# CAVE viewing transforms 3 points

Most VR systems consist of a collection of display surfaces. For example, a simple CAVE might have 4 display surfaces (3 walls + floor). After choosing an appropriate coordinate system (e.g. the center of the CAVE), the geometry of each rectangular display surface can be represented with a point P and two orthogonal vectors **u**, **v** (see Figure 1). The four corners of the display are thus given by P, P+**u**, P+**v**, and P+**u**+**v**. We will assume that P corresponds to the origin of the OpenGL window coordinate system (i.e. the projector will map the pixel at (0,0) to P), and that **u** and **v** vectors correspond respectively to the x and y axes of the OpenGL window coordinate system.

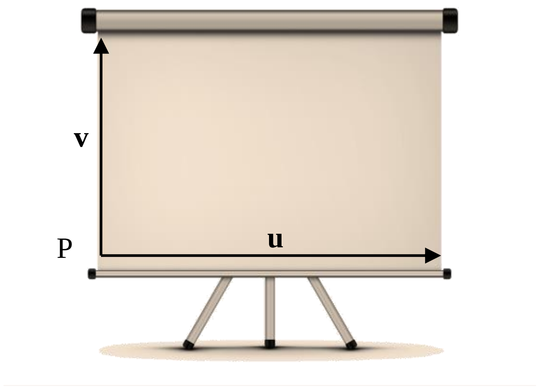

Figure 1: A display surface described by P, **u**, **v**.

We can put this information on a class representing a rectangular display surface.

In C++ code:

```
class DisplaySurface
{
	DisplaySurface(const Point& orig, const Vector& uvector, const Vector& vvector);
	Point origin;
	Vector u,v;
};
```

Or in JavaScript:

```
DisplaySurface = function(orig, uvector, vvector)
{
   this.origin = orig; // Vec3 - Origin of the display
   this.u = uvector;   // Vec3 - Horizontal vector
   this.v = vvector;   // Vec3 - Vertical vector
};
```

The four screens of a CAVE (3 walls + floor), assuming a size of 3m x 3m, can be defined by:

```
DisplaySurface front(Point(-150,-150,-150), Vector(300, 0, 0),  Vector( 0, 300, 0));
DisplaySurface left (Point(-150,-150, 150), Vector(0, 0, -300), Vector( 0, 300, 0));
DisplaySurface right(Point( 150,-150,-150), Vector(0, 0,  300), Vector( 0, 300, 0));
DisplaySurface floor(Point(-150,-150, 150), Vector(300, 0,  0), Vector( 0, 0,-300));
```

where all coordinates/components are given in cm with respect to a coordinate system at the middle of the CAVE (see Figure 2).

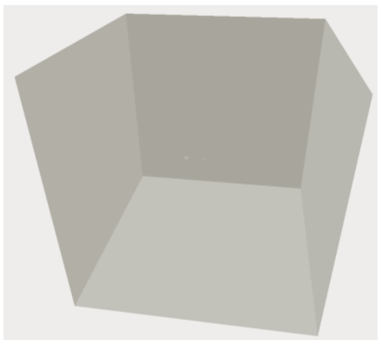

Figure 2: Simple CAVE example

In this exercise, we ask you to write a function (C++ example)

```
QMatrix4x4 viewingMatrix(const DisplaySurface& display, const Point& eye);
```

that, given a display surface and the (x,y,z) coordinates of the eye (both expressed w.r.t. the chosen coordinate system), returns the 4x4 viewing matrix to be used to render a scene onto the display surface from the given eye position.  You might want to use a lookAt() method.

For example, for eye = (50, 20, 100), and the front/left/right/floor displays above, your code should return the following matrices:

```
 1         0         0       -50
 0         1         0       -20
 0         0         1      -100
 0         0         0         1

 0         0        -1       100
 0         1         0       -20
 1         0         0       -50
 0         0         0         1

 0         0         1      -100
 0         1         0       -20
-1         0         0        50
 0         0         0         1

 1         0         0       -50
 0         0        -1       100
 0         1         0       -20
 0         0         0         1
```

## CAVE projection transforms 4 points

Write a function

```
QMatrix4x4 projectionMatrix(const DisplaySurface& display, const Point& eye, float znear,
float zfar);
```

that, given a display surface, the (x,y,z) coordinates of the eye, and znear, zfar clipping planes, returns the 4x4 projection matrix to be used to draw a scene onto the display surface from the given eye position. You might want to use frustum() method.

