ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

MidMaps with OpenGL - Extension

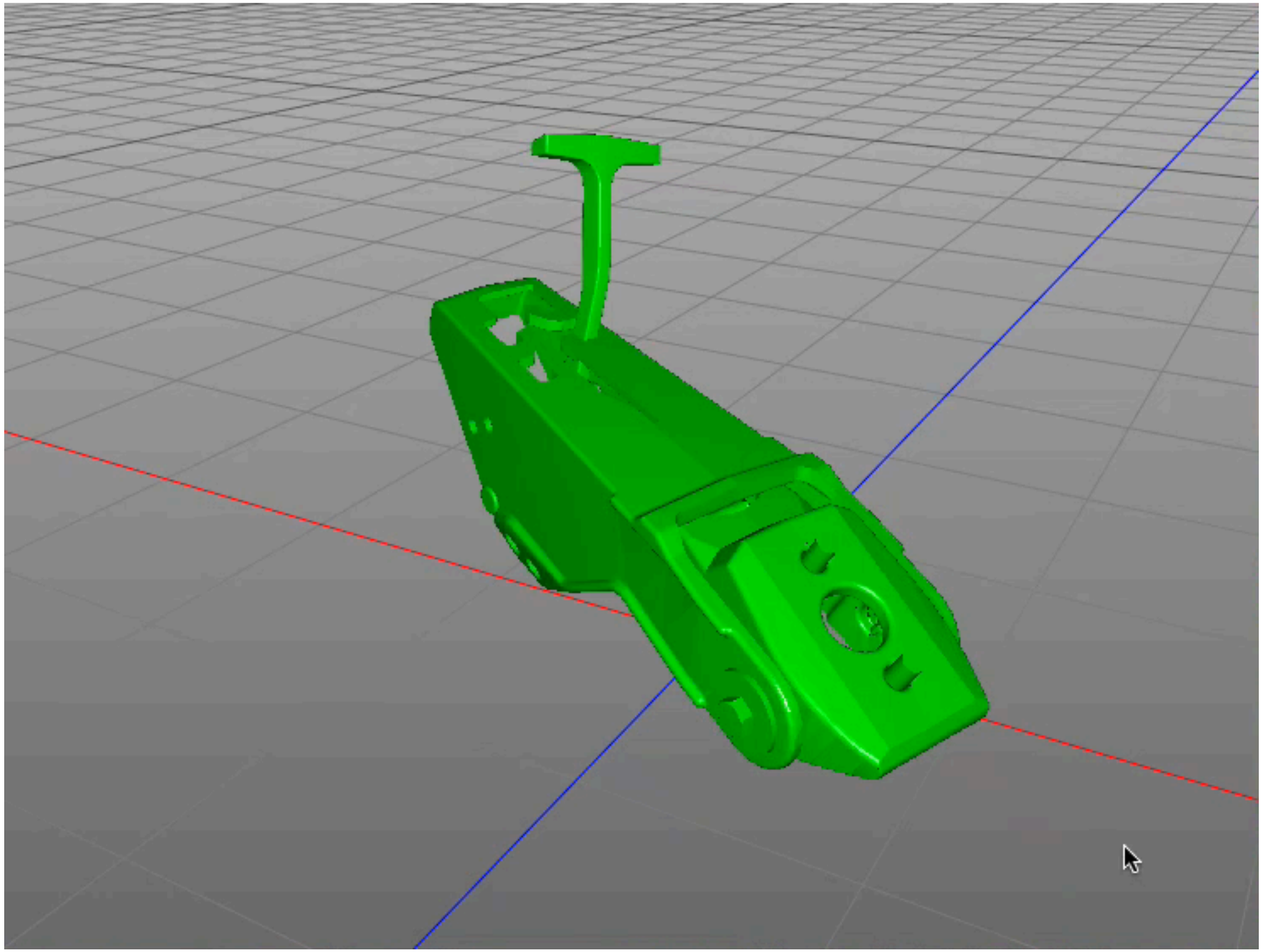
October 20th, 2015

Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Trackball Interaction

ARLAB



First Person Shooter Navigation

ARLAB

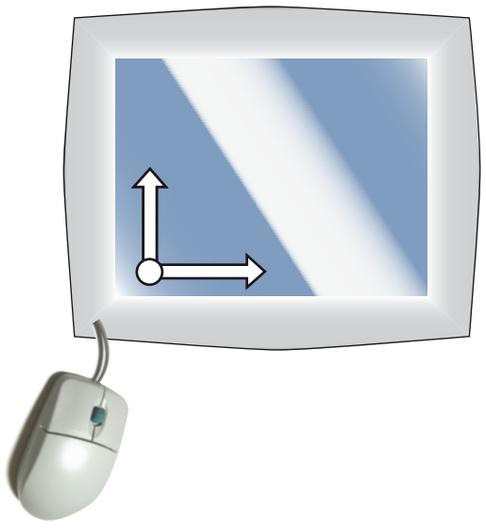
Content

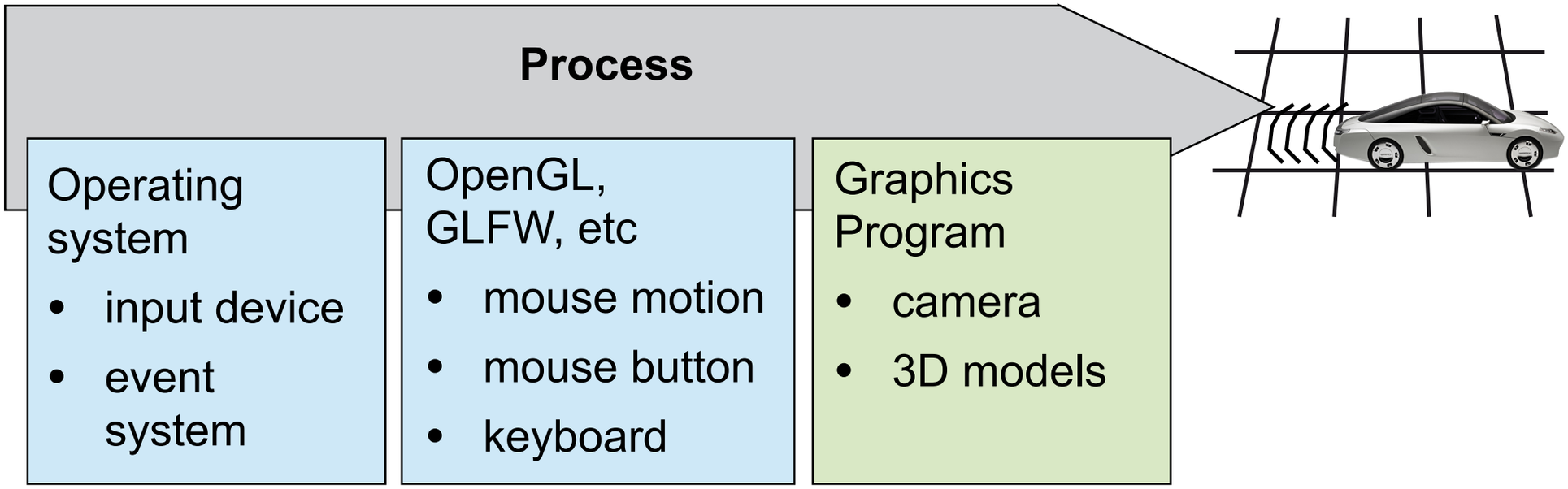
ARLAB

- Review - projection and virtual camera
- Interaction event handling
- Mouse-based object manipulation
  - Trackball Interface
  - Example Program
- Navigation in 3D space
  - Ray-Intersection Test
  - First-person shooter controls
  - Example Program

We talk about the technical part of interaction and not the HCI part

OpenGL & Interaction

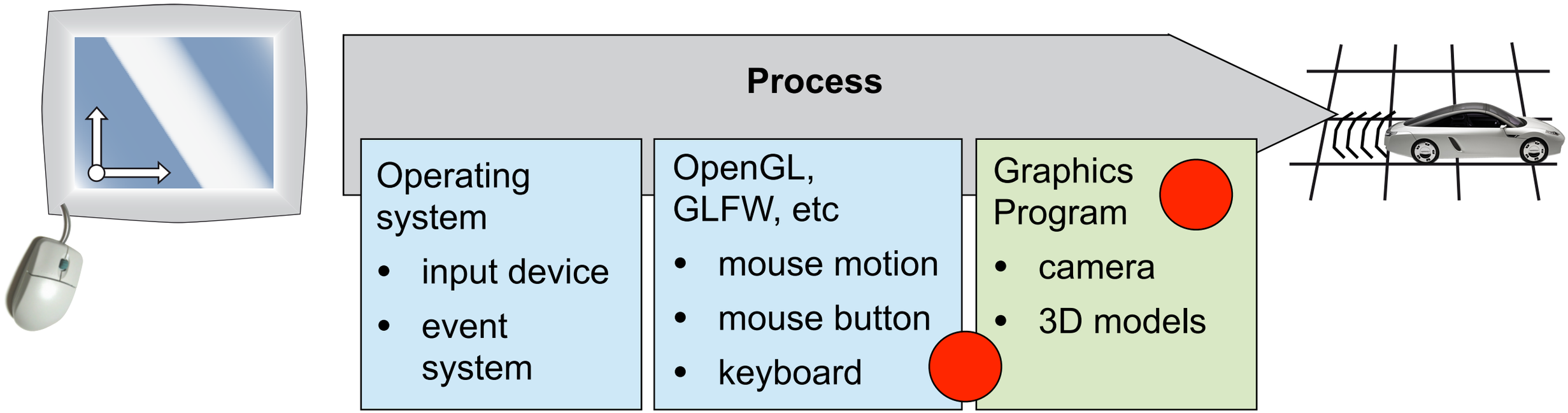




Different components of the hardware / software have different responsibilities:

- Operating system: manages the hardware
- Graphic frameworks: provide the basic message chain to deliver data from devices.
- Graphics program: needs to transfer the motion and interaction data into camera positions etc.

OpenGL & Interaction



We work on two ends:

First we need to program an event handling system as interface between our application and the underlying framework

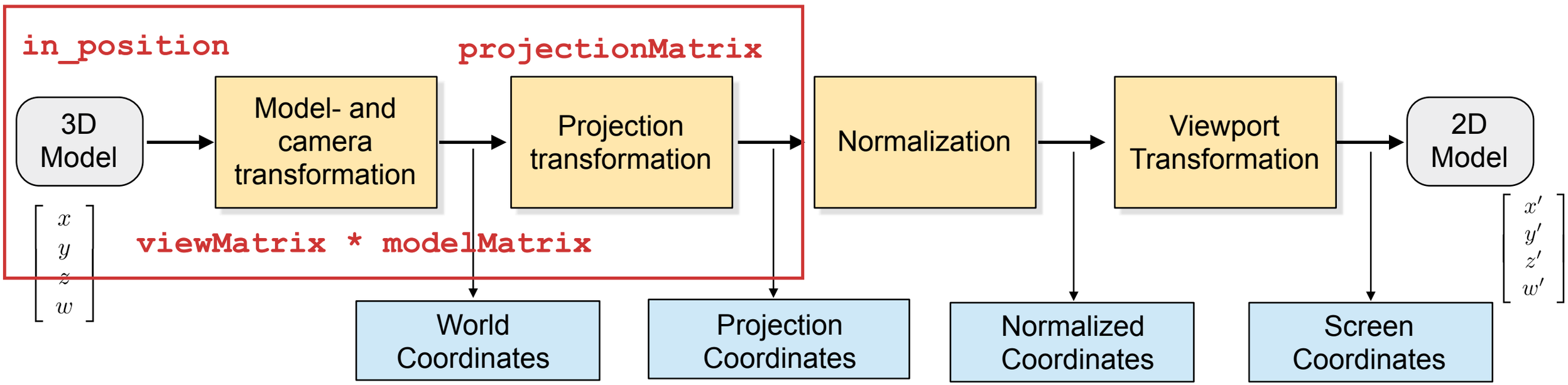
Second, we have to program "the Interactor", the mechanism that transfers mouse movements into a camera position / orientation.

