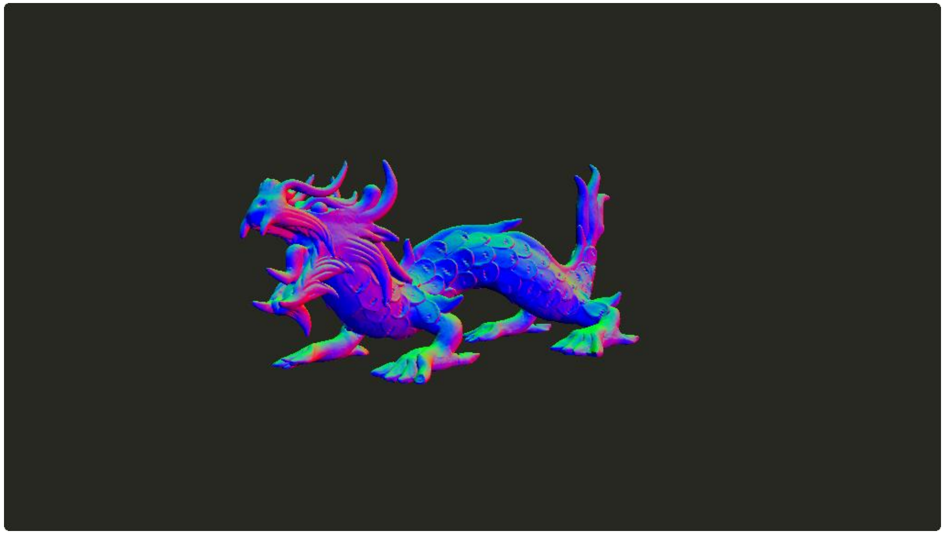
Camera in OpenGL

Computer Graphics
Instructor: Sungkil Lee

Today

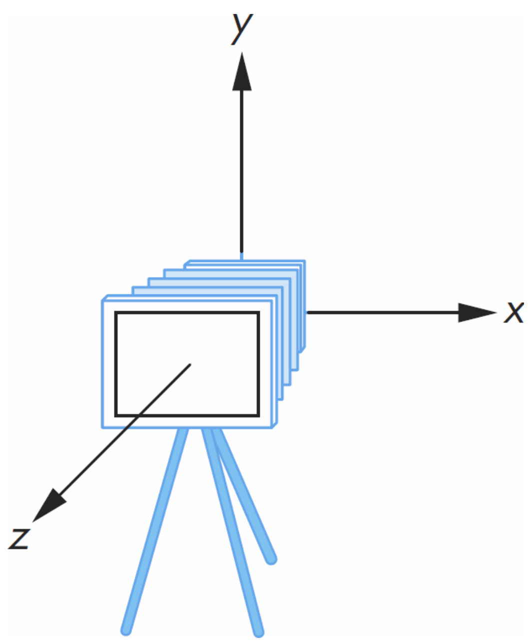
- **Viewing and projection in OpenGL**
  - How to implement look_at() and perspective() functions.
  - The full details are covered in the theory lecture.
- **Virtual trackball**
  - How to implement mouse-based interaction
- **Virtual trackball extension**
  - Hints for panning and zooming

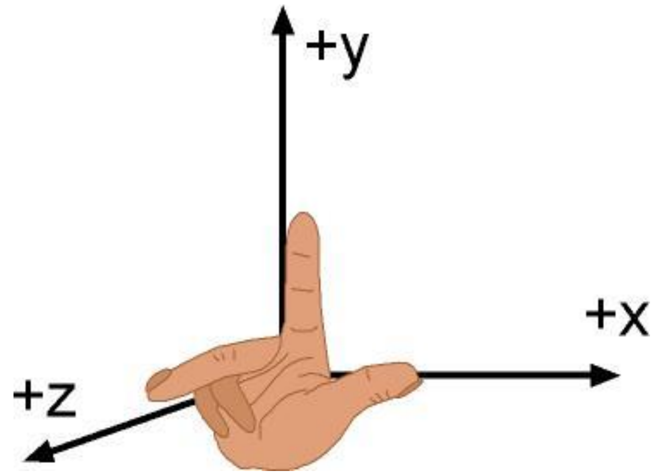


Prerequisites Revisited

Recall: OpenGL Default Camera

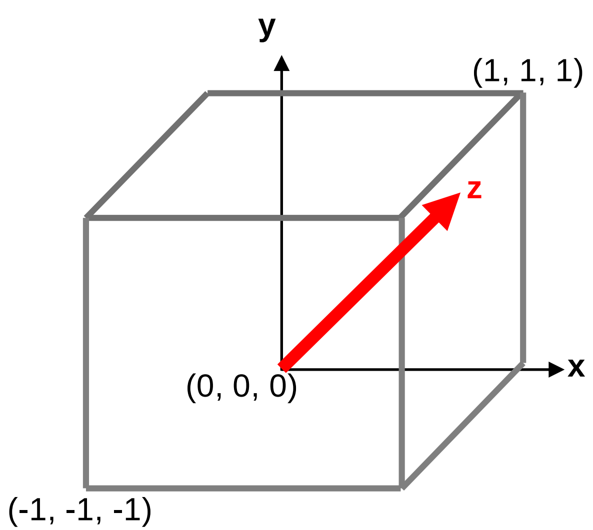
- **Located at origin and directs in the negative z direction.**
  - camera coordinate systems (frames) use RHS convention.
  - Initially the object and camera frames are identical.
  - Default model-view matrix is an identity.





