CS 432 – Interactive Computer Graphics

Lecture 4 – 3D Viewing

Reading

- Angel
  - Chapters 3-4
- Red Book
  - Chapter 5, Appendix E

From Objects to the Screen

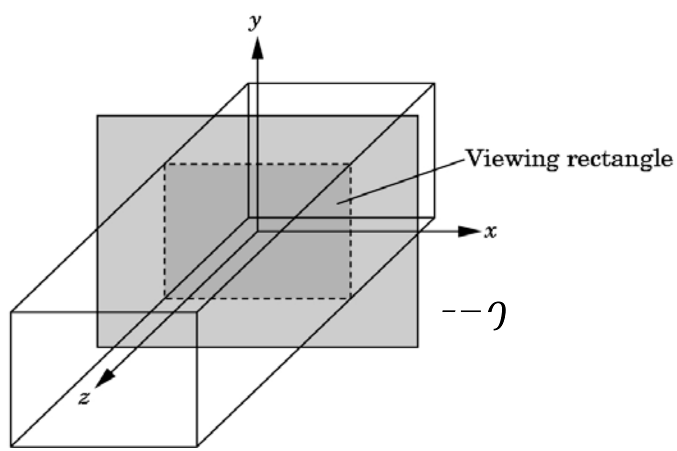
- We started off be specifying objects via vertices that define primitives in 2D space.
  - These vertices were all in the range $-1 \leq x, y \leq 1$
- We added the ability to specify in OpenGL a *viewport* within the window to map this range to.
- In 3D space we can also specify a z-coordinate so that it's in view if $-1 \leq z \leq 1$

From Objects to the Screen

- In 2D space we call the area $-1 \leq x, y < 1$ the *clipping window*
  - Everything outside of this is clipped
- In 3D space, we call the *volume* $-1 \leq x, y, z < 1$ the *clipping volume.*
  - Everything outside of this volume is clipped

From Objects to the Screen

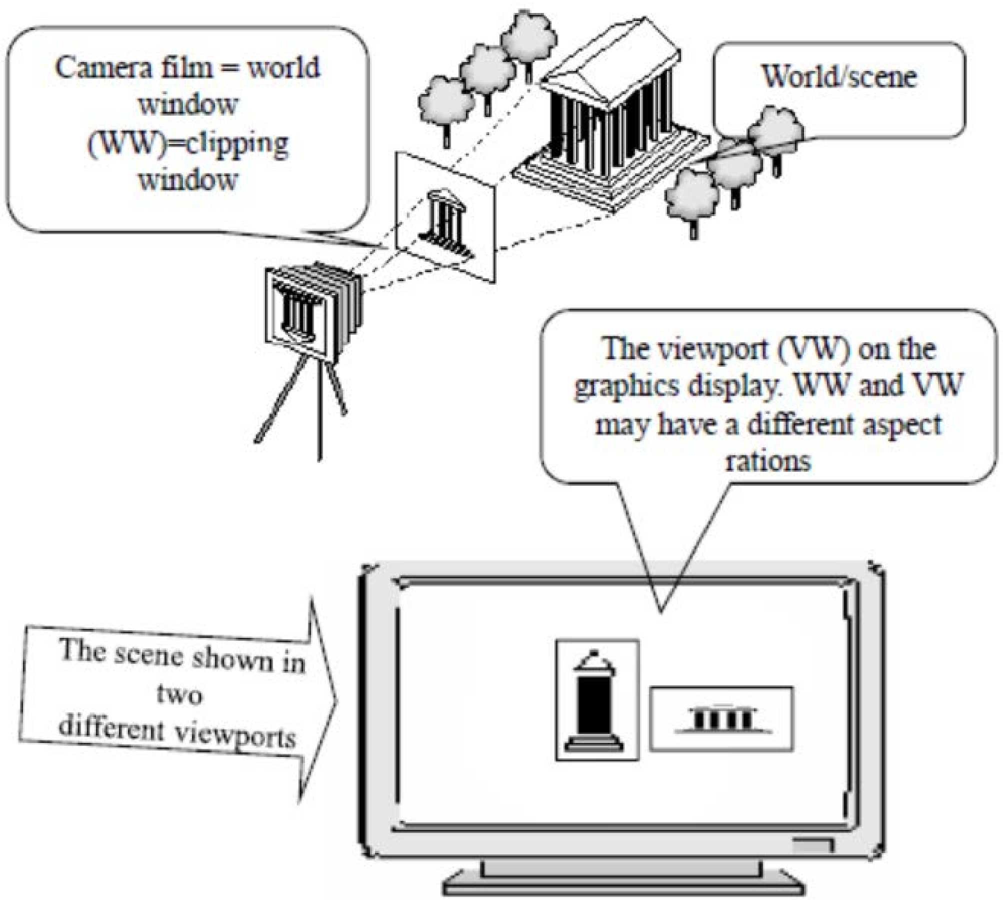
- Conceptually, the job of a virtual camera is to *project* points in the clipping volume onto the clipping area such that $z = 0$.
- These resulting points $(x, y, 0)$ are now considered in *camera coordinates*



