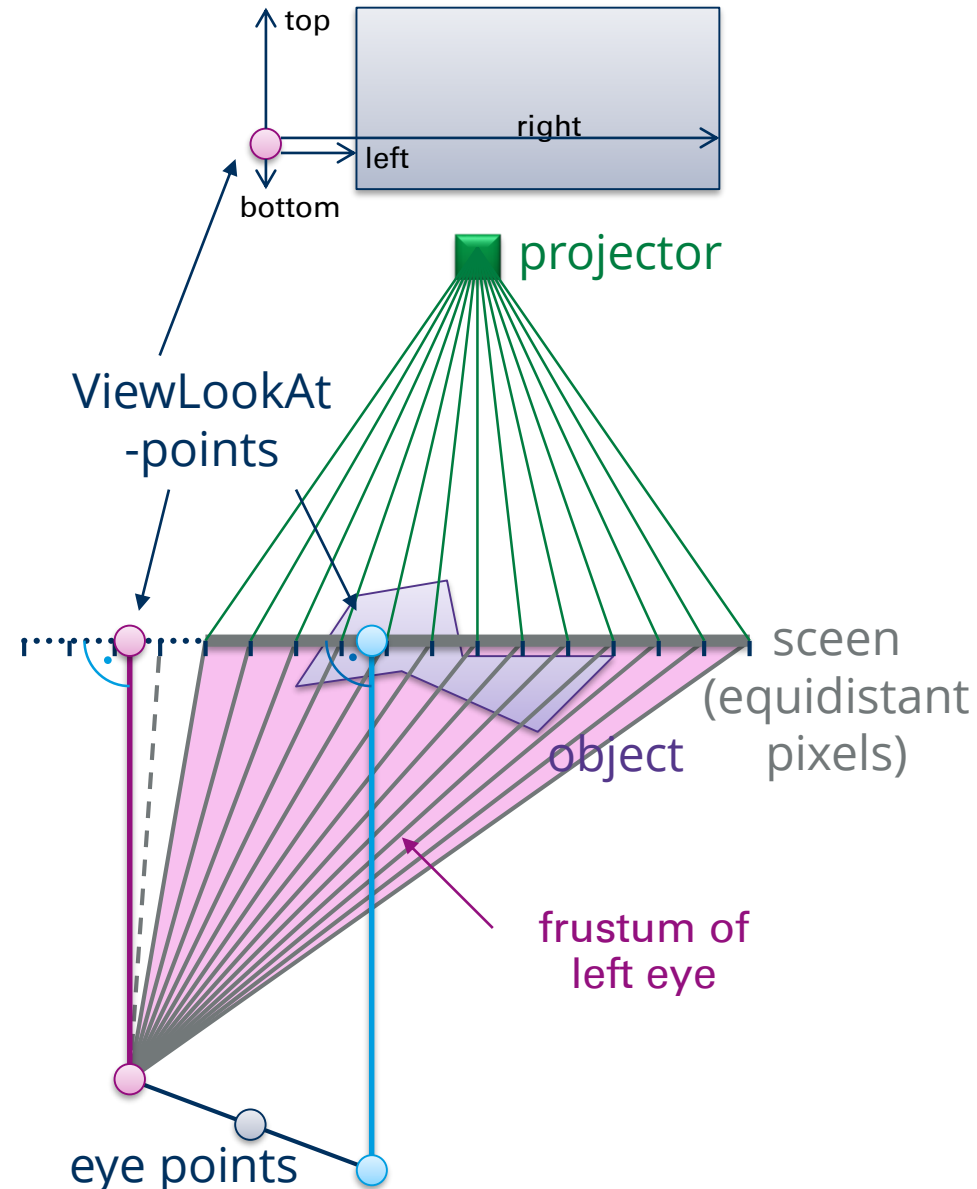


Stereo Rendering – General Case



```
glFrustum(left, right, bottom, top);
```

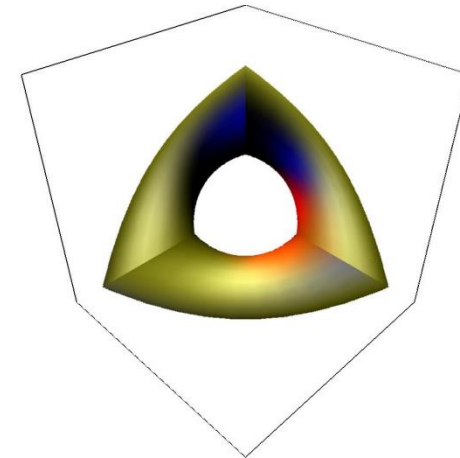
- For moving or multiple viewers eye locations need to be tracked with tracking system
- Projectors are centrally aligned to the projection screen and produce equidistant pixels
- The perspective transformation must project onto same pixel grid per eye:
 - Project eye orthogonally onto plane through projection screen (ViewLookAt is projected point, adopt ViewUp vector from projection screen)
 - Select frustum such that projected screen is selected



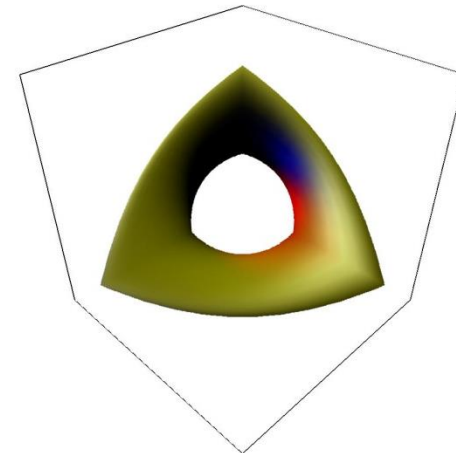
Stereo Rendering - Lighting

- An older rendering optimization was to use the negative z-direction instead of the direction to the viewer during lighting calculations
- Especially, for a multi-wall setup this won't work anymore.
- If you use deprecated OpenGL the wrong behaviour is default!