For example, the front/left/right/floor displays above, eye = (0, 0, 0), znear = 1 and zfar = 1000, your code should compute the matrices below; we also show the (left, right, bottom, top, near, far) parameters for computing the viewing frustum:

```
-1 1 -1 1 1 1000

        1         0         0         0
        0         1         0         0
        0         0    -1.002    -2.002
        0         0        -1         0
```

(for this eye position the other three matrices are identical).

For eye = (50, 20, 100), znear = 0.1, zfar = 100, your code should return:

```
Front wall:
-0.08 0.04 -0.068 0.052 0.1 100

  1.66667         0 -0.333333         0
        0   1.66667 -0.133333         0
        0         0    -1.002   -0.2002
        0         0        -1         0

Left wall:
-0.025 0.125 -0.085 0.065 0.1 100

  1.33333         0  0.666667         0
        0   1.33333 -0.133333         0
        0         0    -1.002   -0.2002
        0         0        -1         0

Right wall:
-0.25 0.05 -0.17 0.13 0.1 100

 0.666667         0 -0.666667         0
        0  0.666667 -0.133333         0
        0         0    -1.002   -0.2002
        0         0        -1         0

Floor:
-0.117647 0.0588235 -0.0294118 0.147059 0.1 100

  1.13333         0 -0.333333         0
        0   1.13333  0.666667         0
        0         0    -1.002   -0.2002
        0         0        -1         0
```

## CAVE simulator 3 points

Implement (recommendation: use the provided example in WebGL) a simple viewer to simulate the projection of an arbitrary scene onto a collection of display surfaces (see Figure 3).

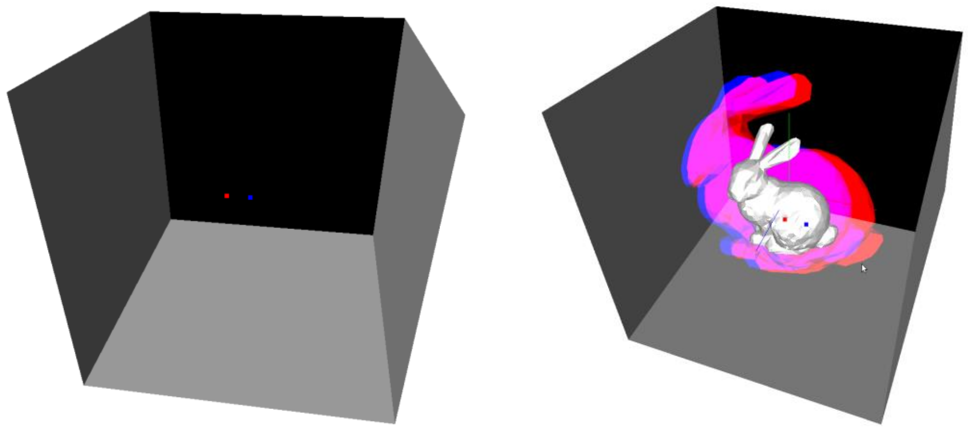
Figure 3: CAVE simulator

The features to be implemented, with respect to a simple 3D viewer, are summarized below:

- You should define a collection of display surfaces, and (x,y,z) positions for the user's left and right eye. For example, you could initialize leftEye = (-3, 20, 50), and rightEye = (3, 20, 50). Notice that this would result in an iod of 6 cm.
- Draw the display surfaces as quads (Figure 3).
- Draw the position of left and right eyes (e.g. as GL_POINTs or as spheres)
- Compute a suitable modeling transform for the scene. For example, centered at the centroid of the display surfaces, and scaled so that it fits inside the bounding box of the display system.
- Draw the projection of the scene onto each display surface, for the left (in red) and right (blue) eyes. The best option is to use multiple rendering passes: first, render the scene (using the matrices returned by your viewingMatrix() and projectionMatrix() methods) onto a texture (you need one texture for each display surface). Then, draw the display surfaces as texture quads. When rendering to texture, you might use glColorMask() to enable/disable writing on RGBA buffers, so that the view from the left eye is written only on red buffers and that for the right eye is written only on blue (maybe also green) buffers.
- Provide some interaction with the scene. The user should be able to move the left/right eyes, change iod, and move/scale the scene.

Notice that a subset of the features above are already implemented in the provided WebGL example.