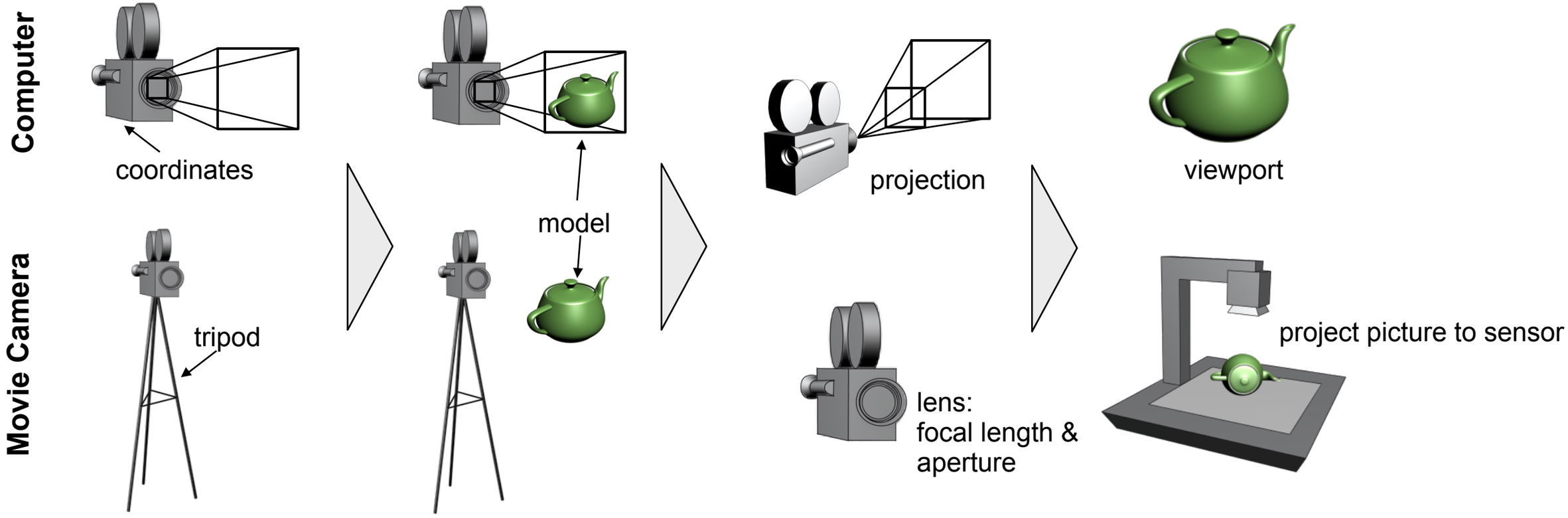
ARLAB

Review - projection and virtual camera

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

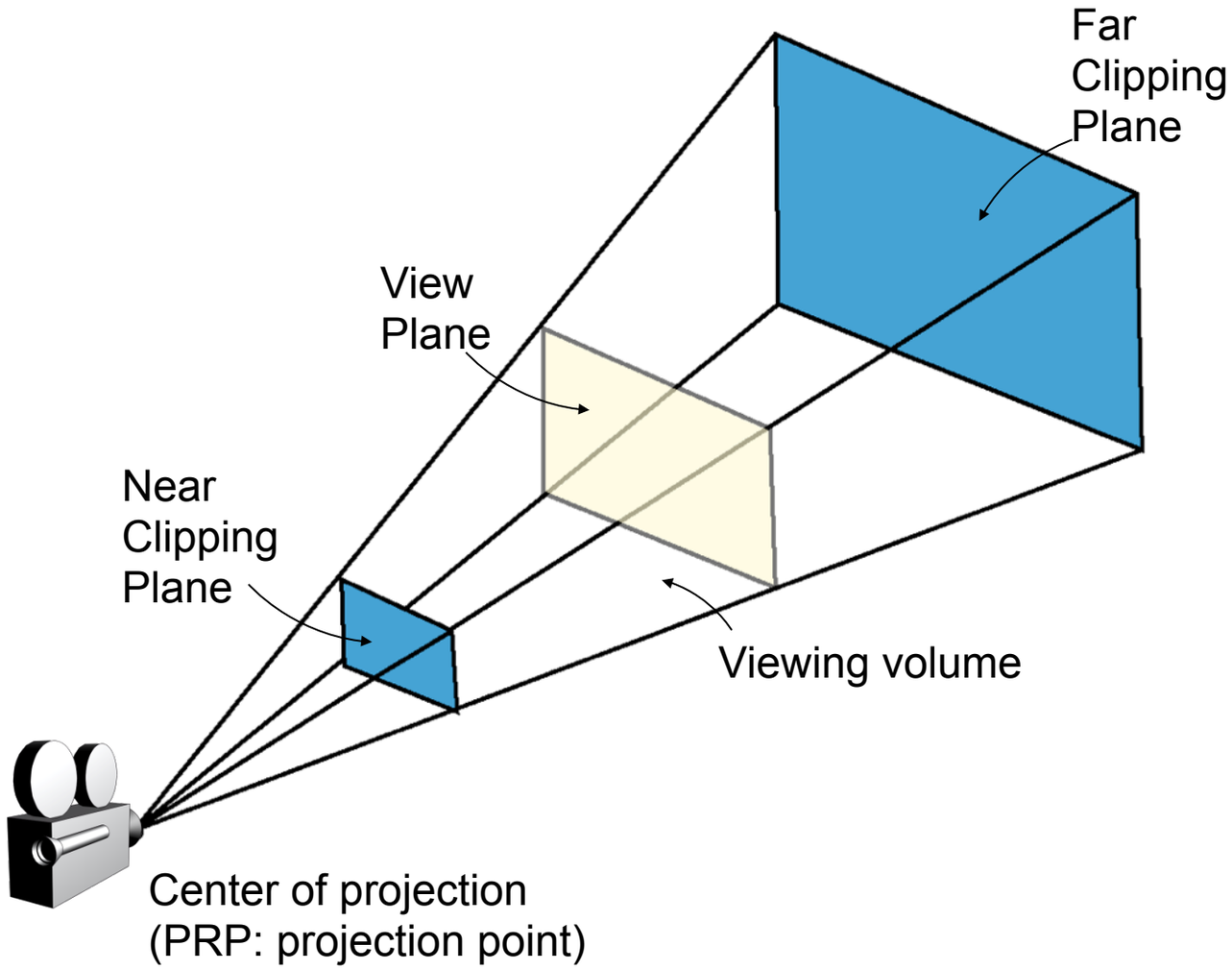
Viewing in 3D





Projection Matrix

- The view frustum



The viewing volume of a perspective projection.

Matrix for perspective projection:

$$P = \begin{bmatrix} \frac{1}{f \tan \theta_h/2} & & & \\ & \frac{1}{f \tan \theta_v/2} & & \\ & & \frac{1}{f} & \\ & & & 1 \end{bmatrix}$$

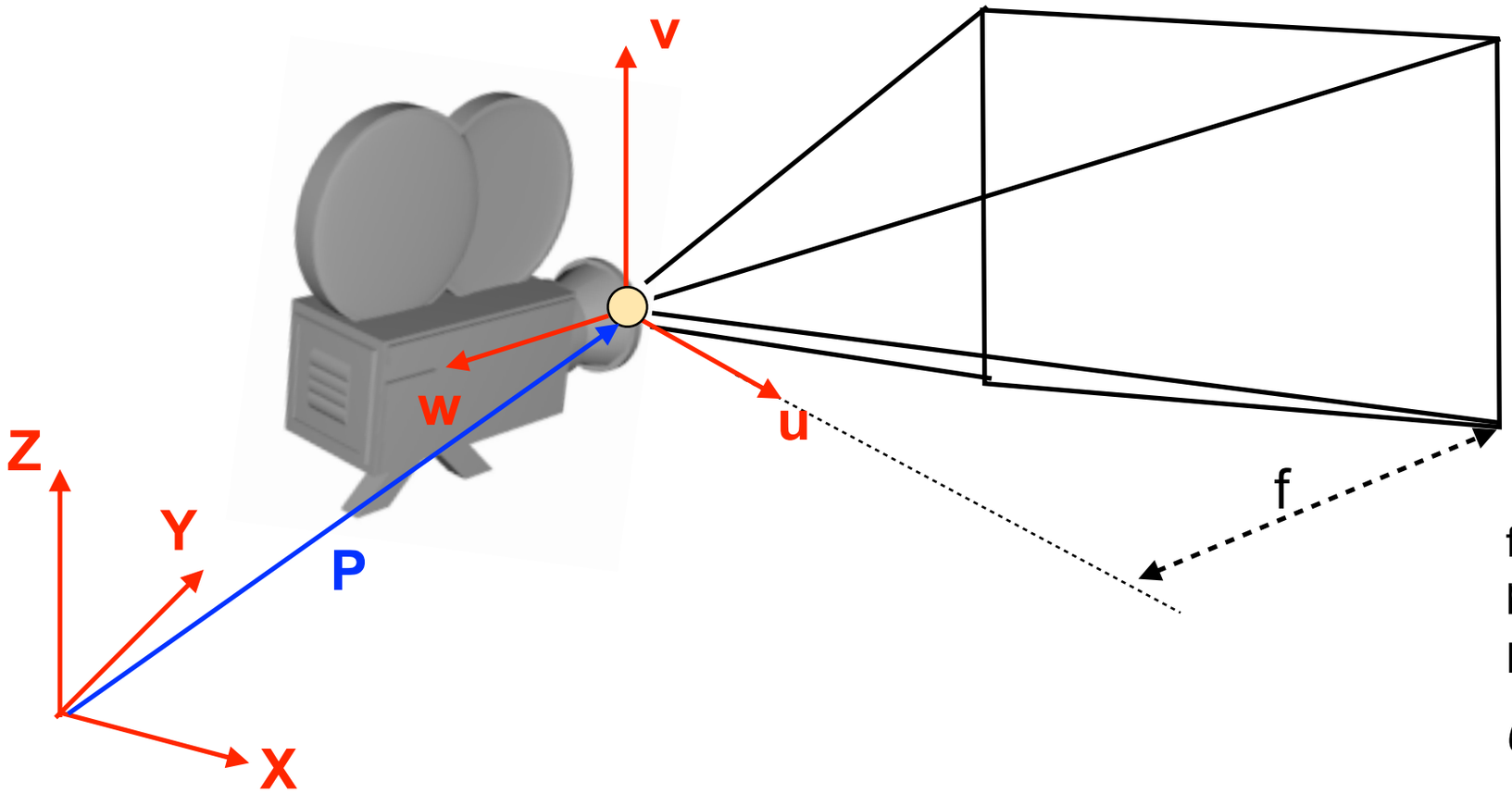
f: focal length / distance to far plane / projection plane