- You need to correct this with

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 1)
```



Z-parallel viewer during lighting yields inconsistent lighting



Consistent lighting for full lighting evaluation

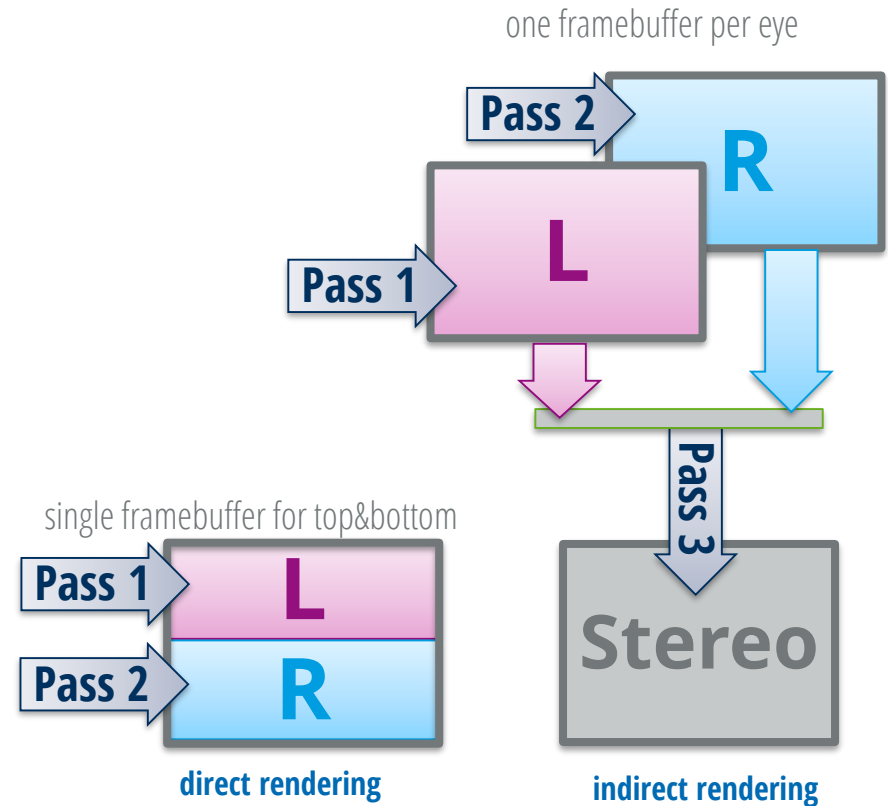
Stereo

STEREO RENDERING

RENDERING PROCESS

Stereo Rendering – Render Process

- There are different ways to organize stereo rendering
- **target mode**
 - **direct rendering** ... rendering is done into the buffers that are displayed on the screen
 - **indirect rendering** ... render into offline buffers and combine images in a finalizing render pass
- **render pass mode**
 - **two-pass rendering** ... rendering is done twice once for left and once for right eye
 - **one-pass** ... both eye images are rendered in single pass



- ◆ direct render mode is possible for anaglyph (via color mask), side-by-side, bottom&top (via viewport and scissor test), full 3D (via nvidia quadro GPU)

NVIDIA Quadro-GPU:

- ◆ Initialization of context:

```
glutInitDisplayMode(  
    GLUT_DOUBLE|GLUT_RGB|  
    GLUT_DEPTH|GLUT_STEREO);
```
- ◆ two-pass rendering:

```
glDrawBuffer(GL_BACK_LEFT);  
glClear(..)  
glFrustum(...)  
// left eye render pass  
glDrawBuffer(GL_BACK_RIGHT);  
glClear(..)  
glFrustum(...)  
// right eye render pass
```

Anaglyph:

- ◆ init: nothing special
- ◆ Two-pass rendering:

```
glClear(GL_COLOR_BIT |  
GL_DEPTH_BIT);  
glColorMask(1,0,0,1);  
glFrustum(...)  
// left eye render pass  
glColorMask(0,0,0,1);  
glClear(GL_COLOR_BIT |  
GL_DEPTH_BIT);  
glColorMask(0,1,1,1);  
glFrustum(...)  
// right eye render pass
```

[download: glsu](#)

- ◆ Indirect mode supports **all stereo** approaches and adds support for lenticular sheets (column interleaved), optimized anaglyph, random dot autostereograms
- ◆ Rendering is similar to direct mode but you need to construct **two framebuffers** each with color and depth buffer
- ◆ An option is to use **multi-sample buffers**, which is important for low or medium resolution HMDs
- ◆ Before each render pass **enable** corresponding **framebuffer** and **disable** after render pass
- ◆ **finalization render pass** can access both color and depth images to produce final stereo image

Stereo Rendering – Indirect Mode

```
protected:
    texture      col_texs[2], dep_texs[2], rnd_tex;
    frame_buffer fbos[2];
    shader_program finalize_prog;
public:
    stereo() : node("stereo"), col_texs{ "[R,G,B]", "[R,G,B]" },
              dep_texs{ "[D]", "[D]" }, rnd_tex("[R,G,B]")
    {
        for (int i = 0; i < 2; ++i) {
            col_texs[i].set_mag_filter(TF_NEAREST);
            col_texs[i].set_min_filter(TF_NEAREST);
            dep_texs[i].set_mag_filter(TF_NEAREST);
            dep_texs[i].set_min_filter(TF_NEAREST);
        }
    }
    void init_frame(context& ctx)
    {
        unsigned w = ctx.get_width(), h = ctx.get_height();