From Objects to the Screen

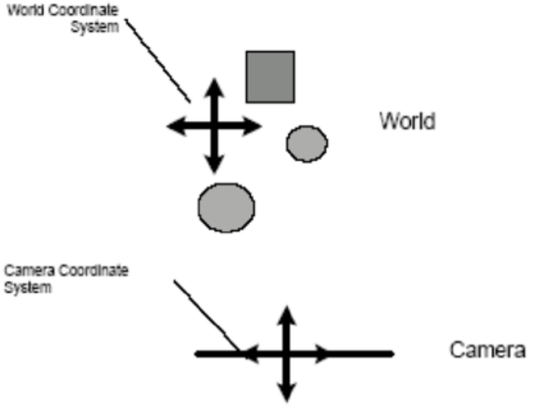
- Once the objects are in camera coordinates they must be mapped to *screen coordinates* (i.e. pixels)
- We can even restrict what area of the screen to show
  - This is called the *viewport*

3D Viewing



Camera film = world window (WW)=clipping window
World/scene
The viewport (VW) on the graphics display. WW and VW may have a different aspect rations
The scene shown in two different viewports

OpenGL Coordinate Systems

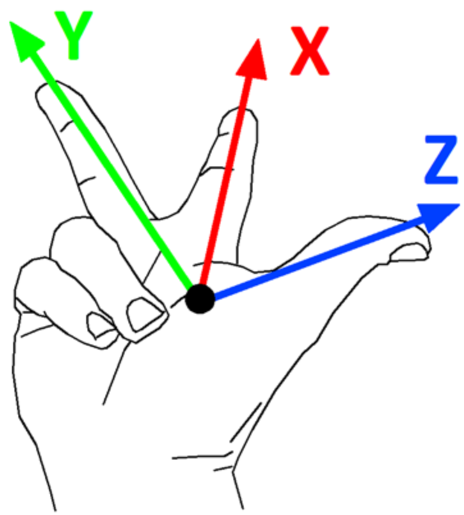


- We now have four coordinate systems!

1. Model Coordinate System
   - Where we build our objects
2. World Coordinate System
   - Where we place our objects via a model matrix transformation
3. Camera Coordinate System
   - Where the objects appear on the camera “film”
4. Window Coordinate System
   - Where the objects on the film appear on the screen (pixels)

OpenGL Coordinate Systems

- Remember, when we were talking about forwards and backwards facing polygons we talked about the *right hand rule*.
- This also defines the orientation of these coordinate systems
- This result is that the positive z-axis points towards the viewer center-of-projection (COP)