P: camera transformation

θ_h : horizontal field of view

θ_v : vertical field of view

Perspective Projection



f: focal length / distance to far plane / projection plane

P: camera transformation

θ_h : horizontal field of view

θ_v : vertical field of view

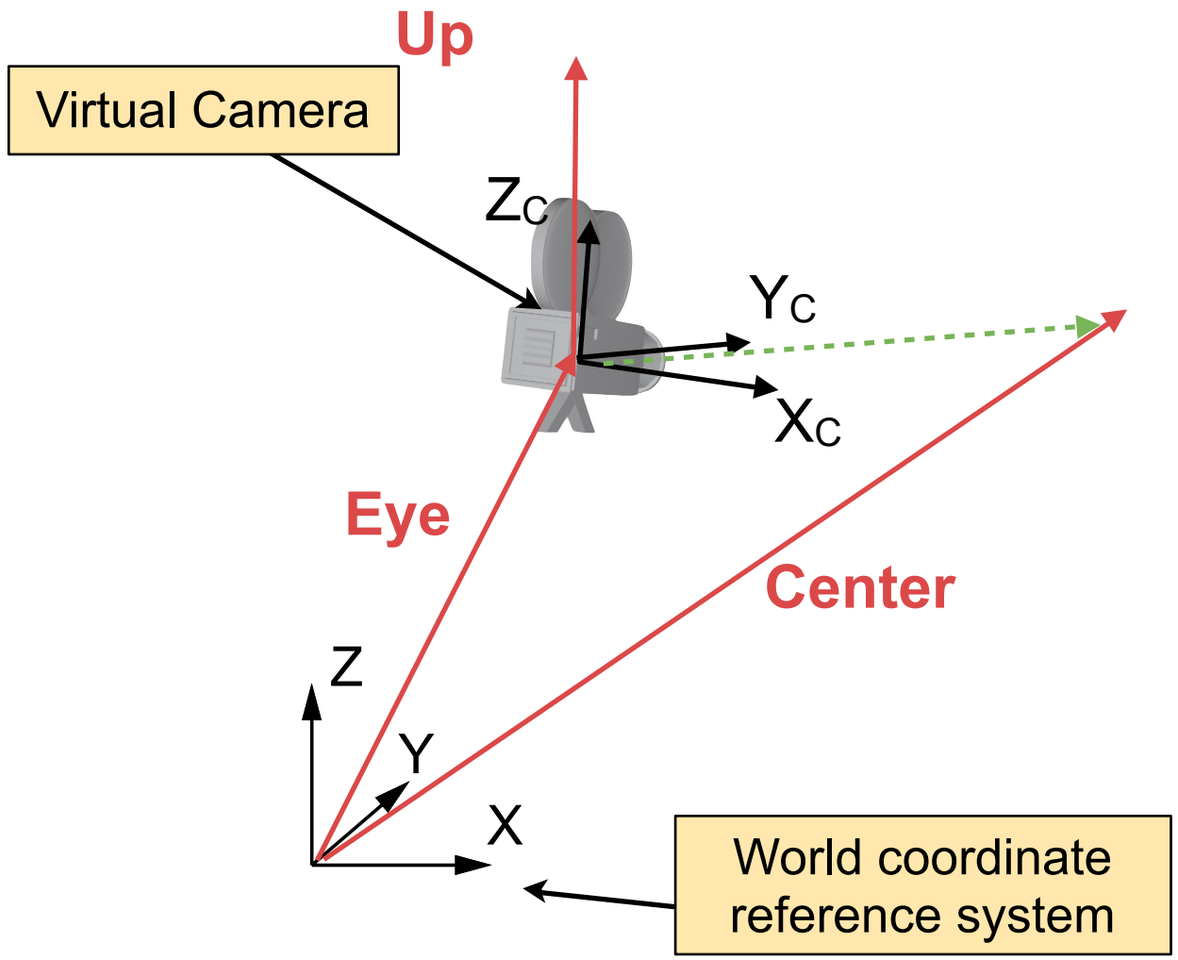
projection matrix

view matrix

$$M = \begin{bmatrix} \frac{1}{f \ \tan\theta_h/2} & & & \\ & \frac{1}{f \ \tan\theta_v/2} & & \\ & & \frac{1}{f} & \\ & & & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View Matrix

- Camera Location and Orientation



Vectors to setup a virtual camera

```
glm::mat4  lookAt (
detail::tvec3< T > const &eye,
detail::tvec3< T > const &center,
detail::tvec3< T > const &up)
```

Parameters to describe a camera position and orientation:

- eye: Specifies the position of the eye.
- center
  Specifies the position of the point to look to.
- up: Specifies the direction of the up vector.

Example Shader Code

The matrices are defined as uniform variables

```
static const string vs_string =
    "#version 410 core                                                  \n"
    "                                                                    \n"
    "uniform mat4 projectionMatrix;                                     \n"
    "uniform mat4 viewMatrix;                                            \n"
    "uniform mat4 modelMatrix;                                           \n"
    "in vec3 in_Position;                                                \n"
    "                                                                    \n"
    "in vec3 in_Color;                                                   \n"
    "out vec3 pass_Color;                                                \n"
    "                                                                    \n"
    "void main(void)                                                     \n"
    "{                                                                   \n"
    "    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0);  \n"
    "    pass_Color = in_Color;                                           \n"
    "}                                                                   \n";
```

- uniform: qualifier to declare the variable as a shared variable, shared between host computer and graphics card.
- mat4: a GLSL 4x4 matrix

$$\mathbf{M} = \begin{bmatrix} r_{01} & r_{02} & r_{03} & t_x \\ r_{11} & r_{12} & r_{13} & t_y \\ r_{21} & r_{22} & r_{23} & t_z \\ r_{31} & r_{32} & r_{33} & 1 \end{bmatrix}$$

ARLAB

Interaction event handling

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Callback Concept

A callback is a function that the programmer provides to a framework via an API. The framework calls (call - back) this functions to react to a specific event (mouse button pressed, email message, keyboard pressed, etc.)

Graphics System

(GLFW)

```
setCallback(&myFunction)
```

Initialization time

```
void myFunction(variables)
{
  // Do something
}
```

You program this function and provide its address to the framework.

```
myFunction()
```

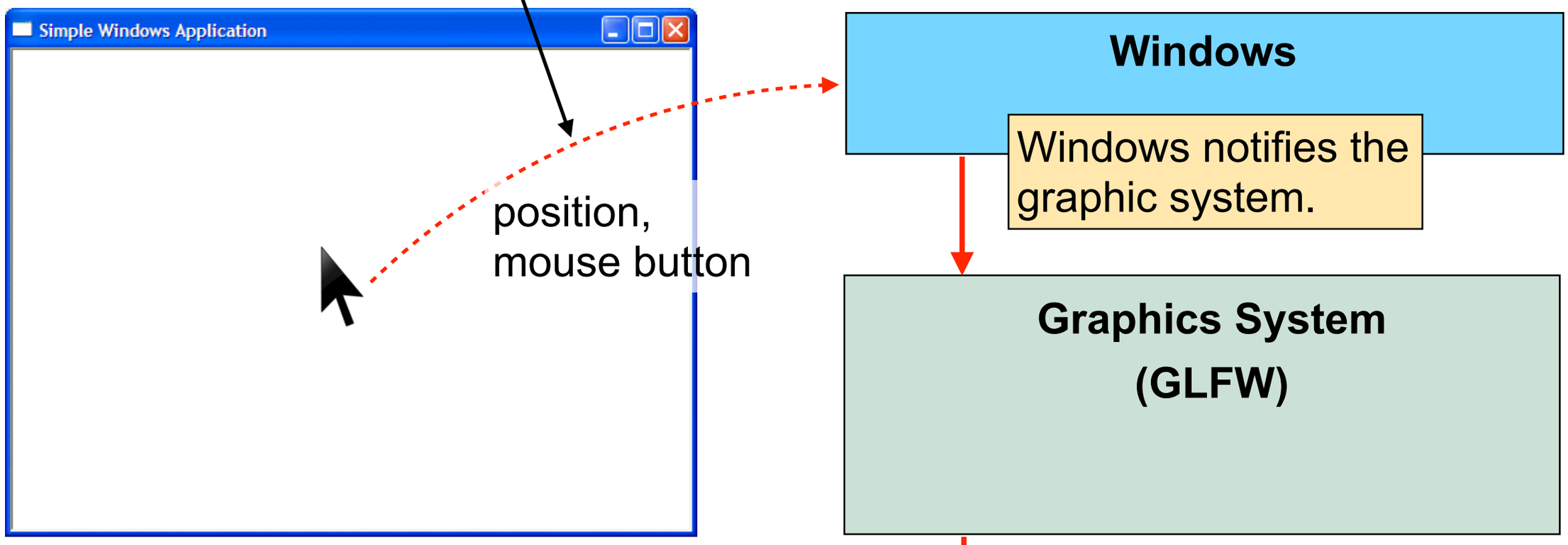
Runtime

```
void myFunction(variables)
{
  // Do something
}
```

The framework calls this function, i.e. every frame, when an event occurs, etc.

Concept

Operating system manages the resource "mouse" and notice the mouse interaction.



The graphics system invokes your function and executes the code.

```
void myFunction( int mouse button, int position)
{
    Interaction: do something, e.g. trackball interaction.
}
```

Interaction Functions

ARLAB

You have to implement and provide a set of functions that execute the interactions.

```
void myMouseMotionFunction( int mouse_id, int position)
{
    Interaction: do something, e.g. trackball interaction.
}
```

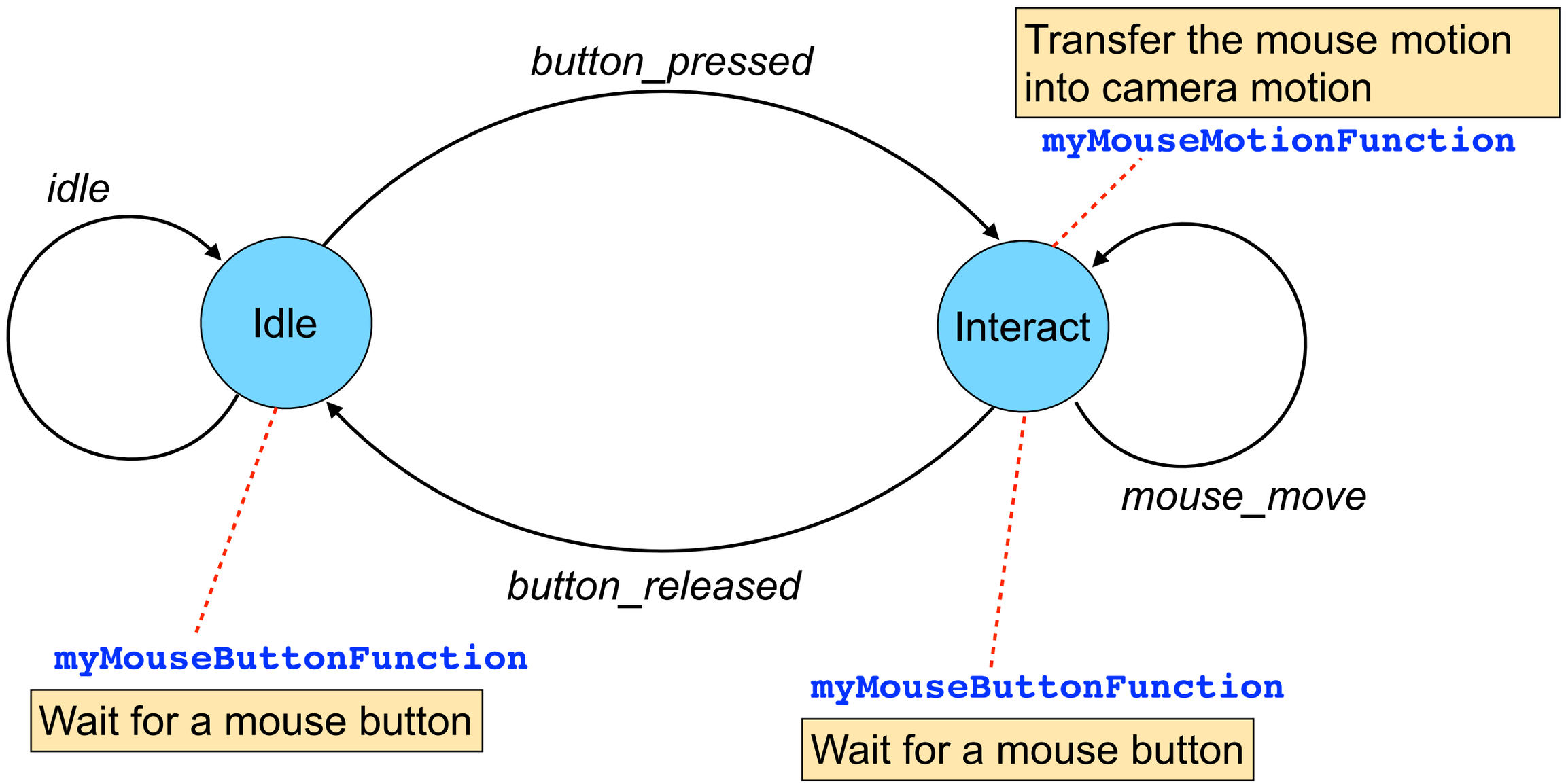
```
void myMouseButtonFunction( int mouse_button, int mouse_button)
{
    Interaction: do something, e.g. trackball interaction.
}
```

```
void myKeyboardFunction( int window, int key, int mode)
{
    Interaction: do something else....
}
```

- The function names are on you.
- The signature of the function is given and expected (API documentation).
- You register your functions before you start to render images.

State Machines

The typical processes for an interaction process:



State Transition

Glfw functions to set callbacks

ARLAB

Search for function on in the Input Handling section of GLFW

http://www.glfw.org/docs/latest/group__input.html

```
GLFWcursorposfun glfwSetCursorPosCallback( GLFWwindow * window,
                                           GLFWcursorposfun cbfun )
```

This function sets the cursor position callback of the specified window, which is called when the cursor is moved

```
GLFWmousebuttonfun glfwSetMouseButtonCallback( GLFWwindow * window,
                                               GLFWmousebuttonfun cbfun )
```

This function sets the mouse button callback of the specified window, which is called when a mouse button is pressed or released.

```
GLFWkeyfun glfwSetKeyCallback( GLFWwindow * window, GLFWkeyfun cbfun)
```

This function sets the key callback of the specified window, which is called when a key is pressed, repeated or released.