        if (!fbos[0].is_created()) {
            for (int i = 0; i < 2; ++i) {
                col_texs[i].set_width(w); col_texs[i].set_height(h);
                col_texs[i].create(ctx);
                dep_texs[i].set_width(w); dep_texs[i].set_height(h);
                dep_texs[i].create(ctx);
                fbos[i].create(ctx, w, h);
                fbos[i].attach(ctx, dep_texs[i]);
                fbos[i].attach(ctx, col_texs[i]);
                if (!fbos[i].is_complete(ctx)) {
                    std::cerr << "ups should be complete!" << std::endl;
                }
            }
        }
    }
};
```

declaration of color & depth textures, framebuffer object and shader program

Initialization of texture format and disabling of texture filtering

creation of textures and fbo as GPU objects based on render context ctx

textures are attached to fbo who is then checked for completeness

Stereo Rendering – Indirect Mode

```
void indirect_two_pass_stereo(context& ctx)
{
```

```
    mat4 MVP[2];
    // render once per eye
    for (int i = 0; i < 2; ++i) {
        // enable rendering to fbo of eye
        fbos[i].enable(ctx);
        // clear framebuffer and setup modelview projection of eye
        :
        // store per eye modelview projection matrix
        MVP[i] = ctx.get_projection_matrix()*ctx.get_modelview_matrix();
        // render scene
        render_scene(ctx);
        // recover previous state
        fbos[i].disable(ctx);
    }
```

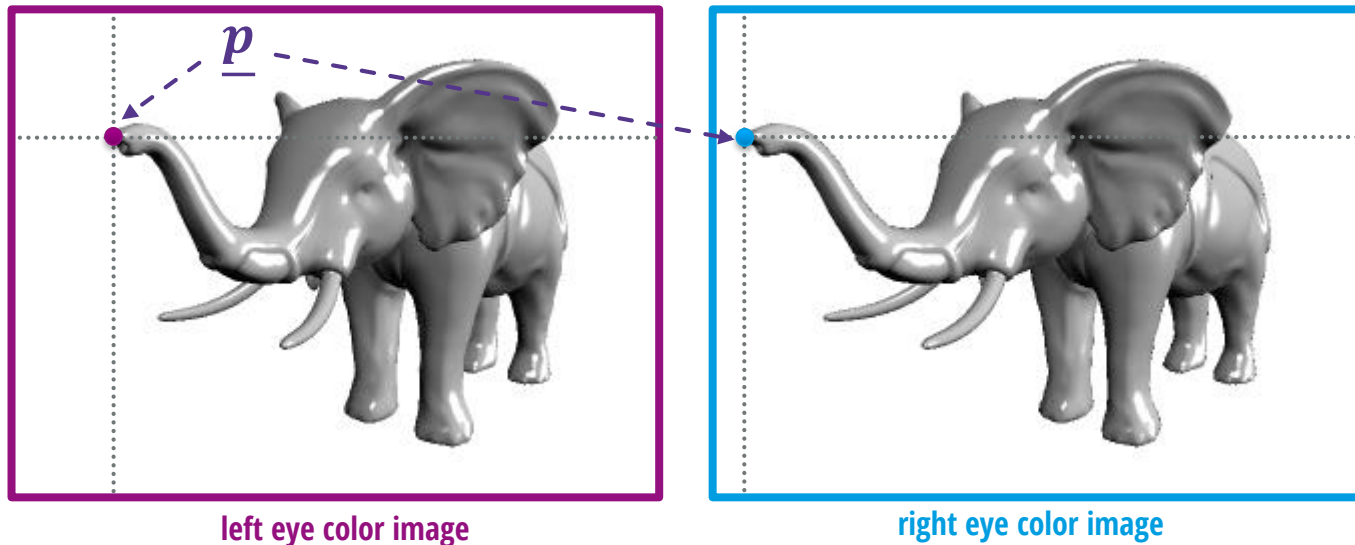
One offline render pass per eye. Modelview projection matrices are memorized

```
    mat4 iMVP[2] = { inv(MVP[0]), inv(MVP[1]) };
    col_texs[0].enable(ctx, 0); finalize_prog.set_uniform(ctx, "col_tex_0", 0);
    dep_texs[0].enable(ctx, 2); finalize_prog.set_uniform(ctx, "dep_tex_0", 2);
    :
    finalize_prog.set_uniform(ctx, "MVP_1", MVP[0]);
    finalize_prog.set_uniform(ctx, "iMVP_1", iMVP[0]);
    :
    glDisable(GL_DEPTH_TEST);
    render_screen_filling_quad(ctx, finalize_prog);
    glEnable(GL_DEPTH_TEST);
    col_texs[0].disable(ctx);
    dep_texs[0].disable(ctx);
```

Invert matrices, enable textures and pass uniforms to program

Render screen filling quad ignoring depth test. Here the finalize_prog is used to combine the two images in a single stereo frame

- ◆ For anaglyph & random dot autostereograms in finalization one can optimize perception of stereo image
- ◆ Per pixel one has access to left and right eye color and depth images
- ◆ In addition it is often necessary to access and synchronize the pixels in the two images that show the same surface point



Stereo Rendering - Remapping

pixel & tex coords

$$\underline{x} = (x, y) \quad \underline{\tau} = (u, v)$$

surface points seen at \underline{x} in each image

$$\underline{p}_0, \underline{p}_1$$

colors and depths at \underline{x}

$$\begin{matrix} \underline{c}_0 & \underline{c}_1 \\ \underline{d}_0 & \underline{d}_1 \end{matrix}$$