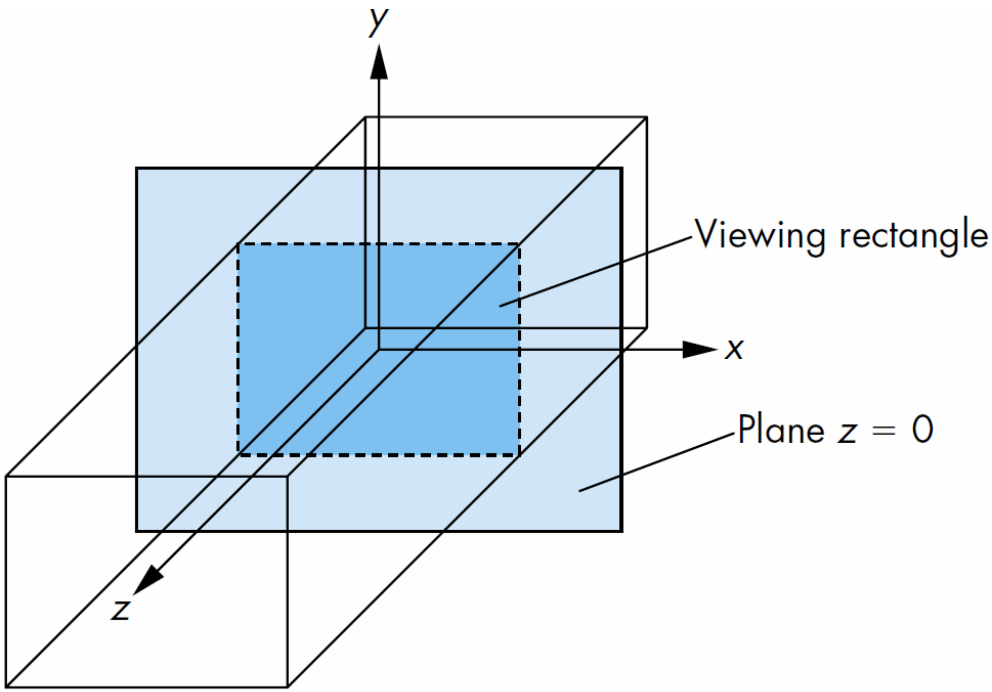
Recall: Canonical View Volume in NDC

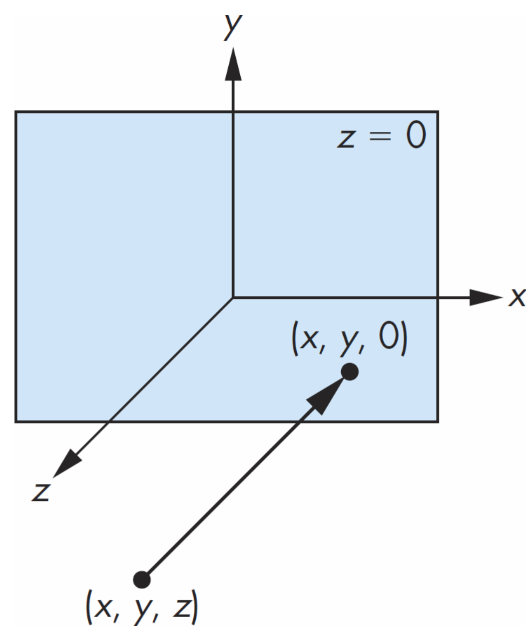
- **Canonical view volume in normalized device coordinates:**
  - Default view volume: cube with sides of length 2 centered at the origin
    - $right = top = far = 1$
    - $left = bottom = near = -1$
  - Default projection matrix is an identity matrix.
    - However, we need to negate z for correct depth test.



OpenGL Default Camera

- **Default projection:**
  - ***orthographic projection*** to $z = 0$ (but, actually we don't set z=0)
  - Objects outside the default view volume get invisible (clipped out).