Glfw functions to set callbacks

ARLAB

You need know (look them up in the documentation) these functions and their signatures ; the parameters that go in.

```
GLFWcursorposfun glfwSetCursorPosCallback( GLFWwindow * window,
                                           GLFWcursorposfun cbfun )
```

This function sets the cursor position callback of the specified window, which is called when the cursor is moved

```
GLFWmousebuttonfun glfwSetMouseButtonCallback( GLFWwindow * window,
                                               GLFWmousebuttonfun cbfun )
```

This function sets the mouse button callback of the specified window, which is called when a mouse button is pressed or released.

```
GLFWkeyfun glfwSetKeyCallback( GLFWwindow * window, GLFWkeyfun cbfun)
```

This function sets the key callback of the specified window, which is called when a key is pressed, repeated or released.

GLFWmousebuttonfun

ARLAB

```
typedef void(* GLFWmousebuttonfun)(GLFWwindow *, int, int, int)
```

Name of the function:
(* abc): it expects a pointer, so the name of the function is up to you.

button action mods

Check the parameter documentation. The names are up to you.

Parameters:

[in] window: The window that received the event.

[in] button: The mouse button that was pressed or released.

[in] action: One of GLFW_PRESS or GLFW_RELEASE.

[in] mods: Bit field describing which modifier keys were held down.

```
/* In GLFW mouse callback */
void mouseButtonCallback( GLFWwindow * window, int button, int action, int mods )
{
    Do something
}
```

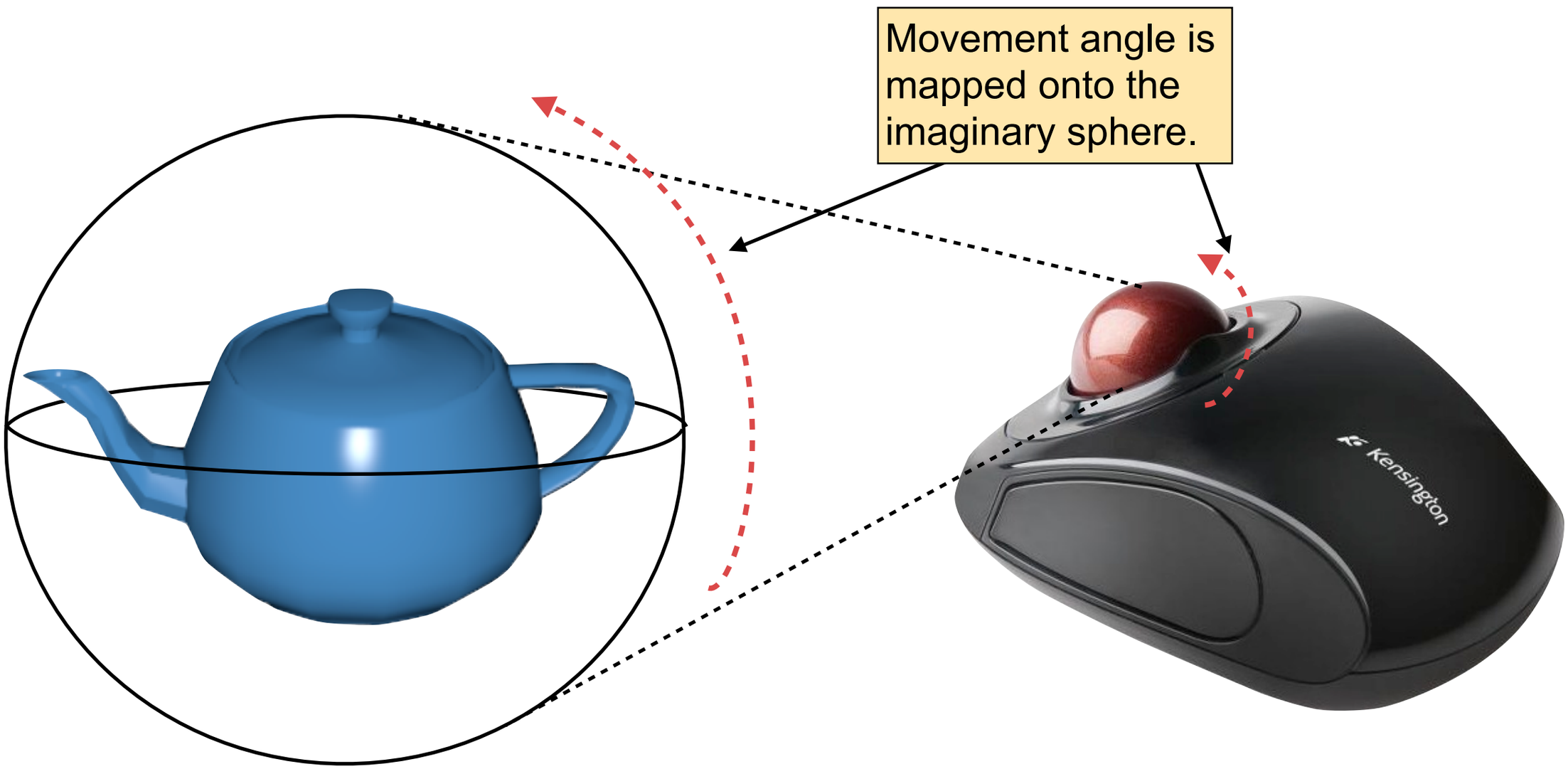
ARLAB

Mouse-based object manipulation

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Trackball

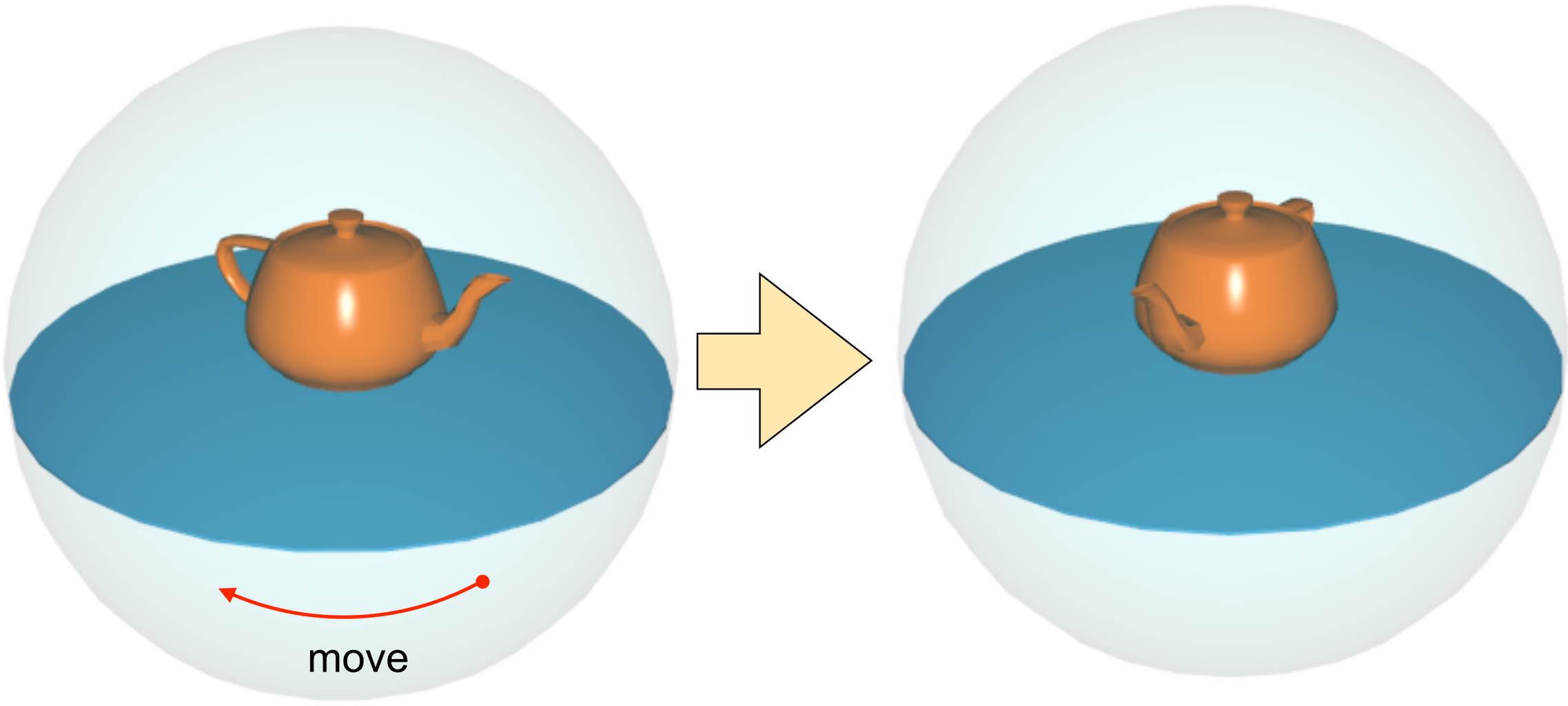
An imaginary sphere encase the object that needs to be rotated. The object moves along with the sphere of the trackball.



Trackball Manipulator

ARLAB

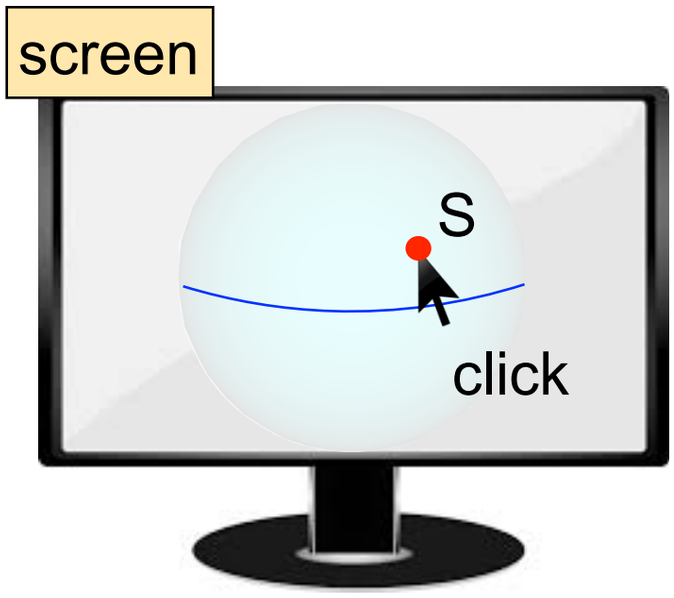
The trackball manipulator simulates the behavior of a trackball by using a mouse.