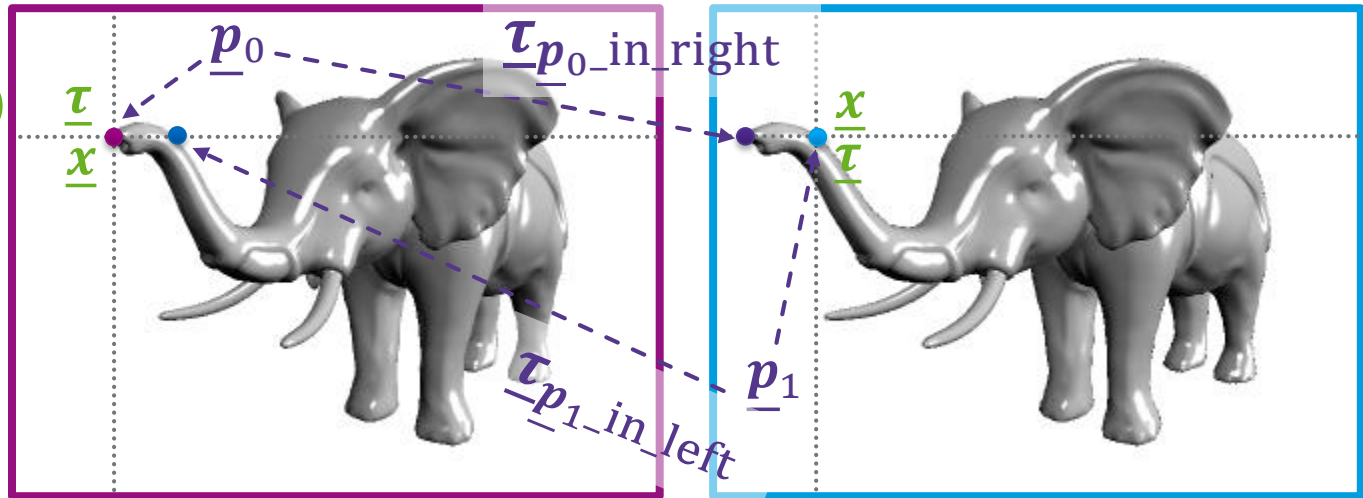
remapping surface points into other image by using texture coordinates

$$\underline{\tau}_{p_0_in_right}$$

$$\underline{\tau}_{p_1_in_left}$$

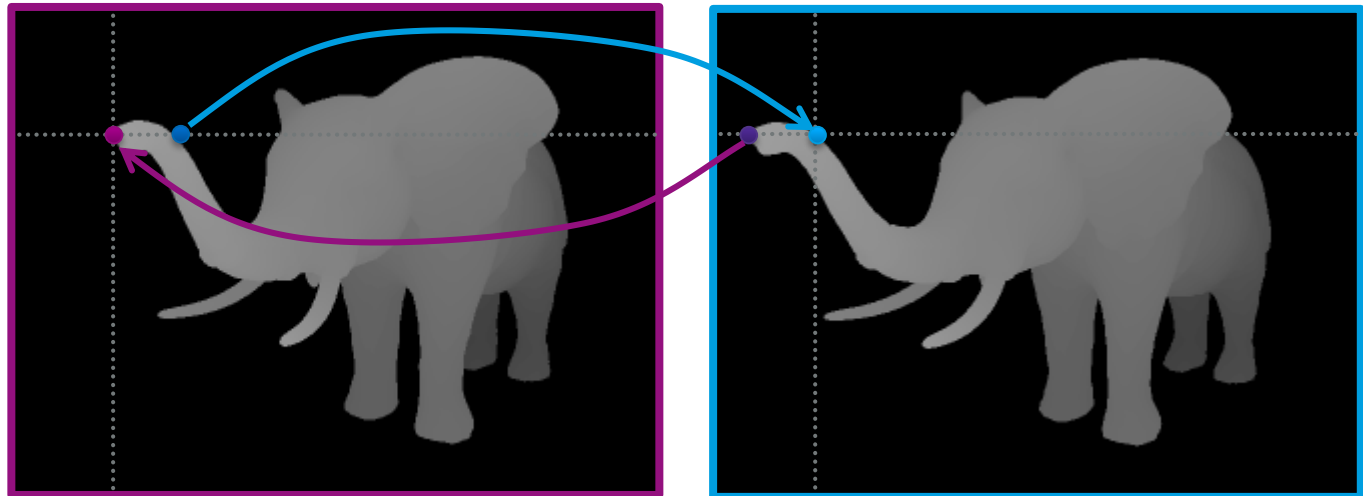
resulting in corresponding colors and depths from other image

$$\begin{matrix} \underline{c}_{0_other} & \underline{c}_{1_other} \\ \underline{d}_{0_other} & \underline{d}_{1_other} \end{matrix}$$



left eye color image

right eye color image



left eye depth image

right eye depth image

Stereo Rendering - Remapping

- For remapping one needs to compute texture coordinates $\underline{\tau}_{p_0\text{-in_right}}$, which is achieved as follows:
 - extract window coordinates of \underline{p}_0 from `gl_FragCoord.xy` and depth value from left eye depth image
 - Transport \underline{p}_0 back to world coordinates with inverse modelview projection matrix (MVP_l) of left eye
 - Transform \underline{p}_0 to clip coordinates of right eye with MVP_r
 - Perform w-clip and extract texture coordinates from normalized device coordinates
- Finally, use $\underline{\tau}_{p_0\text{-in_right}}$ to lookup corresponding color / depth values in right textures

$$\begin{pmatrix} x_{window} \\ y_{window} \\ z_{window} \end{pmatrix} = \begin{pmatrix} \frac{w}{2}(x_{NDC} + 1) + x \\ \frac{h}{2}(y_{NDC} + 1) + y \\ \frac{1}{2}(z_{NDC} + 1) \end{pmatrix}$$

[slide 11](#)

`vec3(gl_FragCoord.xy, texture(depth_1, texcrd))`

$$\begin{pmatrix} x_{window,l} \\ y_{window,l} \\ z_{window,l} \end{pmatrix}$$

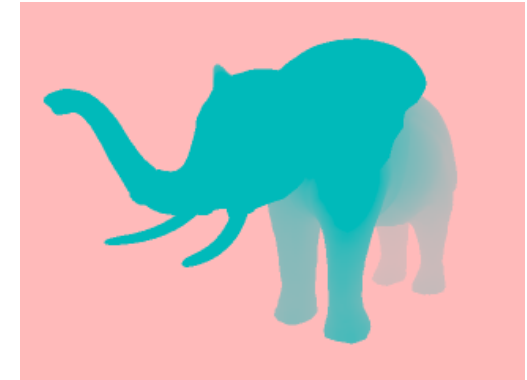
$$\begin{pmatrix} x_{NDC,l} \\ y_{NDC,l} \\ z_{NDC,l} \end{pmatrix} = \begin{pmatrix} \frac{2}{w}(x_{window,l} - x) - 1 \\ \frac{2}{h}(y_{window,l} - y) - 1 \\ 2z_{window,l} - 1 \end{pmatrix}$$

$$\begin{pmatrix} x_{clip,r} \\ y_{clip,r} \\ z_{clip,r} \\ w_{clip,r} \end{pmatrix} = MVP_r MVP_l^{-1} \begin{pmatrix} x_{NDC,l} \\ y_{NDC,l} \\ z_{NDC,l} \\ \mathbf{1} \end{pmatrix}$$

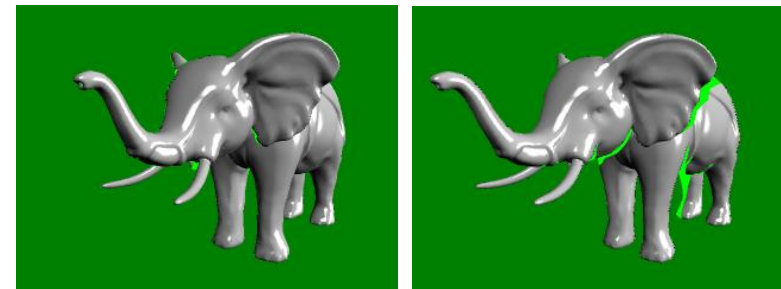
$$\begin{pmatrix} x_{NDC,r} \\ y_{NDC,r} \\ z_{NDC,r} \end{pmatrix} = \frac{1}{w_{clip,r}} \begin{pmatrix} x_{clip,r} \\ y_{clip,r} \\ z_{clip,r} \end{pmatrix}$$

$$\underline{\tau}_{p_0\text{-in_right}} = \frac{1}{2} \begin{pmatrix} x_{NDC,r} + 1 \\ y_{NDC,r} + 1 \end{pmatrix}$$

- With the help of remapping one can
 - Compute the **parallax vector** $\vec{\pi}$ in pixel or texture coordinates:
$$\vec{\pi}_0 = \underline{\tau}_{p_0\text{-in-right}} - \underline{\tau}, \quad \vec{\pi}_1 = \underline{\tau} - \underline{\tau}_{p_1\text{-in-left}}$$
For horizontal eye separation the parallax vectors y/v-component is always zero.
 - Check if surface point p_0 seen in left image is **visible** in right image:
$$d_{0_other} = depth_{\text{right}}(\underline{\tau}_{p_0\text{-in-right}}) \approx d_0$$
If violated, only $d_{0_other} < d_0$ is possible.