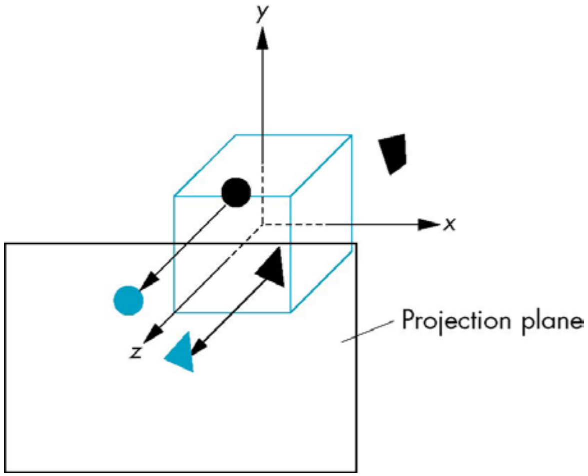


Computer Viewing

- We can simulate viewing the world by transforming objects using the following pipeline:
  1. Position the objects
     - Via their respective model matrices
  2. Positioning the camera
     - Setting the view matrix
  3. Selecting the projection type/parameters
     - Setting the projection matrix
- Then we clip the transformed objects against some default camera view window or volume.

Default Coordinate Systems

- By default the object, world, and camera coordinate systems are the same resulting in the camera located at the shared origin and pointing in the negative z-direction
- The default camera view volume is a cube with sides of length 2 centered at the origin



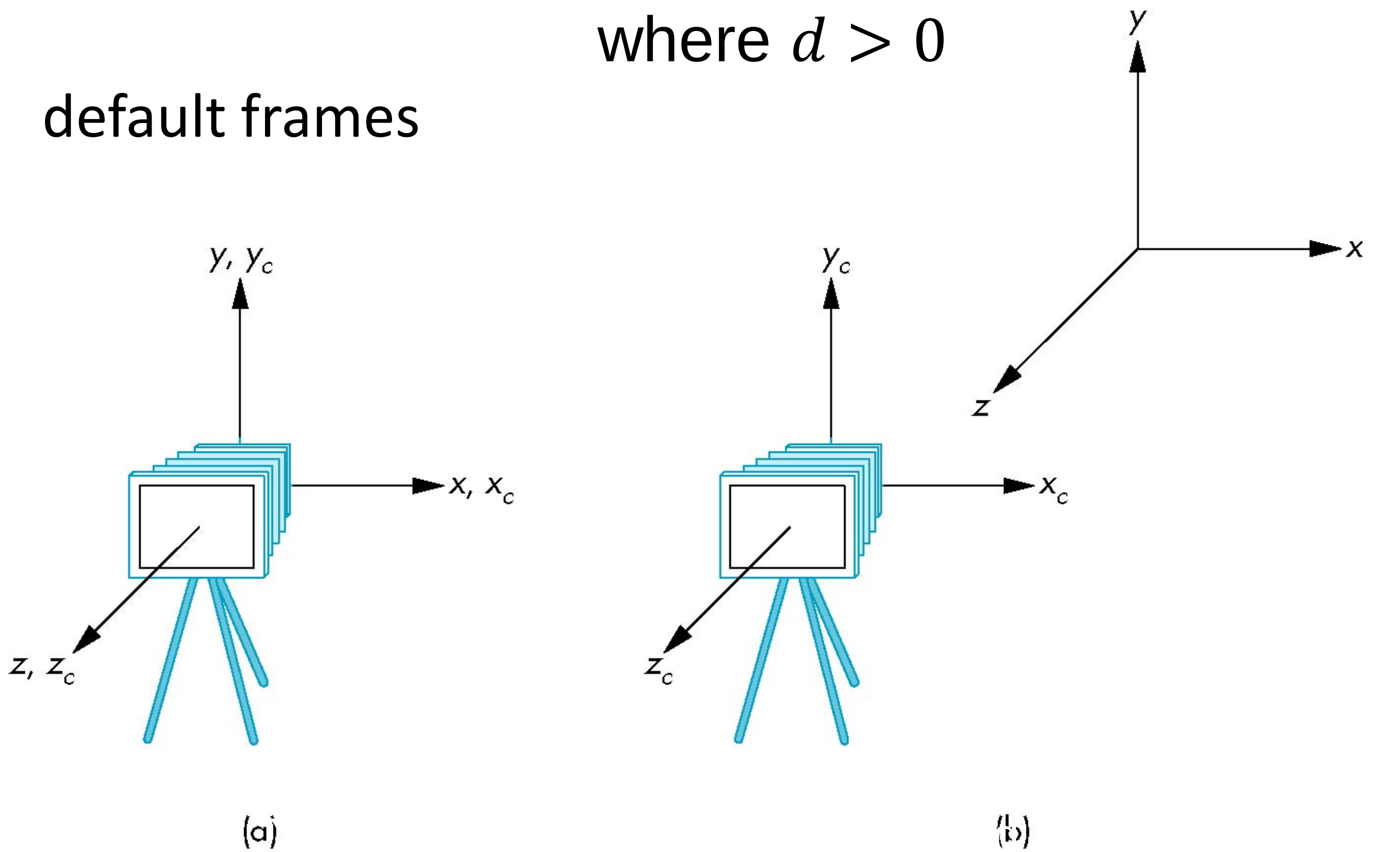
Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these view are equivalent and are determined by the model-view matrix
  - Want a translation: $Translate(0.0, 0.0, -d)$
  - Where $d > 0$