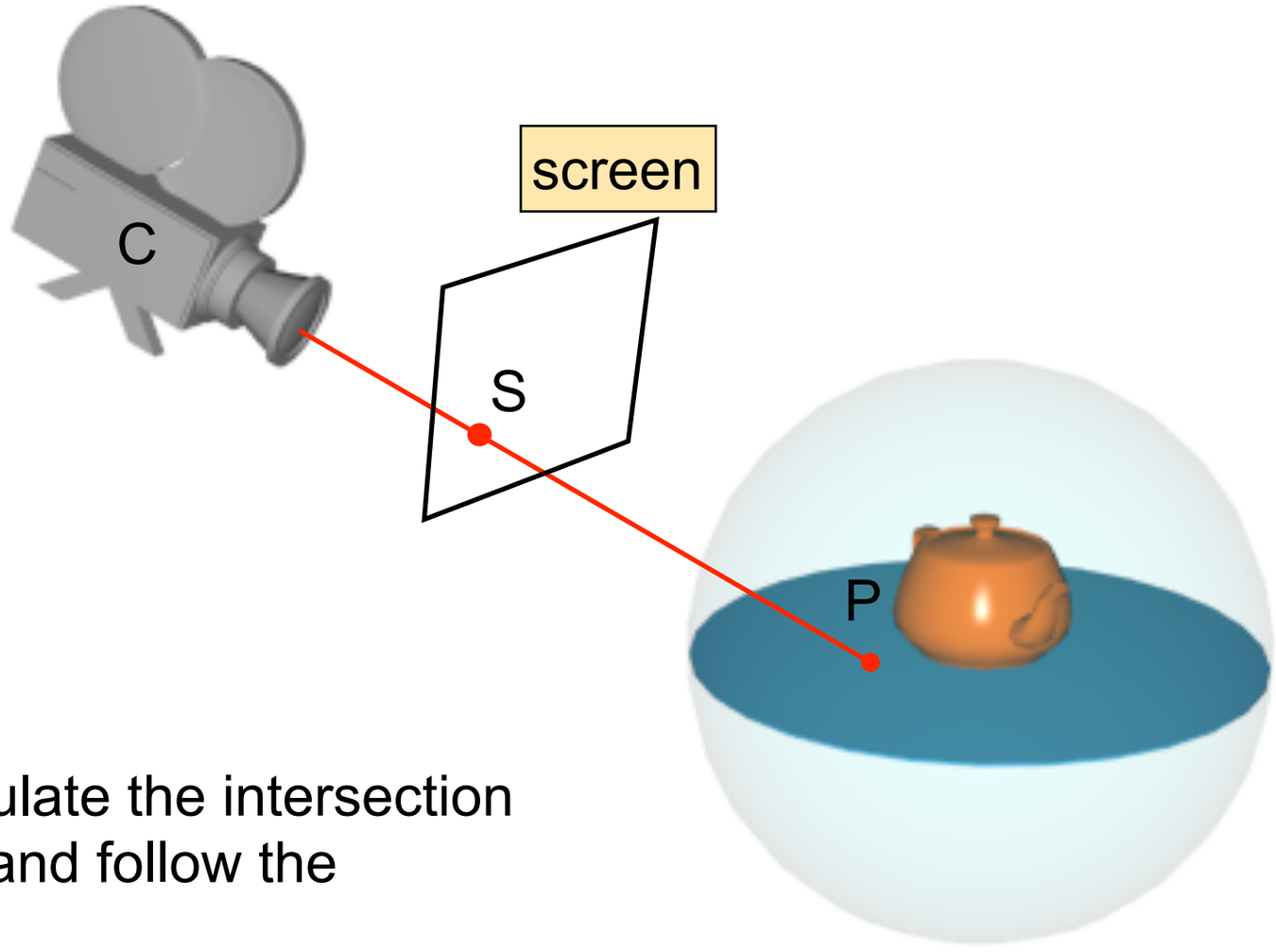


Imagine your object of interest lies in a glass sphere and you move a point on the surface.

Trackball Manipulator

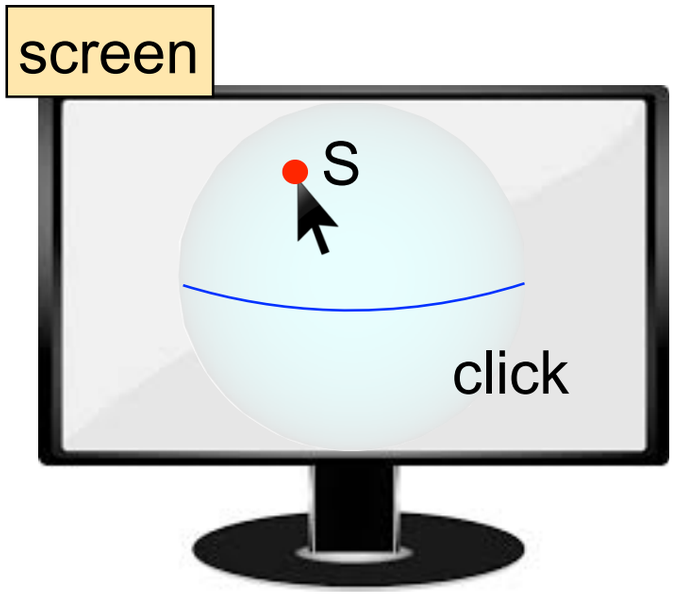
ARLAB

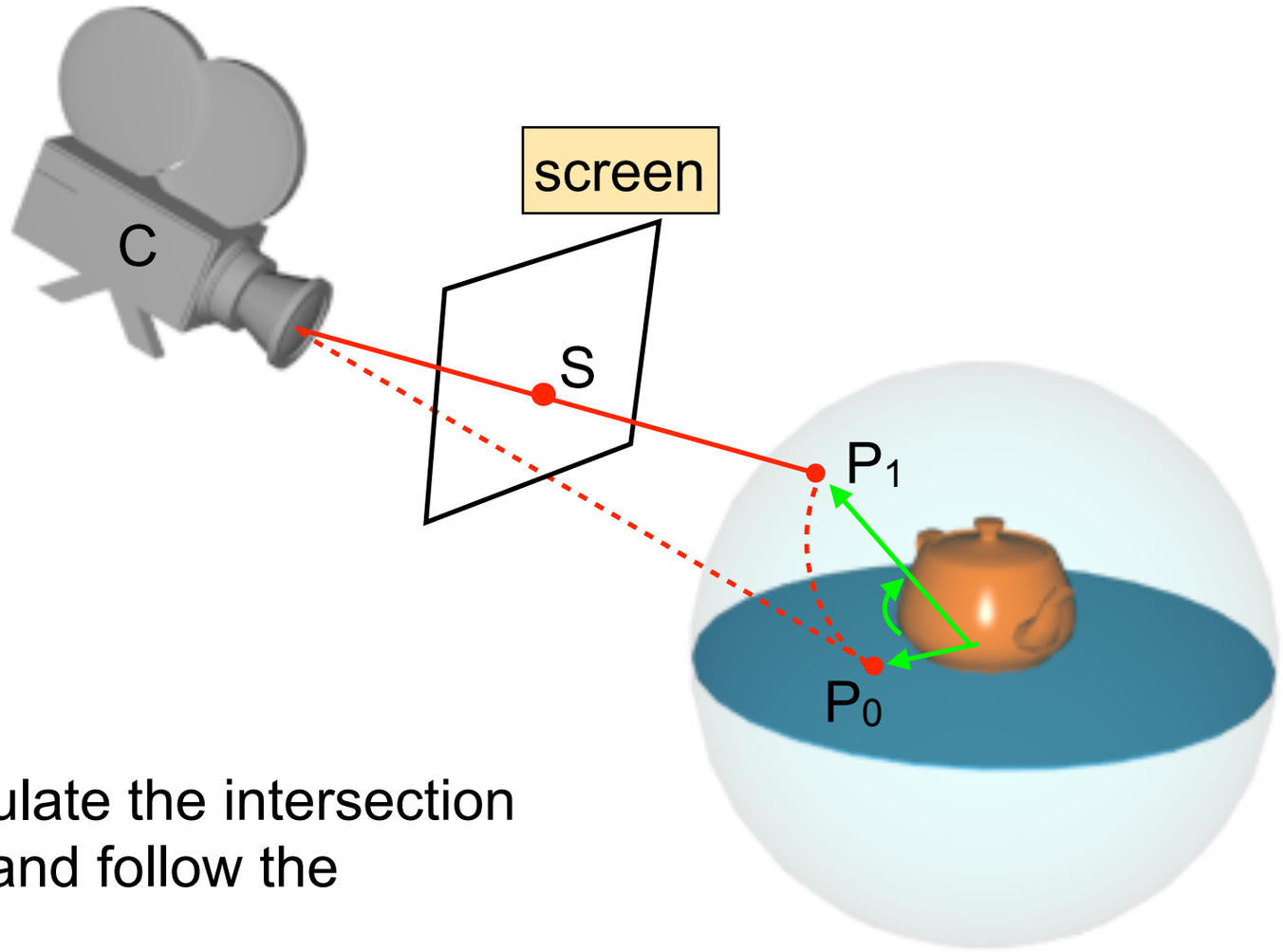




We wait for a mouse click, calculate the intersection of a ray (ray-intersection-test), and follow the movement of a point.

Trackball Manipulator





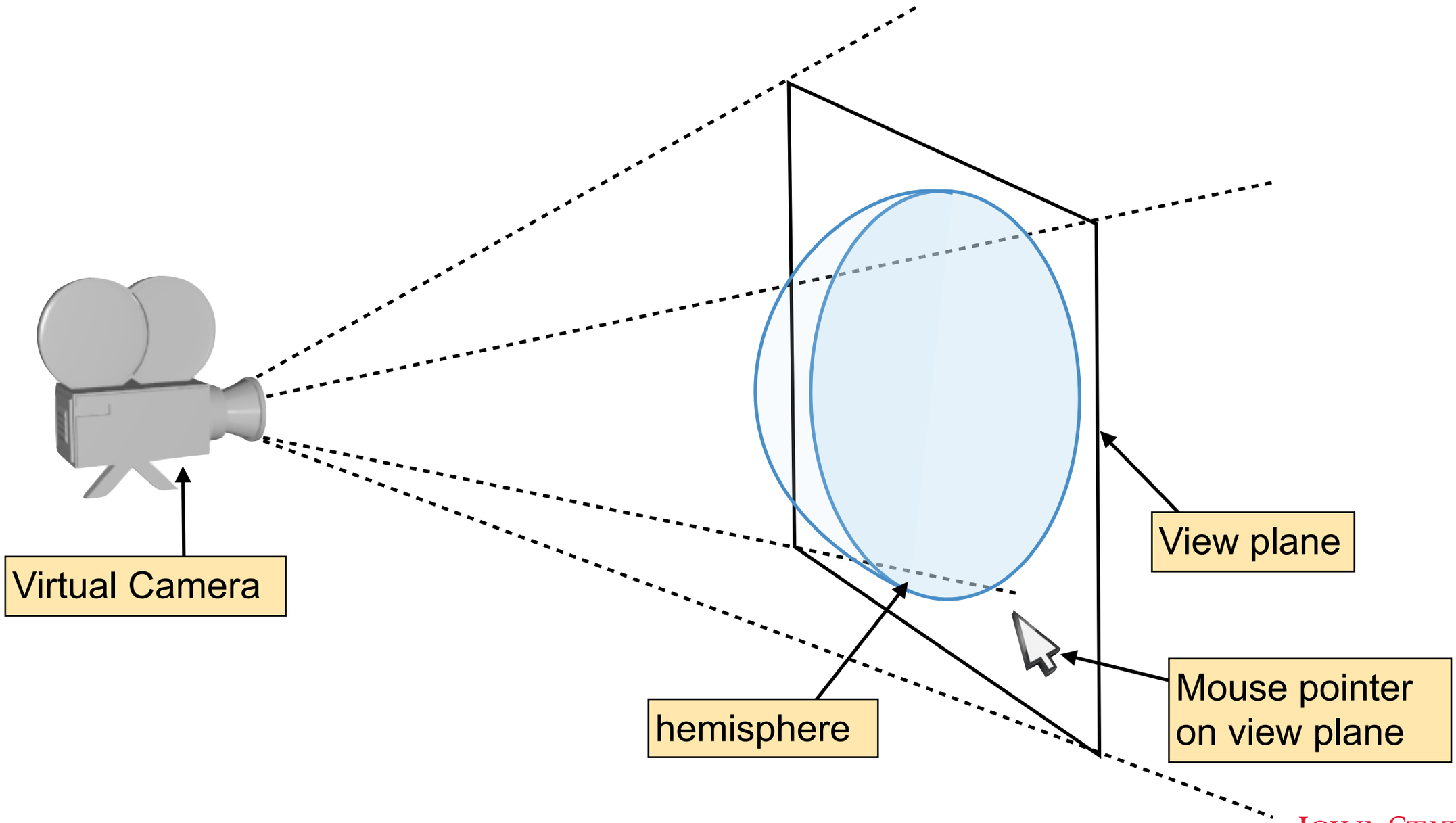
We wait for a mouse click, calculate the intersection of a ray (ray-intersection-test), and follow the movement of a point.

We obtain a second point while the user is moving the mouse, calculate the angle between both points, which we transfer to the model.