- Both approaches work in the other direction too, thus
 - if both visibility tests succeed, eye can fuse corresponding points. Better fusion:
 - for anaglyph ensure equal perceived brightness
 - for autostereograms ensure random dots synchronization



parallax mapped to color
(light blue ... negative x, light red ... positive x)



remapped color on left and right image
(light green shows parts that are invisible in other image)

- ◆ In single pass rendering
 - ◆ scene is traversed once
 - ◆ state changes happen once
 - ◆ draw calls only once
- ◆ In applications where any of these is expensive single pass rendering can **speed up** rendering
- ◆ Approaches
 - ◆ use **geometry shader** to duplicate geometry
 - ◆ use **instancing** to duplicate geometry
- ◆ Important Precondition
 - ◆ need to be able to **select** left or right **color AND depth buffer** in shader program
 - ◆ under DirectX this can be done by using a **stereo surface**
 - ◆ under OpenGL one has to use a single side-by-side or bottom & top framebuffer and control the viewport with the **ARB_viewport_array** extension (core since 4.1):
 - ◆ Define viewport/scissor arrays for left & right images
 - ◆ Select viewport in shader with **gl_ViewportIndex**

1. Restrict maximum parallax:
 $\pm 1.6^\circ \sim \text{parallax} \leq 0.03 \cdot \text{distance to screen}$
2. when single object is examined →
place object center on screen center
3. extended scene → approximately 1/3 negative
parallax, 2/3 positive parallax
4. keep all objects with negative parallax close to screen
center
5. Don't move objects/scene too fast away or towards
observer in order to allow user to adapt vergence



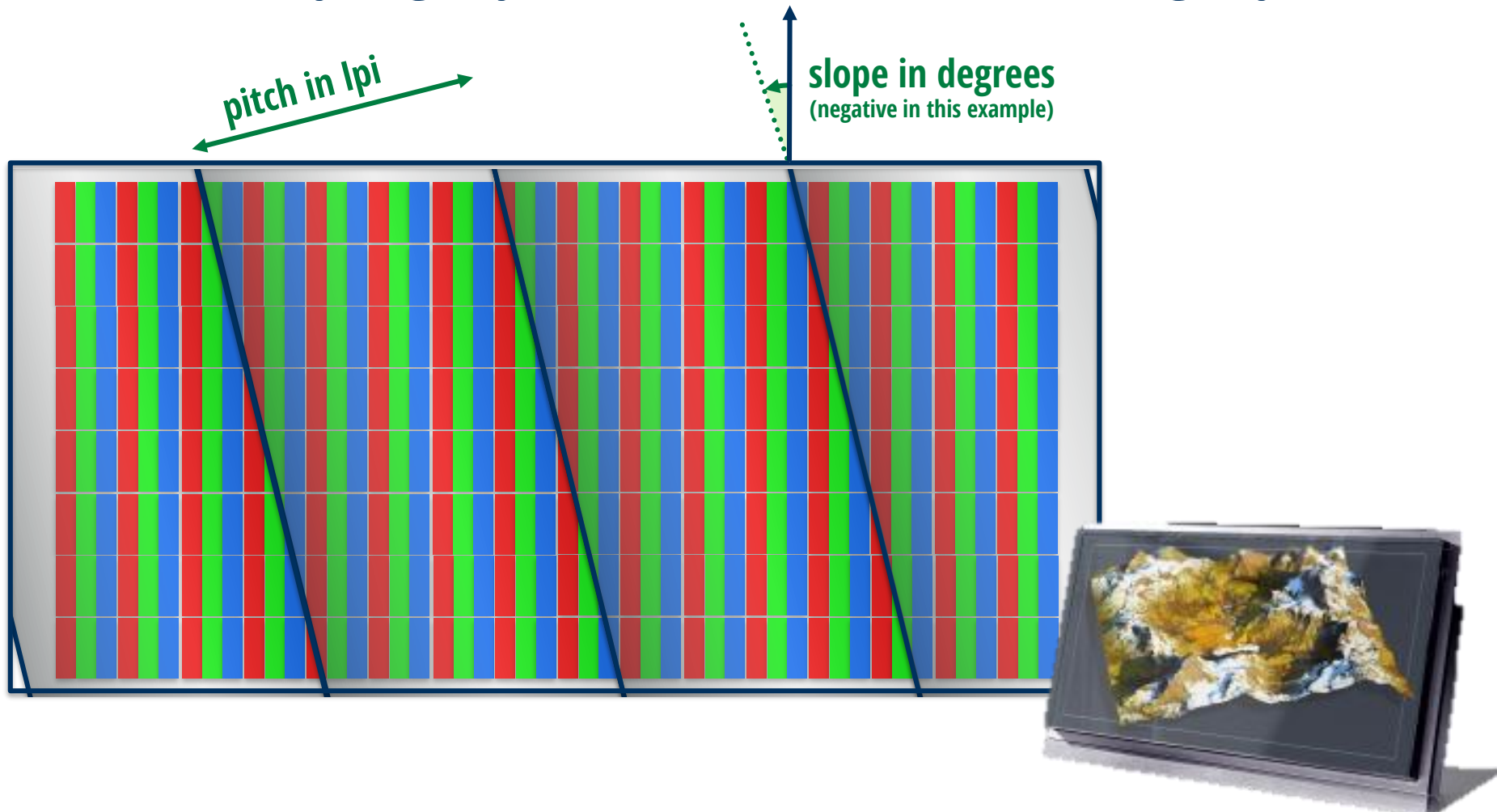
Stereo

STEREO RENDERING

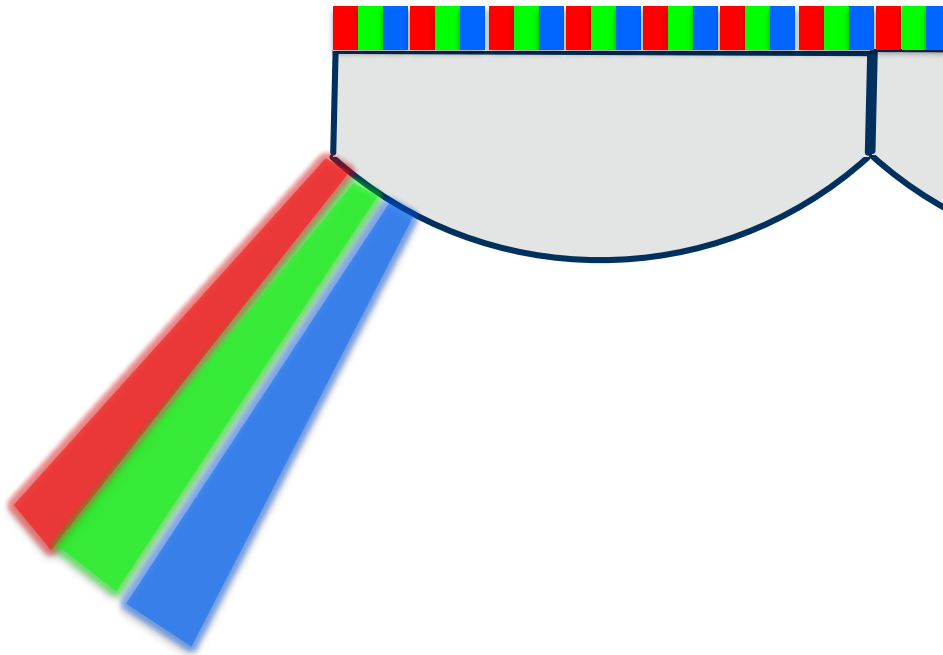
HOLOGRAPHIC RENDERING

Looking Glass Display

- ◆ Display is assembled of a high-resolution TFT-display covered by slightly rotated lenticular sheet slightly