Moving Camera back from Origin

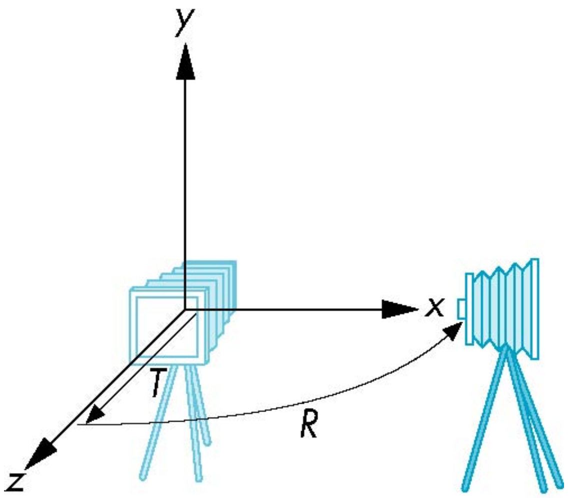
frames after translation by $-d$, where $d > 0$

default frames



Moving the Camera

- We can move the camera to any desired position and orientation by a sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from the origin
  - Transformation matrix $M_{view} = TR$



OpenGL Code

- Remember that the last transformation specified is the first applied
  - `mat4 t = Translate(0.0, 0.0, -d);`
  - `mat4 ry = RotateY(90.0);`
  - `mat4 mView = t*ry;`
- So now we have both a model matrix for an object and a view matrix.
- We have options:
  - Send both independently to the shader and in the shader compute:
    ```
    gl_Position = view_matrix*model_matrix*vPosition;
    ```
  - Compute `view_matrix*model_matrix` in the OpenGL program and send that single matrix to the shader program:
    ```
    modelview_matrix = view_matrix*model_matrix;  //OpenGL
    gl_Position = modelview_matrix*vPosition; //GLSL
    ```
  - And of course you might just update the vertices right in OpenGL and send them to the shader.

Moving the Camera on the GPU

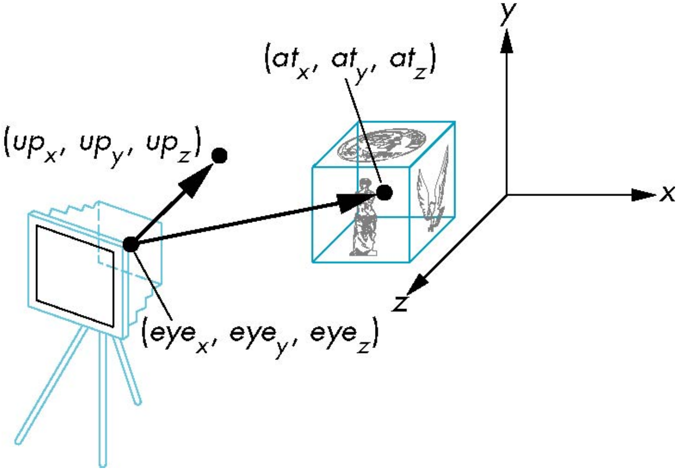
- Your choice may depend on how often things change
  - Never?
  - Model often, camera rarely?
  - Camera rarely, model often?
  - Both often?
- Remember, we most likely need to transpose our matrices when we send them to the shaders:

Transpose

```
GLuint model_view_loc =
    glGetUniformLocation(program,"modelview_matrix");
glUniformMatrix4fv(modelview_loc,1,GL_TRUE,modelview_matrix);
```

Viewing APIs

- Specifying/computing the translation and rotation(s) may not be easy or natural.
- To simplify things, there are several APIs that will provide the model-view matrix for different parameters.
- One of the more popular (and supported in OpenGL) is the Look-At API
- But first we need to review some geometry!

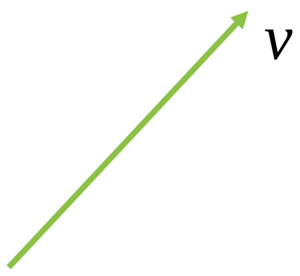


Points

- We already know that to defined objects we need to provide vertex locations.
- Geometrically we can call these locations in space *points*.
- In graphics there are at least three other important geometric objects:
  1. Vectors
  2. Lines
  3. Planes

Vectors

- A vector is a quantity with two attributes
  - Direction
  - Magnitude
- Note: They have not actual position in space
- Examples include
  - Force
  - Velocity
  - Directed line segments
    - Important for graphics!

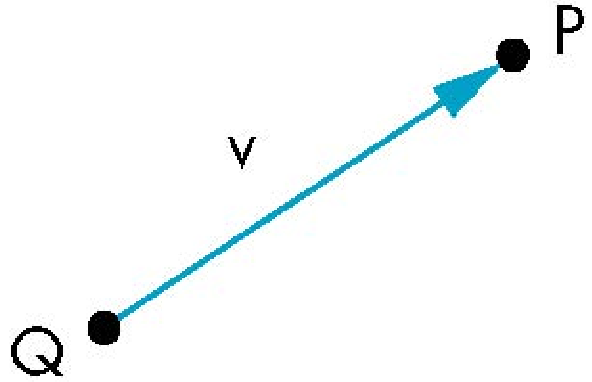


Vectors

- A vector can be defined given two points via subtraction:

$$v = P - Q$$

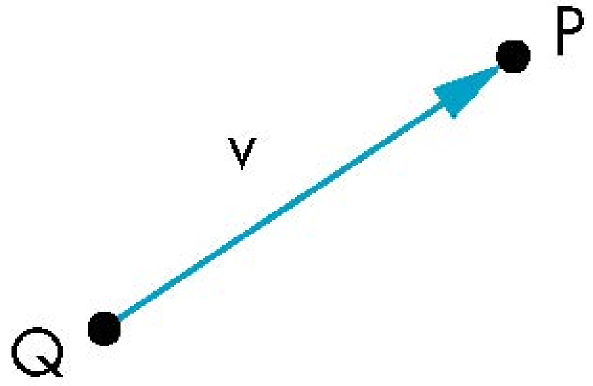
- Here we say the vector goes in the direction *from* point Q *to* point P.



Vectors

- Also given a point and a vector we can arrive at a new point via addition:

$$P = v + Q$$



Vector Operations

- There are additional properties of vectors:
  - Every vector has an inverse
    - Same magnitude but points in the opposite direction
  - Every vector can be multiplied by a scalar
    - Same direction (assuming non-negative magnitude), different magnitude
  - The sum of any two vectors is a vector
    - Use the head to tail axiom

Vectors in Homogenous Coordinates

- We know in 3D we specify vertices (points) via a 4D "vector" (confusing?) with the 4th coordinate equal to one:

$$P = \left(P_x, P_y, P_z, 1\right)$$

- Via vector arithmetic, we then define a vector in 3D homogenous coordinates as a 4D vector with the 4th coordinate equal to zero:

$$v = \left(v_x, v_y, v_z, 0\right)$$

Vectors in Homogenous Coordinates

$$v = \left(v_x, v_y, v_z, 0\right)$$

- The magnitude of this vector is:

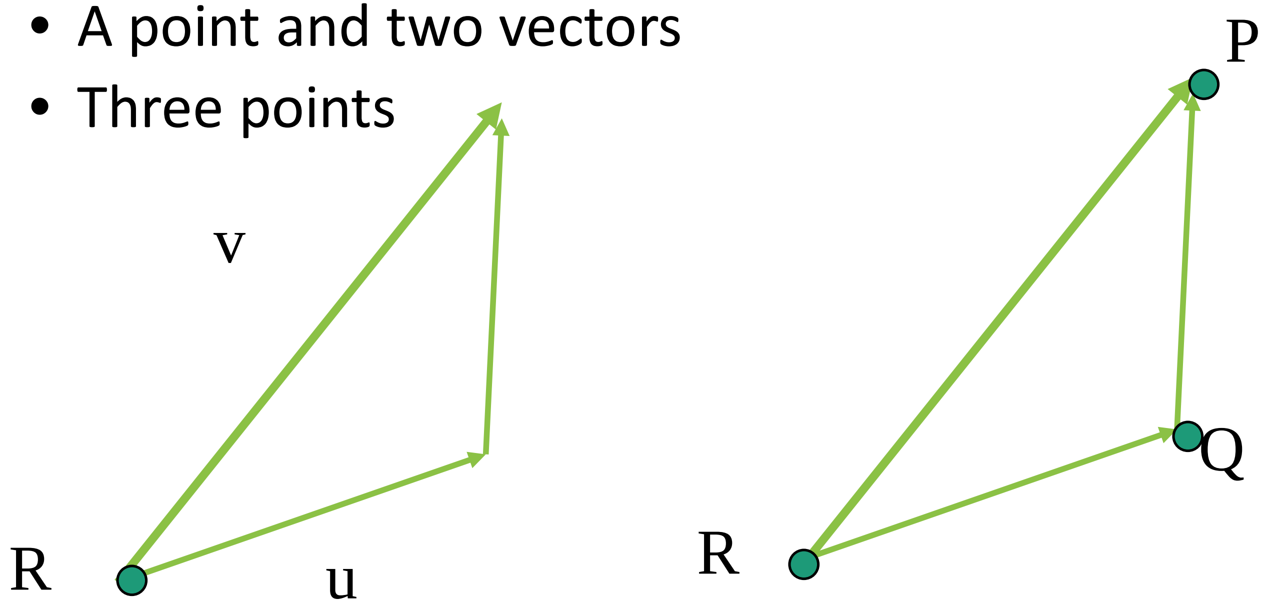
$$||v|| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

- Usually we specify the direction by making the vector have a length of one (unit length, or a normalized):

$$\frac{v}{||v||}$$

Planes

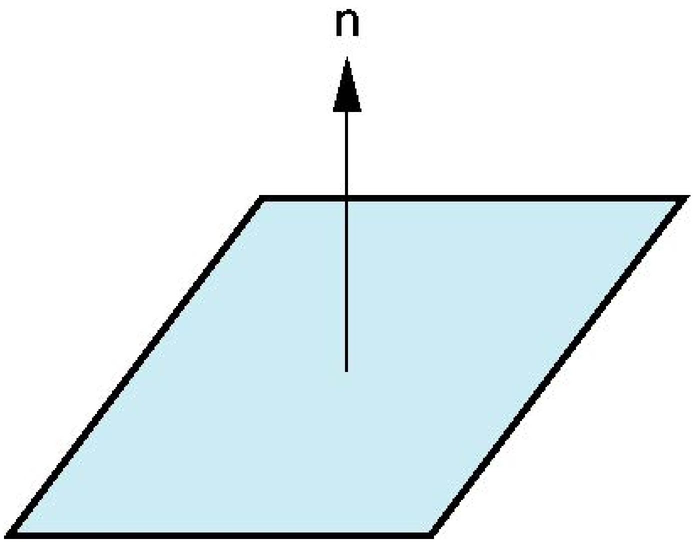
- A plane can be define either by
  - A point and two vectors
  - Three points