Trackball Manipulator

ARLAB

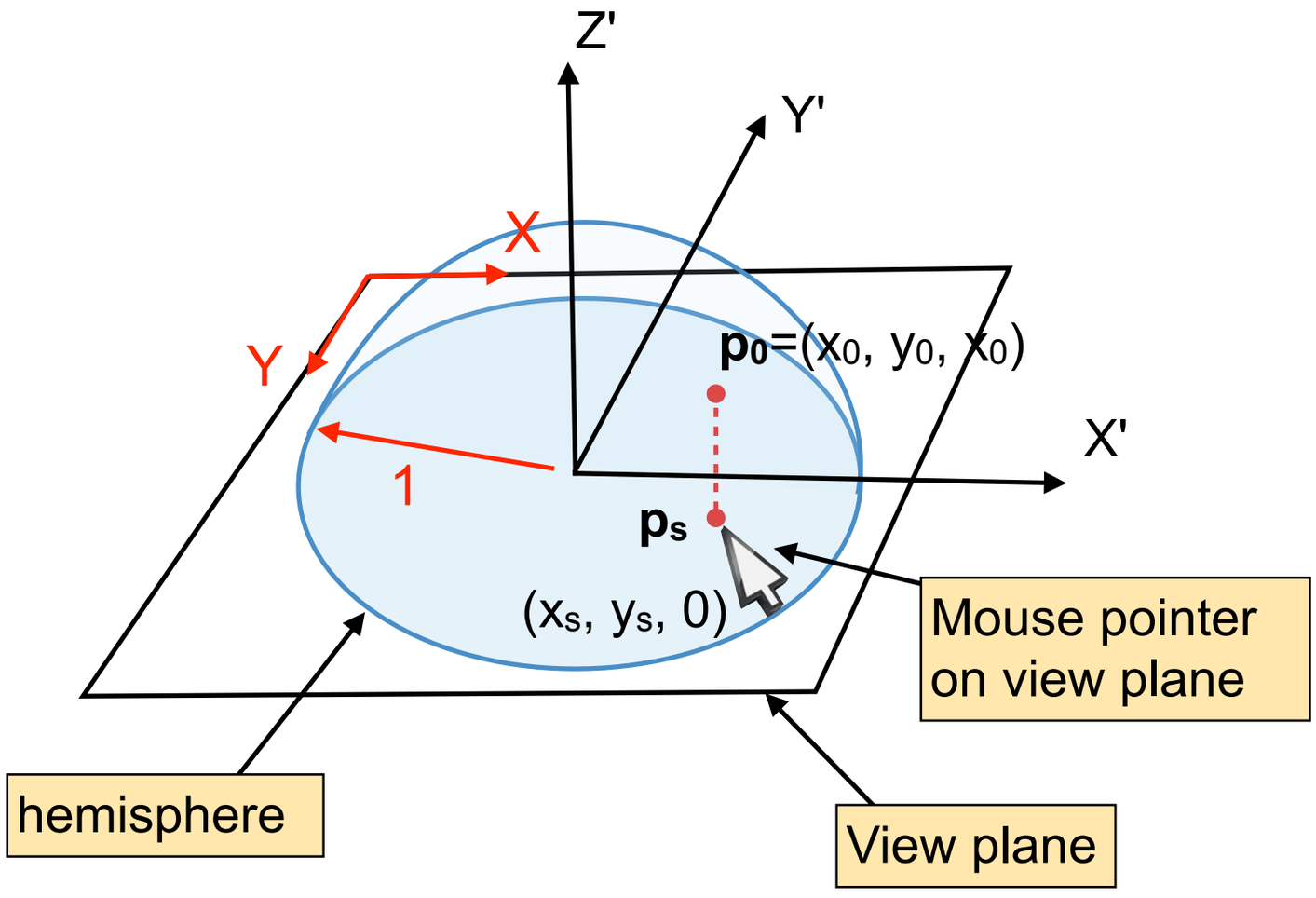
The virtual trackball simulates the navigation of a real trackball using a mouse pointer on screen. Superimpose a virtual hemisphere on top of the view plan.



Trackball Manipulator

ARLAB

1. Find the start point for the rotation



- The mouse pointer position in screen coordinates

  $\boldsymbol{p_s} = (x_s, y_s, 0)$

  is projected onto the surface of the virtual hemisphere

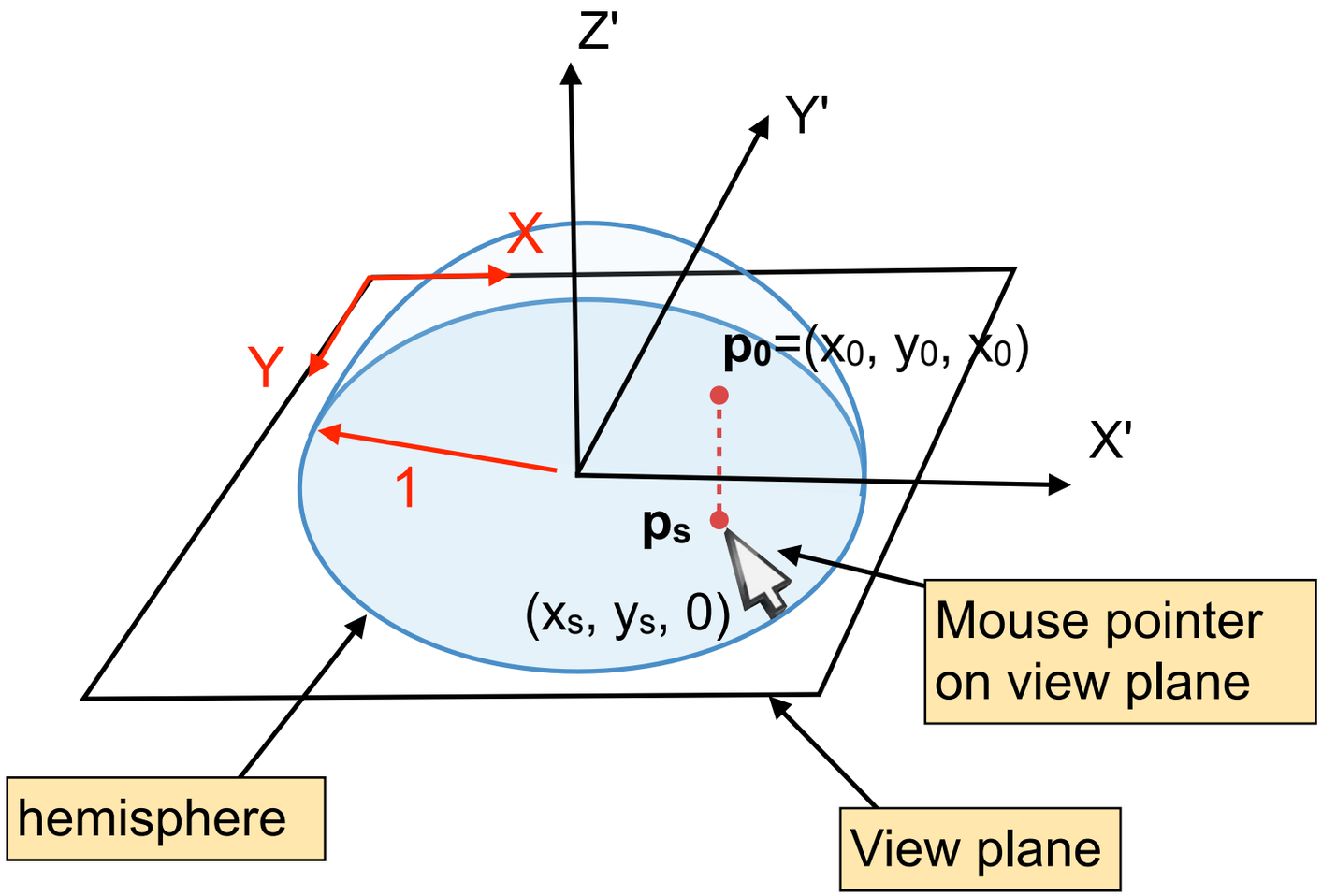
  $\boldsymbol{p_0} = (x_o, y_o, z_o)$

- The mouse pointer comes in screen coordinates.
- We assume a sphere with radius 1.
- Transfer the screen point ps to coordinate for the hemisphere (clamp it between -1 and 1)

Trackball Manipulator

ARLAB

1. Find the start point for the rotation



- Transfer the screen point ps to coordinate for the hemisphere (clamp it between -1 and 1)

$$x = \frac{2 \cdot x - window\_width}{window\_width}$$

$$y = \frac{2 \cdot y - window\_height}{window\_height}$$

$$z = \sqrt{z - length}$$

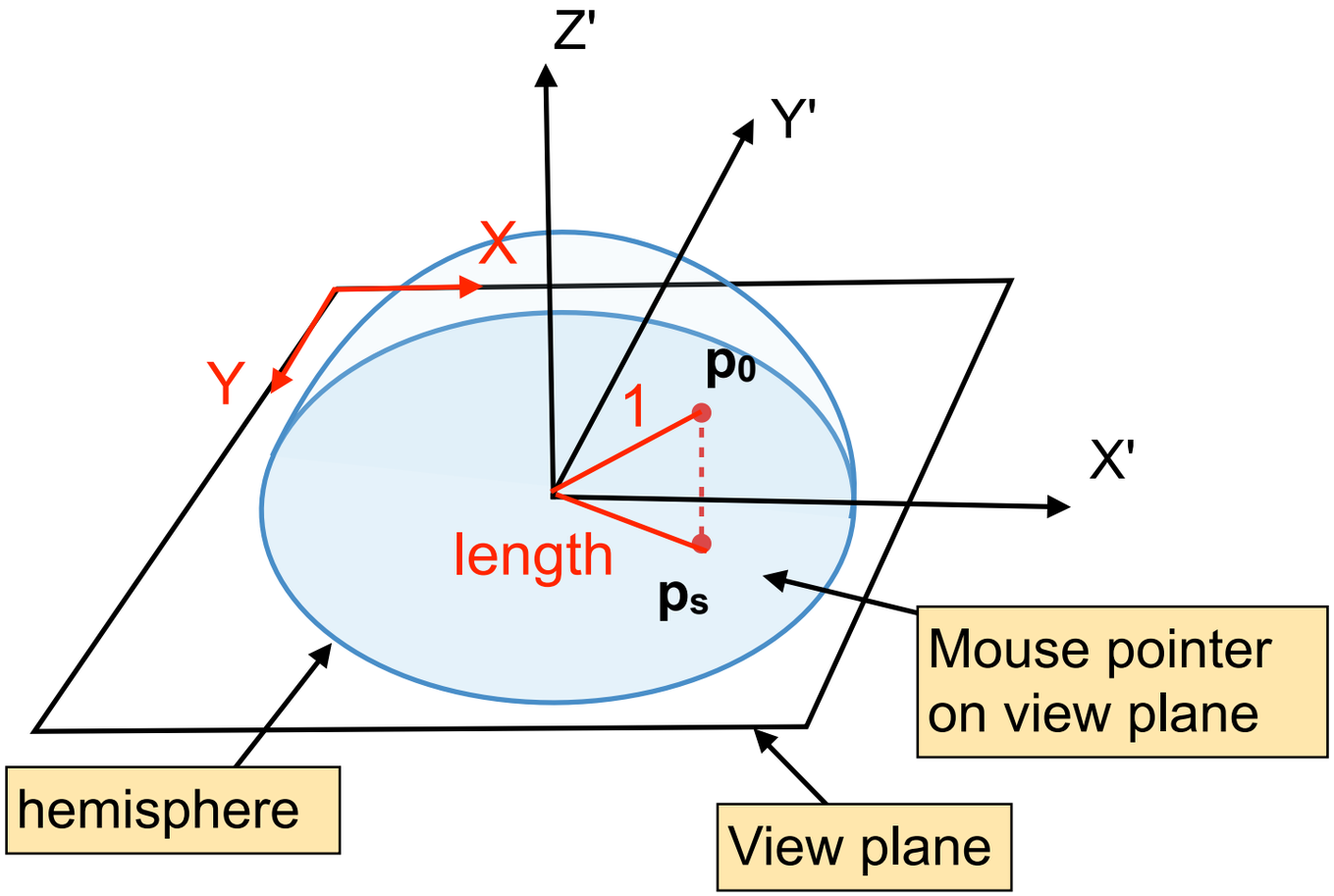
$$length = x^2 + y^2$$

Result in a value between -1 and 1

Trackball Manipulator

ARLAB

1. Find the start point for the rotation



- Transfer the screen point ps to coordinate for the hemisphere (clamp it between -1 and 1)

$$x = \frac{2 \cdot x - window\_width}{window\_width}$$

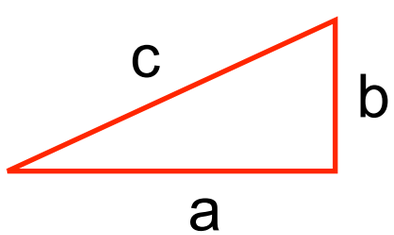
$$y = \frac{2 \cdot y - window\_height}{window\_height}$$

$$z = \sqrt{z - length}$$

$$length = x^2 + y^2$$

Result in a value between -1 and 1

Pythagorean theorem:

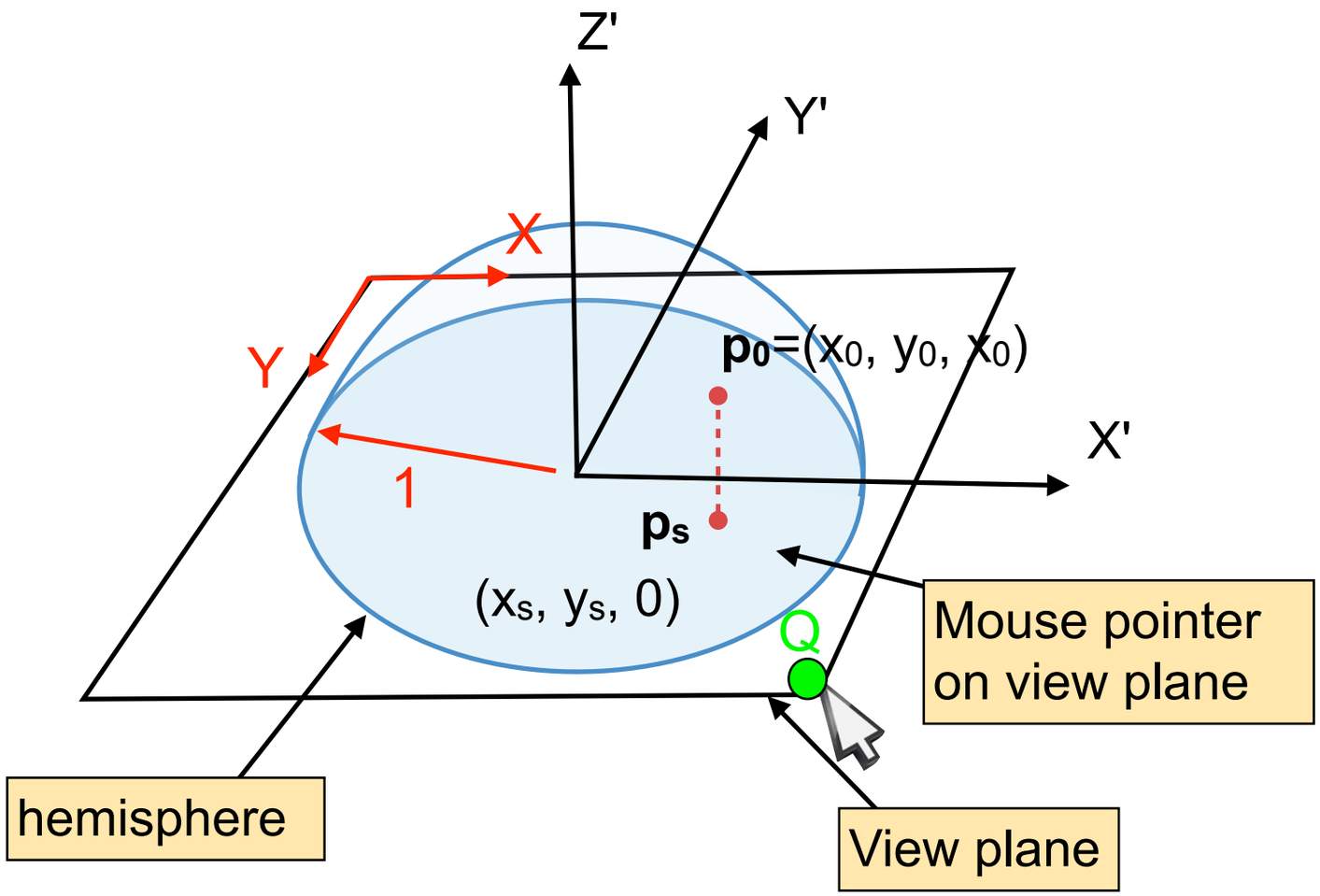


$$a^2 + b^2 = c^2$$

Trackball Manipulator

ARLAB

1. Find the start point for the rotation



- Transfer the screen point ps to coordinate for the hemisphere (clamp it between -1 and 1)

$$x = \frac{2 \cdot x - window\_width}{window\_width}$$

$$y = \frac{2 \cdot y - window\_height}{window\_height}$$

$$z = \sqrt{z - length}$$

$$length = x^2 + y^2$$

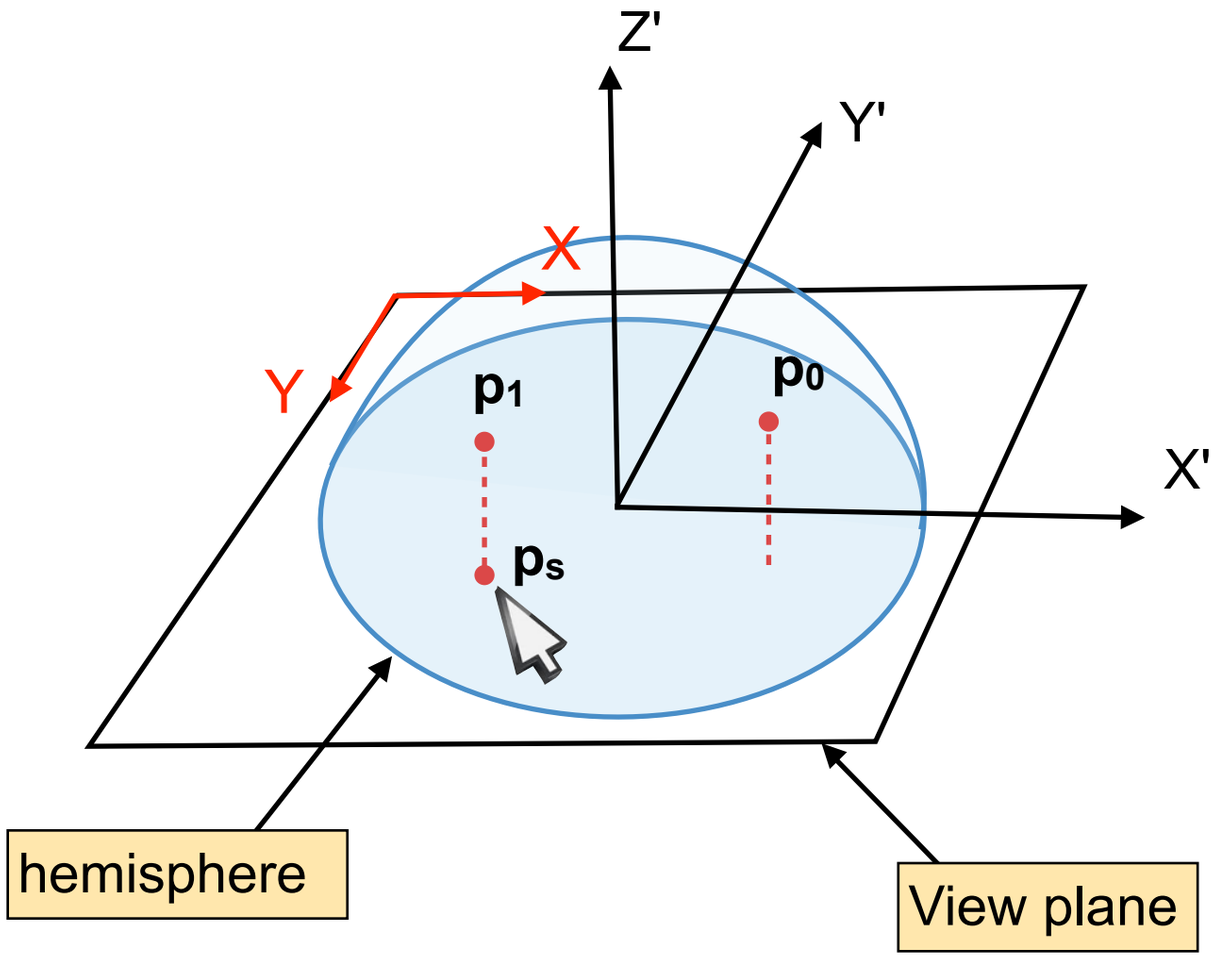
Result in a value between -1 and 1

Question: which value do we get at point Q?

Trackball Manipulator

ARLAB

2. In successive frames, look for the second point

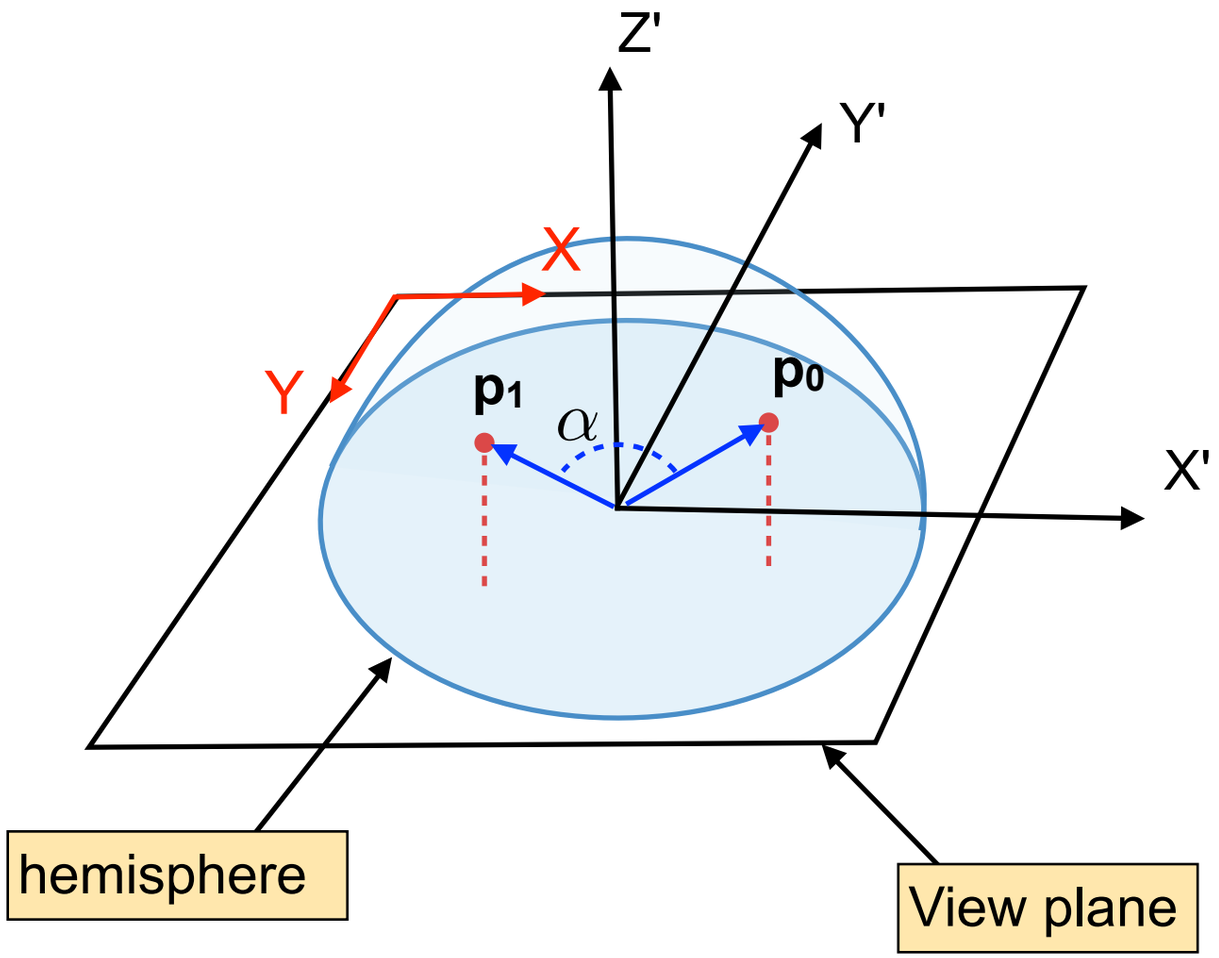


- Keep track of the previous mouse position $\mathbf{p}_1$ and calculate the current one $\mathbf{p}_2$.
- Repeat the same calculation to transfer the point to normalized screen coordinates.