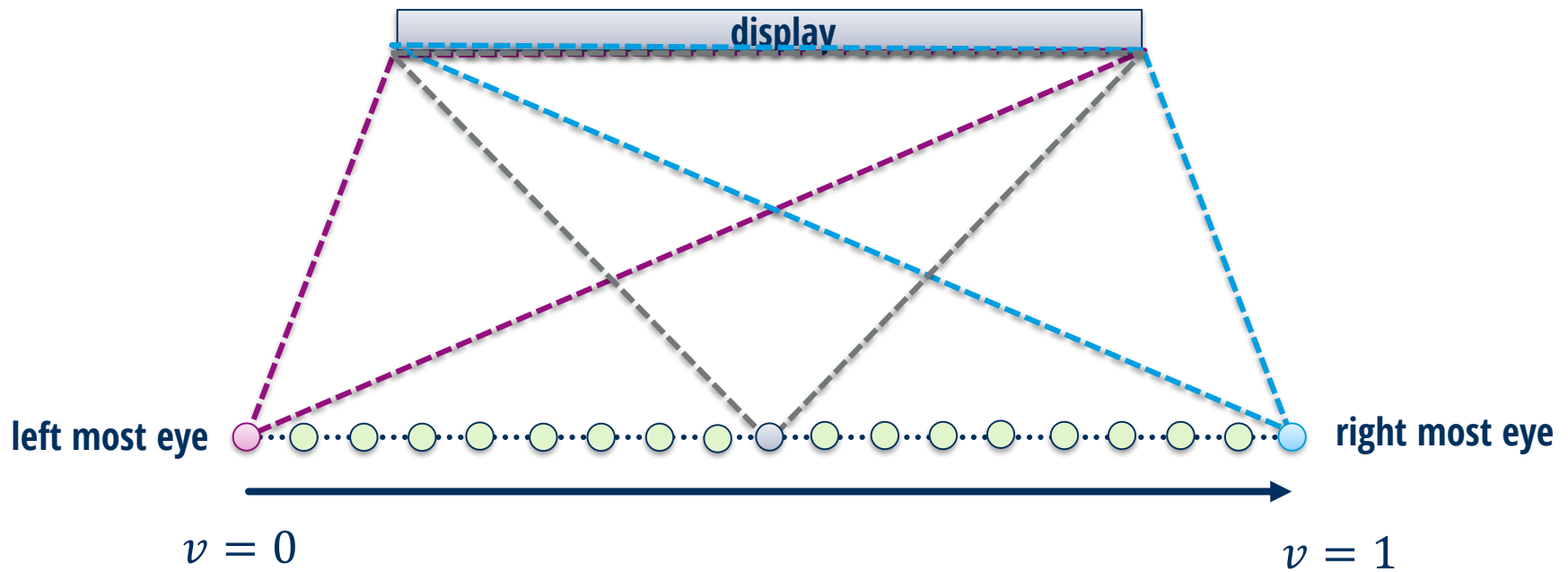


- ◆ Each sub-pixel emits light to a different direction, therefore holographics rendering is done on per sub-pixel basis



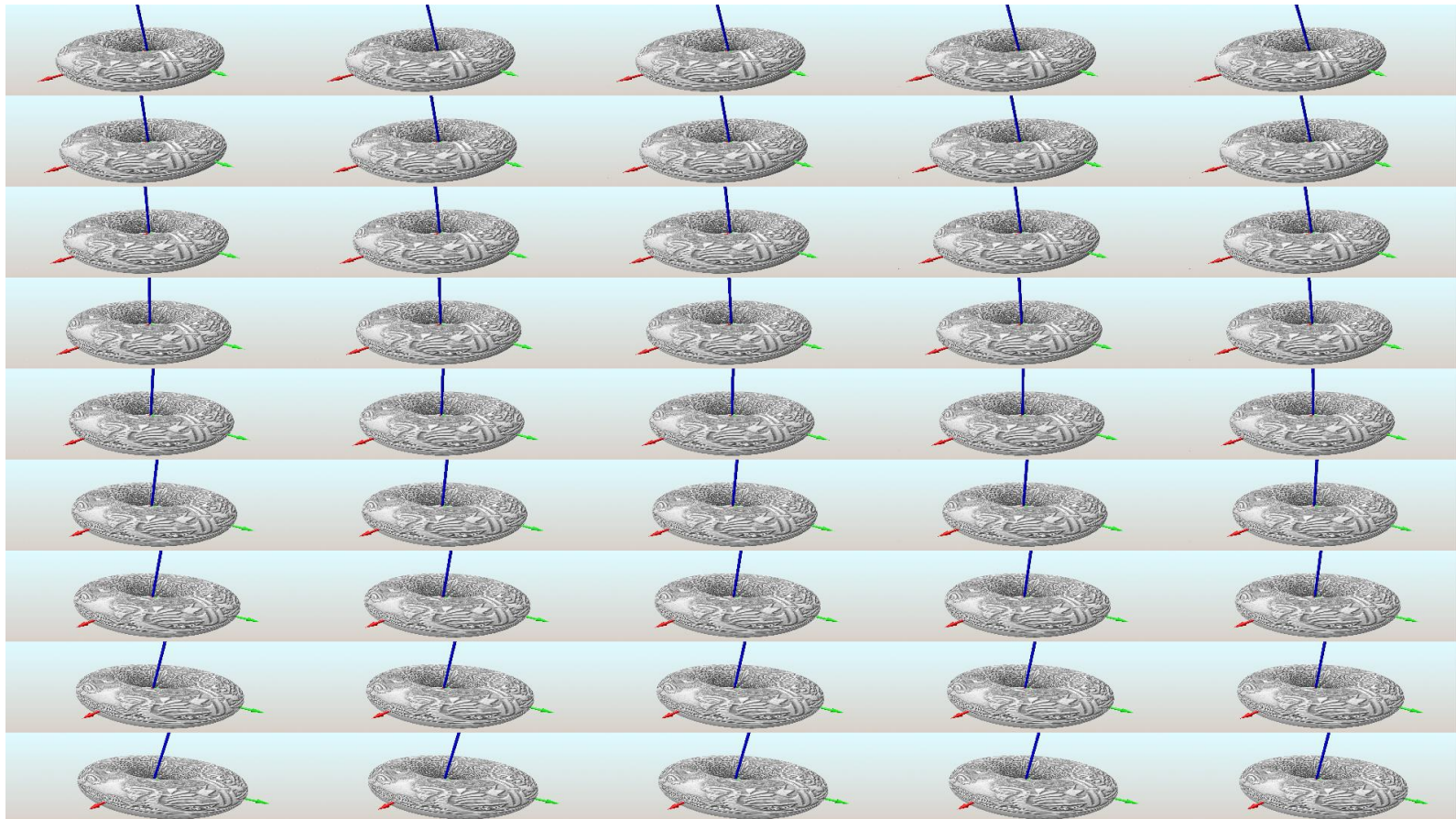
Looking Glass Display – Rendering

- ◆ Default rendering is similar to stereo rendering with wide baseline and large number of views arranged in an image quilt or a volumetric image with baseline location v addressing the 3rd coordinate

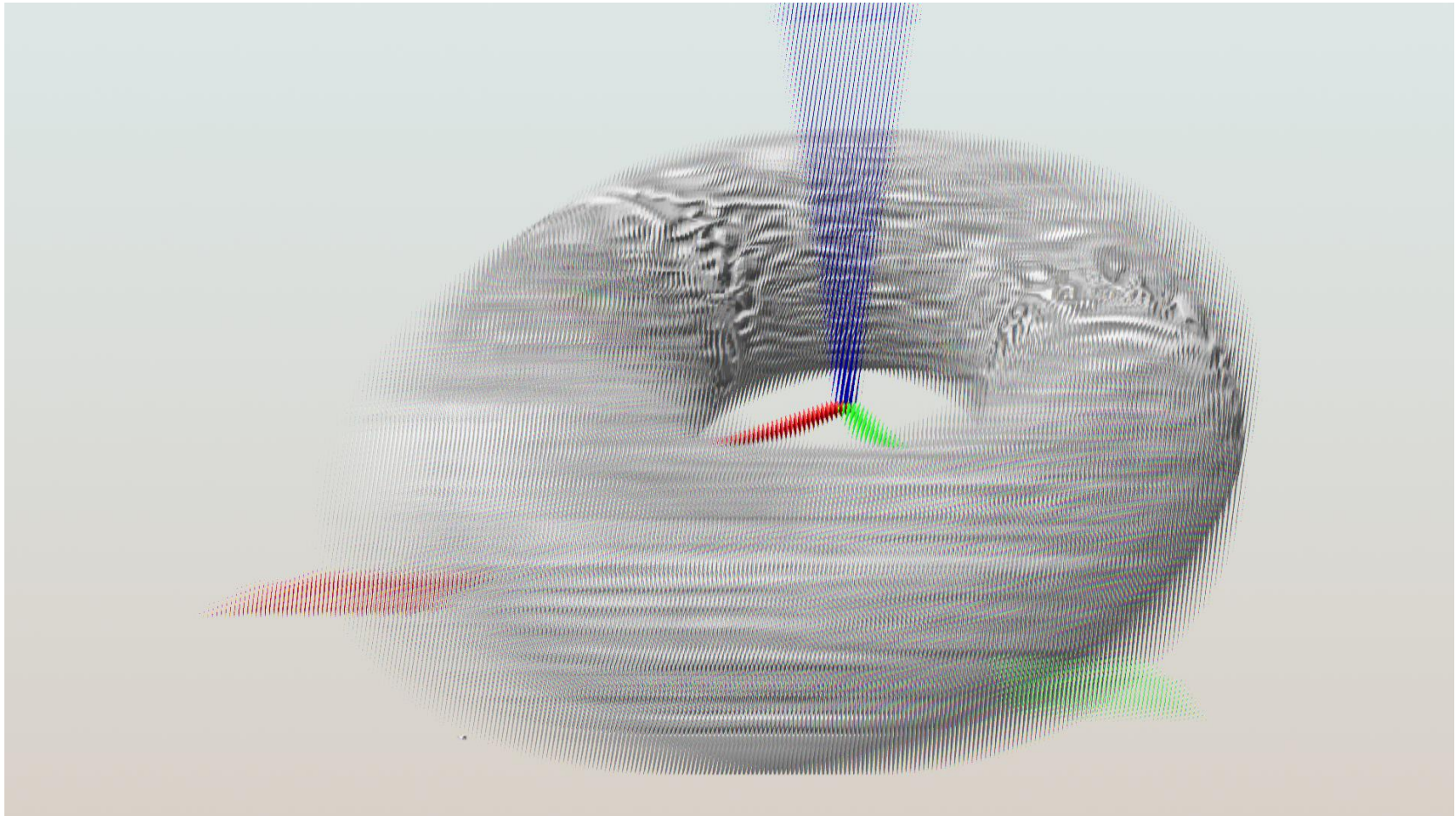


Looking Glass - Quilt

- ◆ Example of quilt were v increases from top left to bottom right in along rows of the quilt



- ◆ The looking glass display driver combines the quilt images into a hologram on a per sub-pixel basis



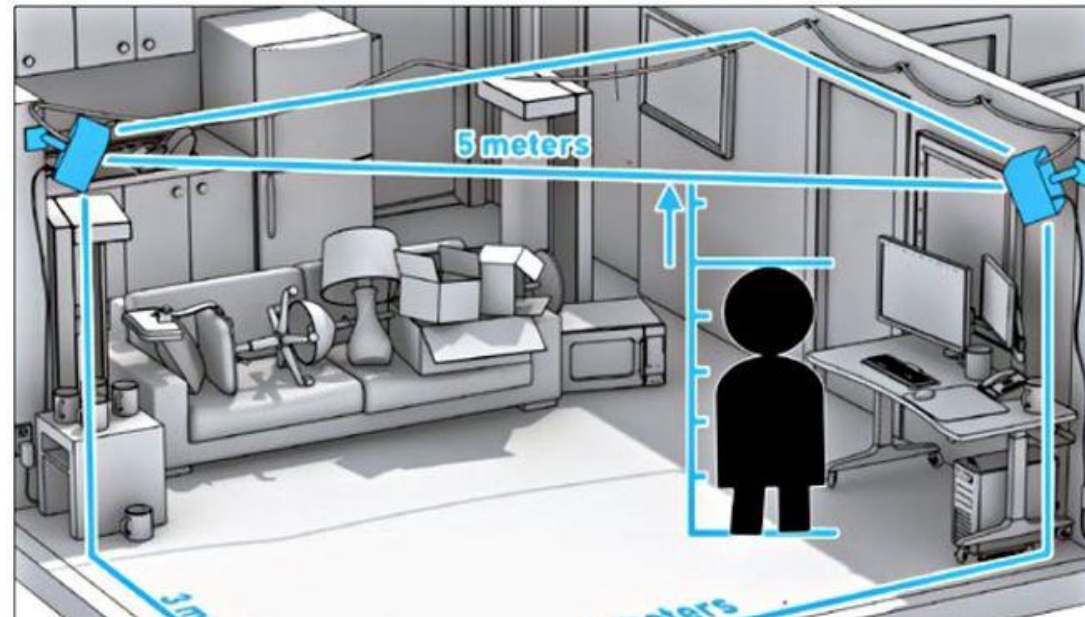
- ◆ Holograms can be created directly for raytracing based rendering
- ◆ The calibration information can be used to setup one ray per subpixel
- ◆ This can also be used in a fragment shader that implements a ray scene intersection test.