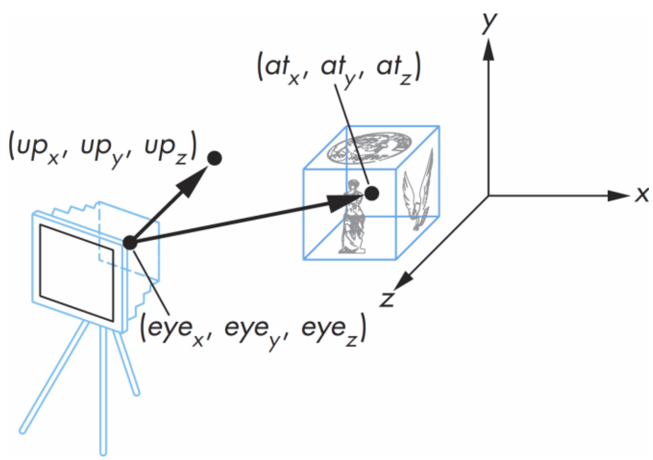
Viewing in OpenGL

Recall: look_at() Method

- **look_at () method**

```
mat4 look_at(eye, at, up)
```

- **Viewing specification with (eye, at, up)**
  - eye: a camera's location
  - at: the center of the destination position to be viewed
  - up: upward direction of the camera frame

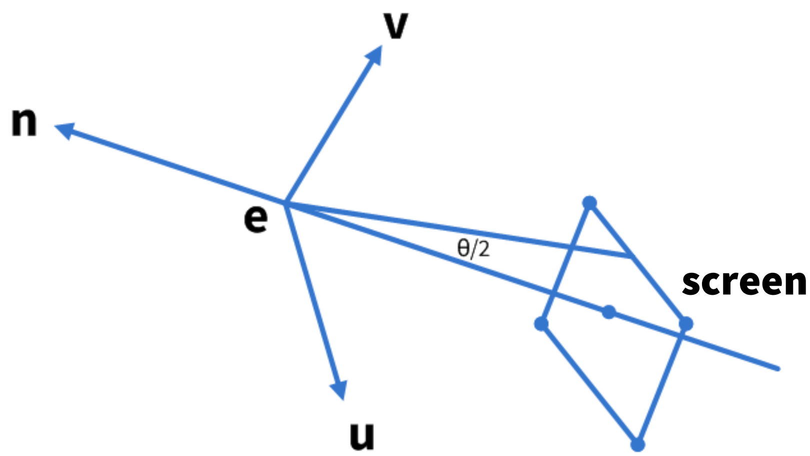


Recall: look_at() Method

- **eye, at, and up can define a camera *frame*, which has**
  - the origin at eye
  - three basis vectors, $\boldsymbol{n}$, $\boldsymbol{u}$, and $\boldsymbol{v}$, defined as:

- **Thus, the viewing transformation can be a *change of frame*,**
  - which changes from a world frame to a camera frame.
  - We can do the view transformation with **4×4 lookat matrix**.

$\boldsymbol{n}$ = normalize($\boldsymbol{eye} - \boldsymbol{at}$)
$\boldsymbol{u}$ = normalize($\boldsymbol{up} \times \boldsymbol{n}$)
$\boldsymbol{v}$ = normalize($\boldsymbol{n} \times \boldsymbol{u}$)



look_at() implementation

$$\mathbf{RT}(-eye) = \begin{bmatrix} u1 & u2 & u3 & 0 \\ v1 & v2 & v3 & 0 \\ n1 & n2 & n3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye.x \\ 0 & 1 & 0 & -eye.y \\ 0 & 0 & 1 & -eye.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
static mat4 look_at( const vec3& eye, const vec3& at, const vec3& up )
{
    return mat4().set_look_at( eye, at, up );
}

mat4& set_look_at( vec3 eye, vec3 at, vec3 up )
{
    set_identity();
    // define camera frame.
    vec3 n = (eye - at).normalize();
    vec3 u = up.cross(n).normalize();
    vec3 v = n.cross(u).normalize();
    // calculate lookAt matrix: a combined form of RT(-eye)
    _11 = u.x;  _12 = u.y;  _13 = u.z;  _14 = -u.dot(eye);
    _21 = v.x;  _22 = v.y;  _23 = v.z;  _24 = -v.dot(eye);
    _31 = n.x;  _32 = n.y;  _33 = n.z;  _34 = -n.dot(eye);
    return *this;
};
```

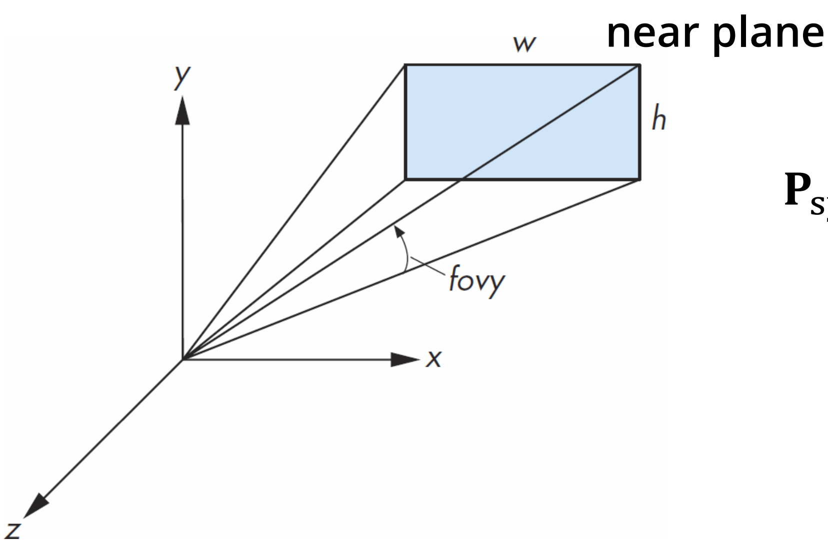
Perspective Projection in OpenGL

Perspective Projection in OpenGL

- **We can apply projection using matrix multiplication.**