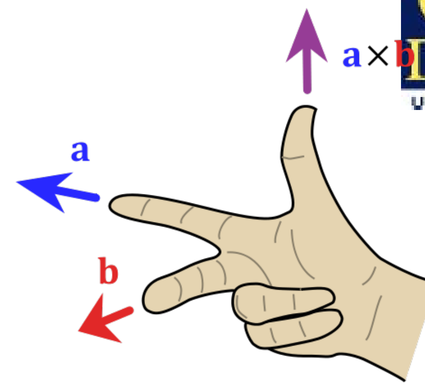
Normals

- Every plane has a vector n *normal* (perpendicular, orthogonal) to it
- Given two non-collinear vectors on a plane, u and v, we can use the **cross product** to find a plane's normal:

$$n = u \times v$$



Cross Product



- The cross product is defined as

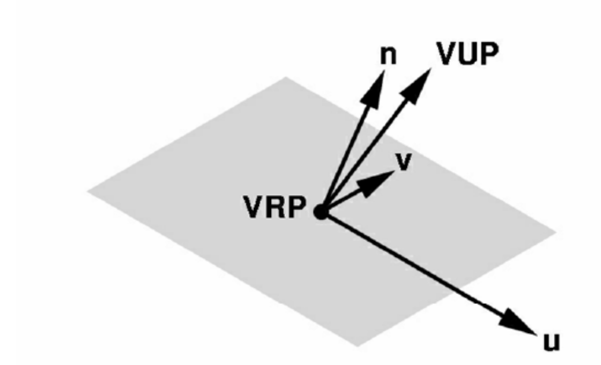
$$a \times b = \|a\| \|b\| \sin(\theta)\, n$$

  - Where n is the normal.
- So by solving for the cross product we get the normal multiplied by a scalar.
- Given two vectors u, v we can compute the cross product as

$$u \times v = \left((u_y v_z - u_z v_y), (u_z v_x - u_z v_z), (u_x v_y - u_y v_x), 0 \right)$$

- Fortunately both OpenGL and GLSL also give us a *cross* function
- **But be careful…**
  - Always make sure your 4th component is 0 so that it's treated as a vector!

Viewing APIs



- Ok back to a viewing API…..
- We can define a camera's location and orientation by specifying:
  - The position of the camera, often called the **eye** or the *view reference point* (VRP)
  - A vector that specifies the normal of the camera's view plane. This is often called the view plane normal (VPN, n in the figure below)
    - It is in the viewing direction In OpenGL the VPN is in the direction opposite the one in which the camera is looking
    - The direction opposite the VPN is often called the **at** vector.
  - Another vector, called the *view-up vector*, is in a direction specifying which is the approximate "**up**" direction for the camera

The LookAt Function

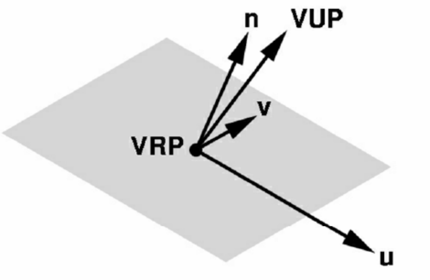
- The GLU library contained a function *`gluLookAt`* to form the required model-view matrix through a simple interface
- Replaced by `LookAt()` in `mat.h`
    - These too will have to be transposed when sent to the shader programs.

```
mat4 view_matrix = LookAt(vec4 eye, vec4 at, vec4 up);
```