The open source cgv-framework supports this. Typical fragment shader main function implementation:

```
void main() {  
    vec3 ro[3], rd[3];  
    compute_sub_pixel_rays(ro, rd);  
    vec3 rgb, depth;  
    for (int c = 0; c < 3; c++)  
        rgb[c] = rayTracePixel(ro[c], rd[c], depth[c])[c];  
    if (!finalize_sub_pixel_fragment(rgb, depth))  
        discard;  
}
```


Stereo

IMMERSIVE SETUPS AND VISUALIZATION

Immersive Setups – VIVE



Immersive Setups – Walking



Immersive Setups - Climbing



what the climber
would have seen

what the
bystanders saw

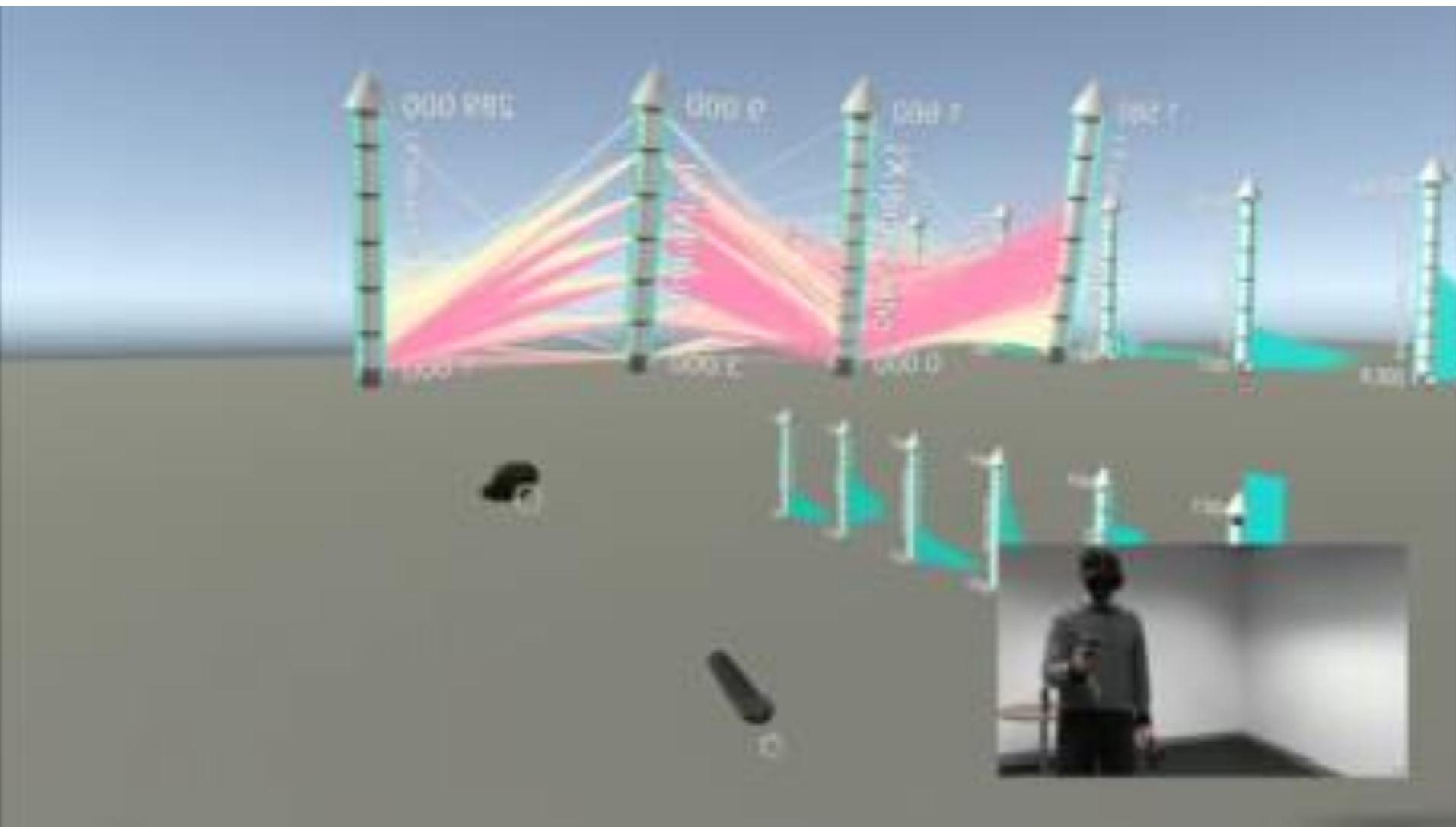
what the climber actually saw



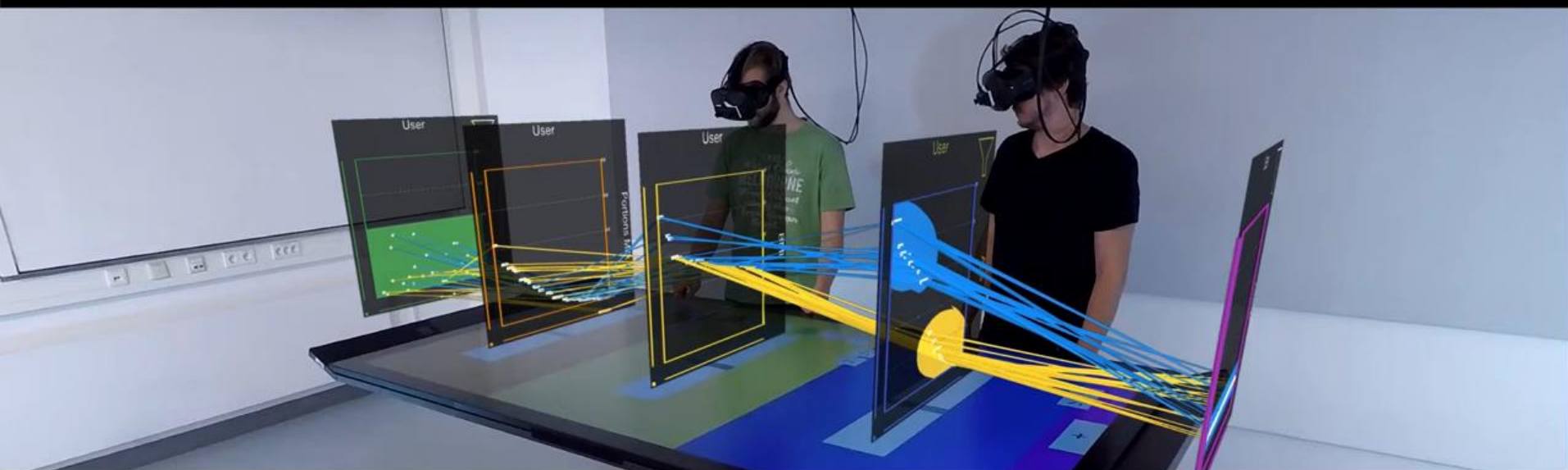
Climbing in a virtual mountain environment

https://www.dfki.de/fileadmin/user_upload/import/9045_climbing-mixed-reality.pdf

Immersive Setups - ImAxes



Clusters, Trends, and Outliers: How Immersive Technologies Can Facilitate the Collaborative Analysis of Multidimensional Data



Simon Butscher
HCI Group, University of Konstanz
simon.butscher@uni.kn

Sebastian Hubenschmid
HCI Group, University of Konstanz
sebastian.hubenschmid@uni.kn

Jens Müller
HCI Group, University of Konstanz
jens.mueller@uni.kn

Johannes Fuchs
Visualization Group, University of Konstanz
johannes.fuchs@uni.kn

Harald Reiterer
HCI Group, University of Konstanz
harald.reiterer@uni.kn