```
mat4 perspective(fovy, aspect, near, far)
```

  - perspective() returns a matrix for symmetric perspective projection.
  - $fovy$ (field of view) and $aspect$ (width/height of sensor/window):



$$\mathbf{P}_{\text{sym}} = \begin{bmatrix} \cot\left(\frac{fovx}{2}\right) & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$t = n * \tan(\frac{fovy}{2}) \qquad r = t * aspect \qquad \cot\left(\frac{fovx}{2}\right) = \cot\left(\frac{fovy}{2}\right) / aspect$$

perspective()

- **Note:**
  - We cannot use reserved ‘near’ and ‘far’ for variable names in C++.

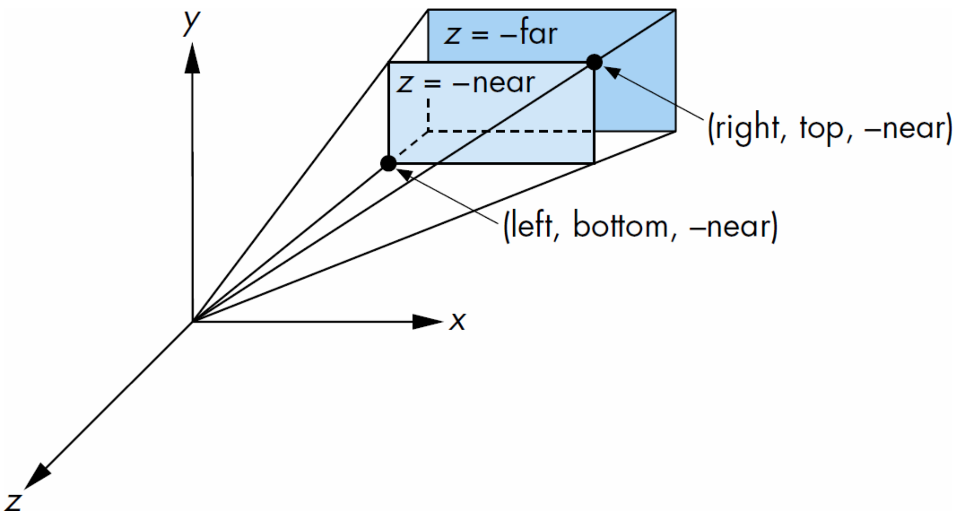
```
static mat4 perspective(float fovy, float aspect, float dnear, float dfar)
{
    return mat4().set_perspective(fovy, aspect, dNear, dFar);
}

mat4& set_perspective(float fovy, float aspect, float dnear, float dfar)
{
    set_identity();
    _22 = 1 / tan(fovy / 2.0f);
    _11 = _22 / aspect;
    _33 = (dnear + dfar) / (dnear - dfar);
    _34 = (2 * dnear * dfar) / (dnear - dfar);
    _43 = -1;
    _44 = 0;
    return *this;
}
```

frustum() for General Perspective Projection

- **The general perspective projection matrix**

```
mat4 frustum(left, right, bottom, top, near, far)
```

  - Parameters: r(right), l(left), b(bottom), t(top), n(near), f(far)



$$\mathbf{P} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Camera in OpenGL: Programming Example

Example of Camera Structure

- **Structure for camera**

```
struct camera
{
    vec3 eye = vec3( 0, 30, 300 );  // position of camera.
    vec3 at = vec3( 0, 0, 0 );      // position where the camera looks at
    vec3 up = vec3( 0, 1, 0 );
    mat4 view_matrix;               // result of look_at function.

    float fovy = PI/4.0f;           // in radian
    float aspect;                   // window_size.x / window_size.y
    float dnear = 1.0f;
    float dfar = 1000.0f;
    mat4  projection_matrix;
};
```

Update()

```
void update()
{
   // update projection matrix
   cam.aspect = window_size.x/float(window_size.y);
   cam.projection_matrix = mat4::perspective(
      cam.fovy, cam.aspect, cam.dnear, cam.dfar );

   ...