Angle.h Lookat

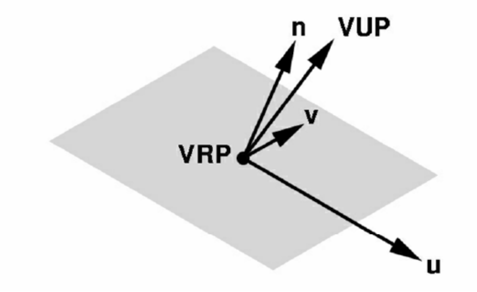
```
mat4 LookAt(const vec4& eye, const vec4& at, vec4& up){
    vec4 n = normalize(eye-at);
    vec4 u = normalize(cross(up,n));
    vec4 v = normalize(cross(n,u));
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u,v,n,t);
    return c*Translate(-eye);
}
```

The LookAt Function



- If we want to update our camera (move it, rotate it, etc..) we'll want to keep track of some things…
  - The location of the camera： eye
  - Three vectors that define the view plane:
    1. The view plane normal (opposite the at direction), n
    2. The "right" direction， u
    3. For convenience, we'll keep track of the third vector orthogonal to the other two (approximately the up direction)， v

The LookAt Function



- From these vectors you should easily be able to compute the *`at`* and *up* vectors needed for the `LookAt` API
  - `at = eye-n;`
  - `up = v;`

Other Viewing APIs

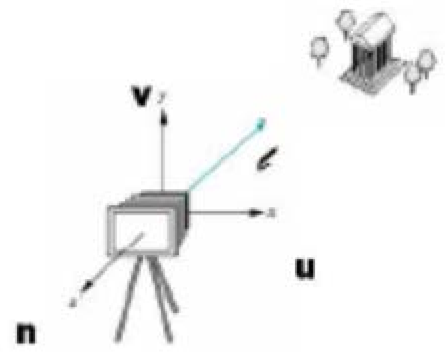
- The `LookAt` (and `gluLookAt`) function is one of many possible APIs for positioning the camera
- Others include
  - View reference point, view plane normal, view up
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles

Interface: Flying Cameras

- Often we like to use the keyboard to navigate around the "world"
- This is akin to moving the camera (and re-orienting it)
- If we use the Look-At API on each relevant keystroke we must
  - Compute the new eye point
  - Compute the new at point
  - Compute the new up vector

Interface: Flying Cameras

- It may be conceptually easier to move the camera according to another common API: pitch/yaw/roll
  - Pitch: rotate around current u axis
  - Roll: rotate around current n axis
  - Yaw: rotate around current v axis
- And update our coordinate system



Interface: Flying Cameras

- From our Look-At API we already have `v` (up) and `n=e-a`.
  - So `a=e-n`
  - And we can get `u=normalize(cross(v,n));`
- Pitch (rotate around `u` axis) by α radians
  - $u' = u$
  - $v' = \cos(\alpha)v - \sin(\alpha)n$
  - $n' = \sin(\alpha)v + \cos(\alpha)n$
- We can do this similarly for roll and yaw
- Notes:
  - Use the *normalize* function when necessary to get unit length vectors
    - **Again, be careful that your 4th component is zero before doing this.**
  - Make sure to call the `glutPostRedisplay()` after changing view matrix so that it is redrawn immediately

Interface: Flying Cameras

- The last thing we may want to do is move allow (or away from) the direction of $-\mathrm{n}$
- We can do this by change the eye and at points by the same amount along the n vector
  - $e = e + \beta n$
  - $a = a + \beta n$
- Interface:
  - X→Pitch down
  - x→Pitch up
  - C→Yaw clockwise in un plane
  - c→Yaw counter-clockwise in un plane
  - Z→Roll clockwise in the uv plane
  - z→Roll counter-clockwise in the uv plane
  - GLUT_KEY_UP → Move towards at point
  - GLUT_KEY_Down → Move away from at point
- NOTE: GLUT_KEY_UP/DOWN are "special keys" and are called via the `glutSpecialFunc` callback

A Camera Class

- For convenience you'll likely want to create a `Camera` class.
- This can store:
  - Current `view_matrix`
  - `eye`
  - `u,v,n`
- And can do things like
  - Get the `view_matrix`
  - Update `eye, u, v, n` based on interactions.
- And we'll add more (the projection matrix) in a minute…
- And when it comes time to draw an object, just pass it the current `view_matrix` so it can either use it or pass it to the shader.