   // update uniform variables in vertex/fragment shaders
   GLint uloc;
   uloc = glGetUniformLocation( program, "view_matrix" );
   if(uloc>-1) glUniformMatrix4fv( uloc,1,GL_TRUE, cam.view_matrix );
   uloc = glGetUniformLocation( program, "projection_matrix" );
   if(uloc>-1) glUniformMatrix4fv( uloc,1,GL_TRUE,cam.projection_matrix);
}
```

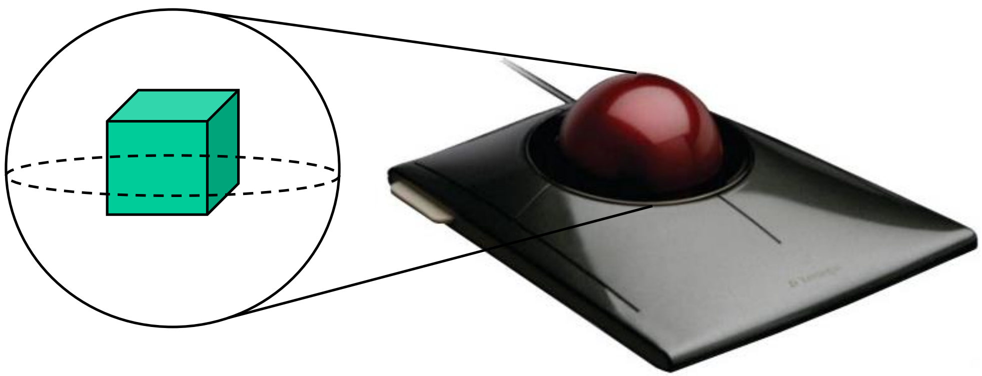
Vertex Shader: trackball.vert

```
// vertex attributes and output
in vec3 position;
in vec3 normal;
in vec2 texcoord;
out vec3 norm;

// matrices
uniform mat4 model_matrix;
uniform mat4 view_matrix;
uniform mat4 projection_matrix;

void main()
{
   vec4 wpos = model_matrix * vec4(position,1); // w: world
   vec4 epos = view_matrix * wpos; // e: eye or camera
   gl_Position = projection_matrix * epos;

   // pass eye-space normal to fragment shader
   norm = normalize(mat3(view_matrix*model_matrix)*normal);
}
```

Virtual Trackball

Physical Trackball

- **Trackball is an “upside down” mouse.**
  - Imagine the objects are rotated along with an imaginary sphere.

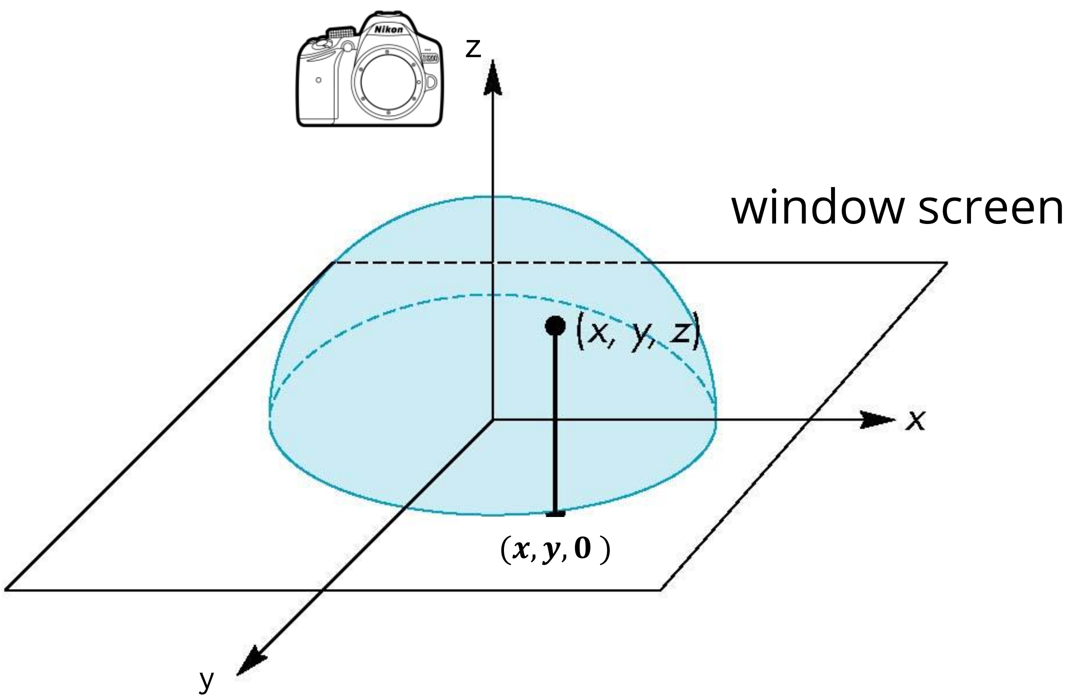


  - Allow a user to define 3D rotation using touch (mouse clicks) in window.
    - When the input occurs, the camera location is changed.

Virtual Trackball

- **A 2D point (x, y) on the window corresponds to:**
  - the 3D point (x, y, z) on the upper hemisphere where

$$r^2 = x^2 + y^2 + z^2$$
$$z = \sqrt{r^2 - x^2 - y^2}, \quad \text{if} \quad r \geq |x|, |y| \geq 0$$



Virtual Trackball Source Code

- **Class for virtual trackball**
  - Here's an interface for trackball class.

```
struct trackball
{
   bool    b_tracking = false;
   float   scale;         // controls how much rotation is applied
   mat4    view_matrix0; // initial view matrix
   vec2    m0;            // the last mouse position

    trackball( float rot_scale=1.0f ):scale(rot_scale){}
    bool is_tracking() const { return b_tracking; }

    void begin( const mat4& view_matrix, vec2 m );
    void end();
    void update( vec2 m );
}
```

Callback: `mouse()`

- **mouse():**
  - When the button is pressed, call begin(). Otherwise call end().
  - Retrieve a mouse position, and pass to tb.begin() with view matrix.

```
void mouse( GLFWwindow* window, int button, int action, int mods )
{
   if(button==GLFW_MOUSE_BUTTON_LEFT)
   {
      dvec2 pos; glfwGetCursorPos(window,&pos.x,&pos.y);
      vec2 npos = cursor_to_ndc( pos, window_size );

      if(action==GLFW_PRESS) tb.begin( cam.view_matrix, npos );
      else if(action==GLFW_RELEASE) tb.end();
   }
}
```

cursor_to_ndc()

- **cursor_to_ndc():**
  - Converts a position in window coordinates to normalized coordinates.
  - Here, we first normalize to $[0,1]^2$

```
vec2 cursor_to_ndc( dvec2 cursor, ivec2 window_size )
{
    // normalize window pos to [0,1]^2
    vec2 npos = vec2( float(cursor.x)/float(window_size.x-1),
                      float(cursor.y)/float(window_size.y-1) );
    ...
```

cursor_to_ndc()

- **cursor_to_ndc():**
  - Vertical flipping is applied while normalizing to [-1,1]^2
    - Window/GLFW systems define y from top to bottom, while our virtual trackball (and OpenGL) defines y from bottom to top

```
vec2 cursor_to_ndc( dvec2 cursor, ivec2 window_size )
{
    ...

    // normalize window pos to [-1,1]^2 with vertical flipping
    // vertical flipping: window coordinate system defines y from
    // top to bottom, while the trackball from bottom to top
    return vec2(npos.x*2.0f-1.0f,1.0f-npos.y*2.0f);
}
```

Virtual Trackball Source Code

- **Methods for trackball class**
  - At `begin()`, we mark we are tracking the mouse movements.
  - Also, we record the initial mouse position and view matrix.

```
void begin( const mat4& view_matrix, vec2 m )
{
    b_tracking = true;           // enable trackball tracking
    m0 = m;                      // save current mouse position
    view_matrix0 = view_matrix;  // save current view matrix
}
```

  - At `end()`, we just disable tracking, and do not need to touch others.

```
void end(){ b_tracking = false; }
```

Callback: `motion()`

- **motion():**
  - if not tracking, return
  - otherwise, calls update() when tracking

```
void motion( GLFWwindow* window, double x, double y )
{
    if(!tb.is_tracking()) return;
    vec2 npos = cursor_to_ndc( dvec2(x,y), window_size );
    cam.view_matrix = tb.update( npos );
}
```

Virtual Trackball Source Code

- **update():**

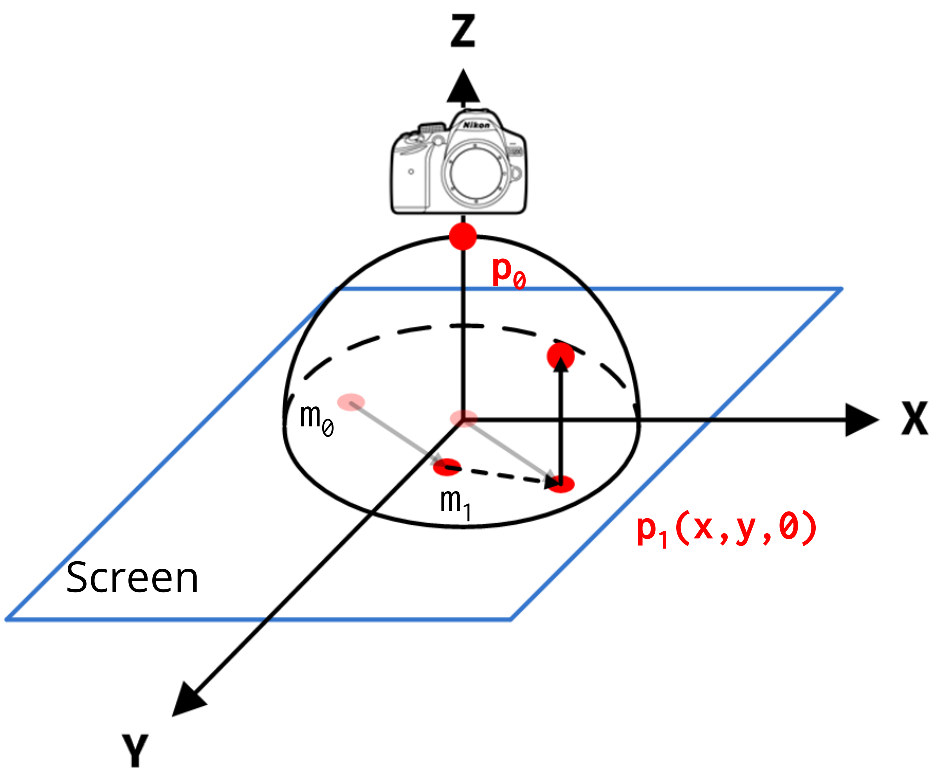
```
mat4 update( vec2 m )
{
   // project a 2D mouse position to a unit sphere
   vec3 p0 = vec3(0,0,1.0f);  // reference position on sphere
   vec3 p1 = vec3(m-m0,0);    // displacement
```

  - We then define the reference point `p0` on the virtual sphere.
  - Then, define `p1` as a displacement of mouse position.

Virtual Trackball Source Code

- **update():**
  - Visualization of p0 and p1 on the unit sphere

```
vec3 p0 = vec3(0,0,1.0f);   // reference position on sphere
vec3 p1 = vec3(m-m0,0);     // displacement
```



Virtual Trackball Source Code

- **update():**
  - Then, we detect a subtle/trivial movement, and ignore it.

```
// ignore subtle movement
if( !b_tracking || length(p1)<0.001f ) return view_matrix0;
...
```

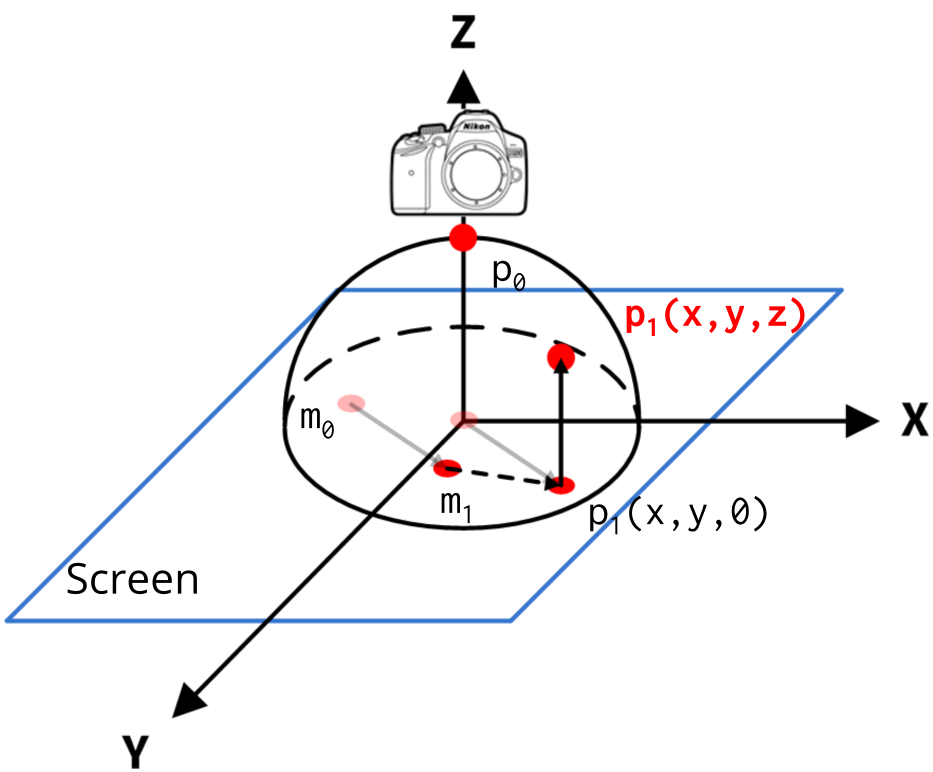
  - Then, apply rotational scale

```
// apply rotation scale
p1 *= scale;
```

Virtual Trackball Source Code

- **update():**
  - Then, back-project z=0 to the unit sphere.

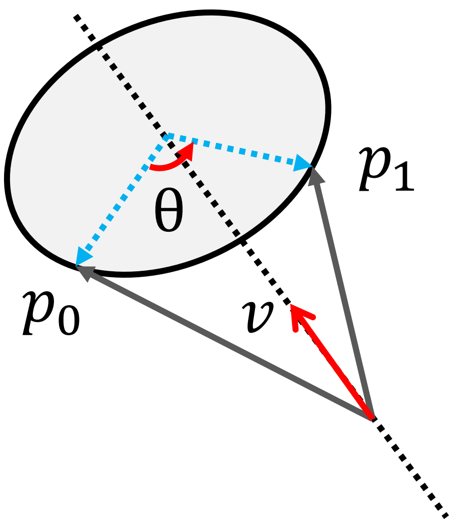
```
// back-project z=0 onto the unit sphere: z^2 = 1 - (x^2 + y^2)
p1.z = sqrtf(max(0,1.0f-length2(p1)));
p1 = p1.normalize();
```



Virtual Trackball Source Code

- **update():**
  - Find the rotation axis and angle

```
// find rotation axis and angle in world space
// - trackball self-rotation should be done at first in the world space
// - mat3(view_matrix0): rotation-only view matrix
// - mat3(view_matrix0).transpose(): inverse view-to-world matrix

vec3 v = mat3(view_matrix0).transpose()*p0.cross(p1);
float theta = asin( min(v.length(),1.0f) );
```



Virtual Trackball Source Code

- **update():**
  - Return the rotation with the initial view transformation

```
mat4 update( float x, float y )
{
    ...

    // resulting view matrix, which first applies
    // trackball rotation in the world space
    return view_matrix0 * mat4::rotate(v.normalize(),theta);
}
```

Exercises:
Virtual Trackball Extension

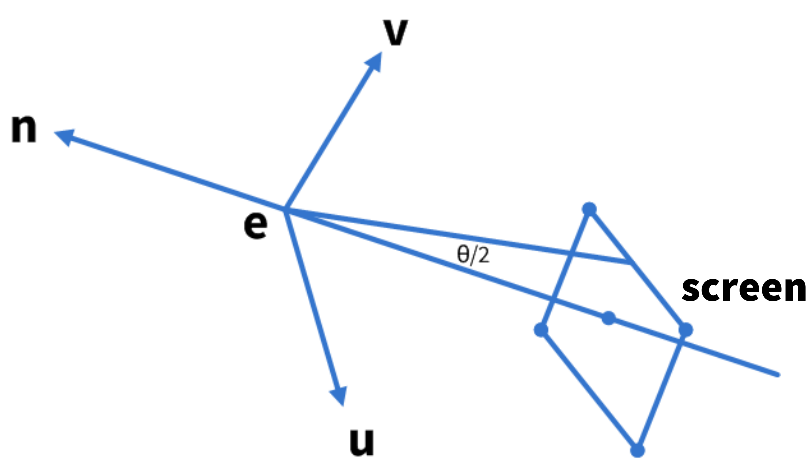
Extending Virtual Trackball: Hints!

- **Changes in your mouse and motion():**

```
void mouse( GLFWwindow* window, int button, int action, int mods ){
{
    ...
    tb.button = button;
    tb.mods = mods;
}

void motion( GLFWwindow* window, double x, double y )
{
    ...
    if(button==GLFW_MOUSE_BUTTON_LEFT&&mods==0)
        tb.update(npos);
    else if(button==GLFW_MOUSE_BUTTON_MIDDLE||
            (button==GLFW_MOUSE_BUTTON_LEFT&&(mods&GLFW_MOD_CONTROL)))
        tb.update_pan(npos);
    else if(button==GLFW_MOUSE_BUTTON_RIGHT||
            (button==GLFW_MOUSE_BUTTON_LEFT&&(mods&GLFW_MOD_SHIFT)))
        tb.update_zoom(npos);
}
```

Extending Virtual Trackball: Hints!

- **How to implement panning**
  - The mouse displacement is mapped to translation along *uv* plane.
  - Then, move eye and at using the displacement, and rebuild your view matrix using look_at().
  - It is good to scale the amount of eye's panning based on distance to the scene center (or at).
- **How to implement zooming**
  - The mouse displacement is mapped to translation along *n* axis.
  - Displace only eye (or both eye and at), and rebuild the view matrix.



Extending Virtual Trackball: Hints!

- **Potential changes in tracking**
  - In the basic example for the virtual trackball, we directly multiply the new rotation from the trackball. When you need to apply panning and zooming as well, there could be many ways for that.
  - One of the ways is to apply the new transformation to the eye, at, and up in the last step of update(), instead of directly applying to the view matrix. Then, build the new view matrix using those three transformed parameters.
  - More specifically, eye could also need the translation as well as rotation. The center may be kept the same. The up only needs rotation (since it is a direction vector).
  - Then, you can apply zooming and panning using the new transformed eye, at, up to the eye, at, up. Then, rebuild the view matrix using them.