Trackball Manipulator

ARLAB

3. Calculate the angle between both vectors.



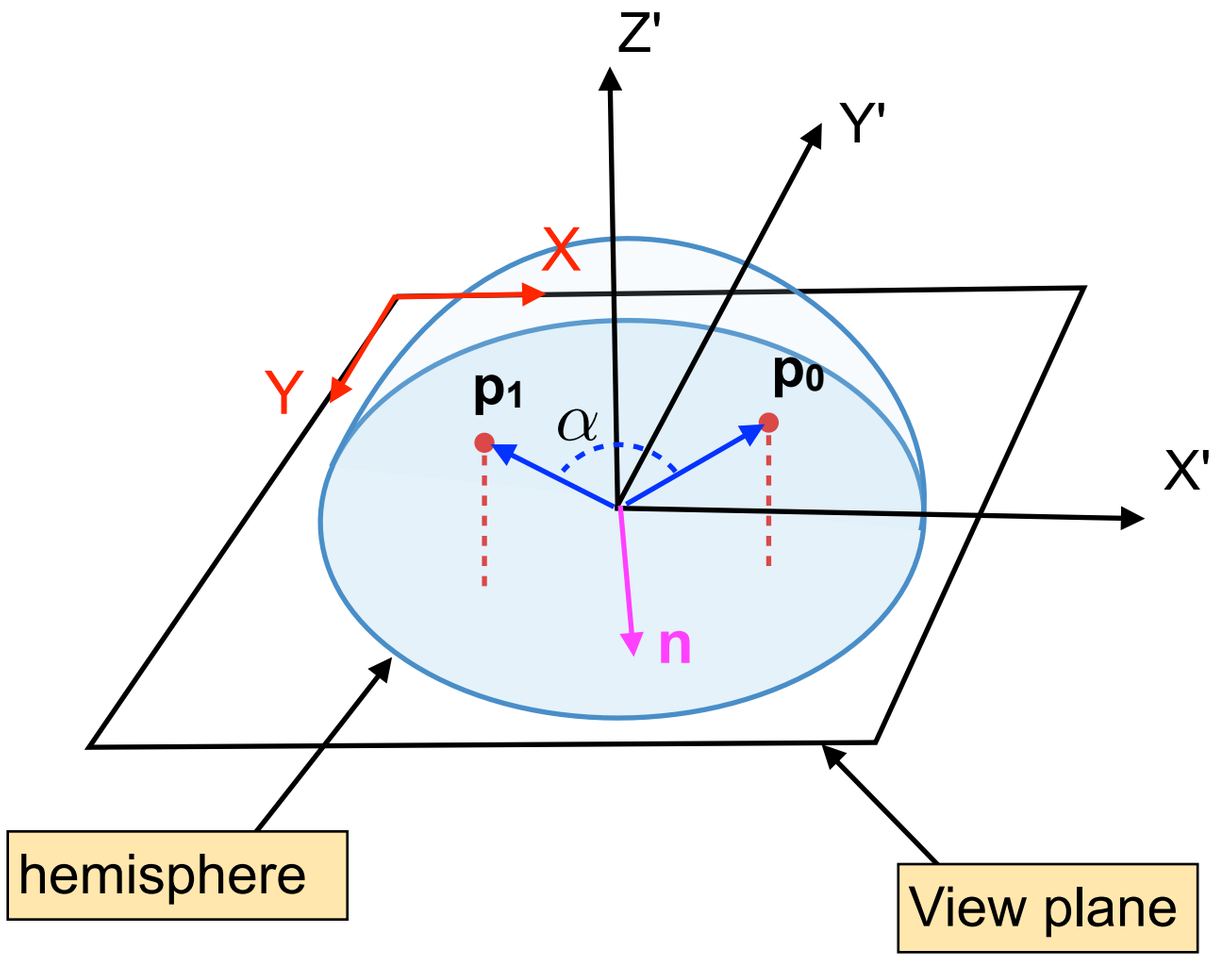
- Calculate the angle as dot product between the two vectors $\mathbf{p_0}$ and $\mathbf{p_1}$.

$$\alpha = \mathbf{p}_0 \cdot \mathbf{p}_1$$

Trackball Manipulator

ARLAB

4. Calculate the rotation axis



- Calculate the angle as dot product between the two vectors $\mathbf{p_0}$ and $\mathbf{p_1}$.

$$\alpha = \mathbf{p}_0 \cdot \mathbf{p}_1$$

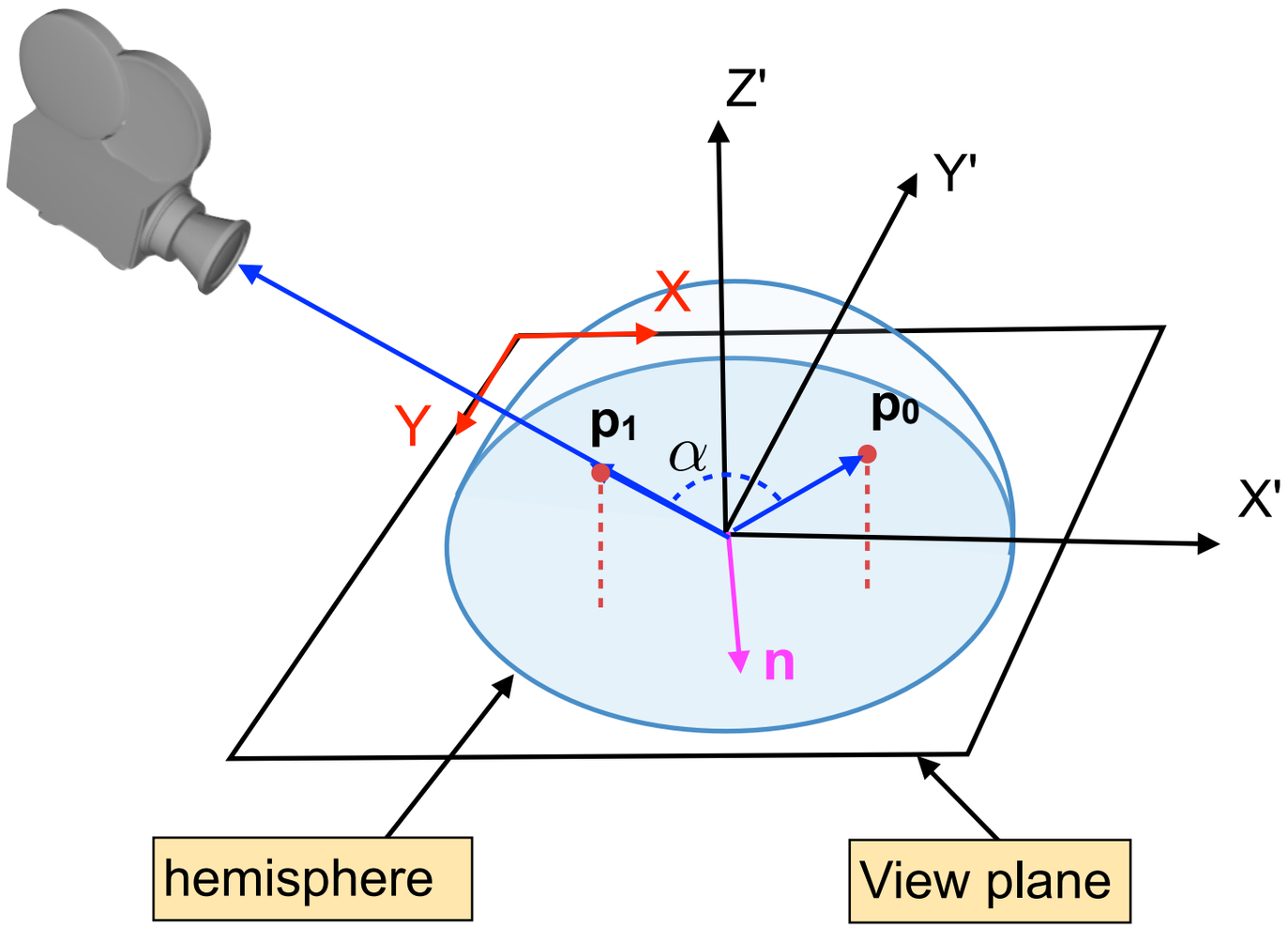
- Calculate the axis of rotation as cross product between the two vectors $\mathbf{p_0}$ and $\mathbf{p_1}$.

$$\mathbf{n} = \mathbf{p}_0 \times \mathbf{p}_1$$

Trackball Manipulator

ARLAB

5. Calculate a rotation matrix



- Rotate a unit matrix around the given angle

Rodrigues rotation

allows us to construct a rotation matrix from a vector.

$$\mathbf{n} = \{n_x, n_y, n_z\}$$

Let

$$W = \begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix}$$

be the Rodrigues rotation matrix,

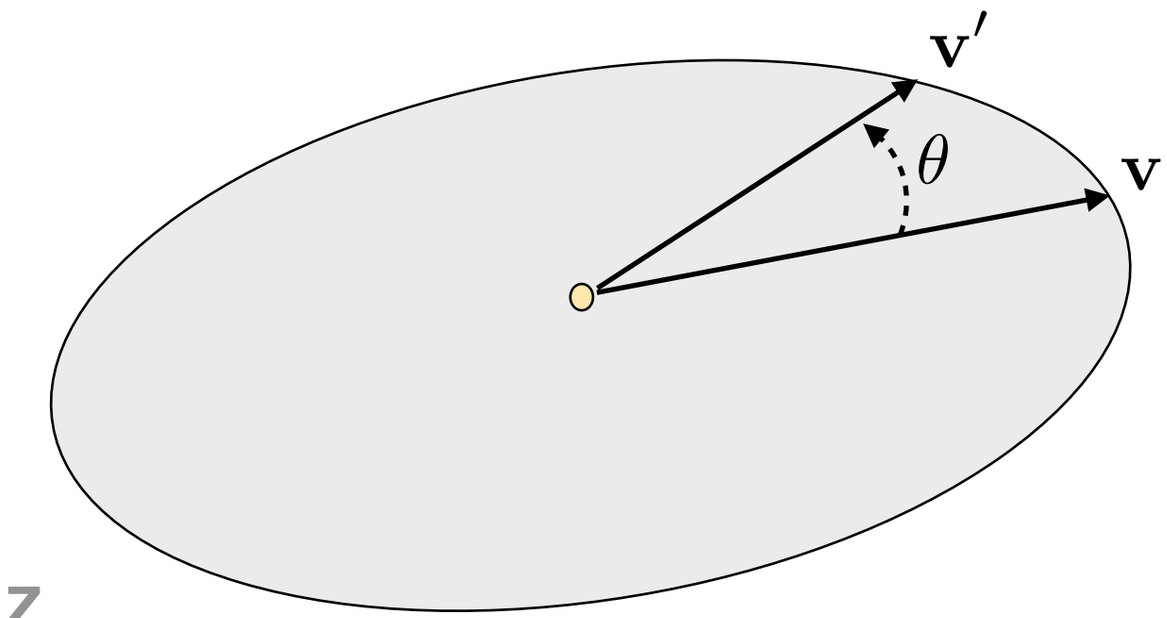
$$\mathbf{R} = \mathbf{I} + (\sin\alpha)\mathbf{W} + (1 - \cos\alpha)\,\mathbf{W}^2$$

is the rotation matrix

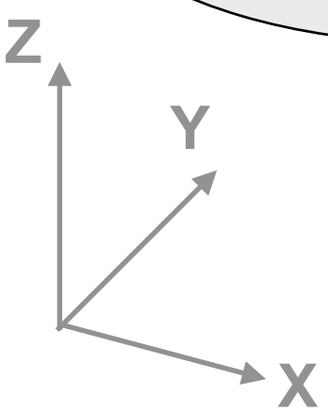
Axis-Angle Description or Rodrigues Rotation

ARLAB

Every rotation in 3d-space can be represented as a rotation about some axis by some angle.



- We want to rotate the point **v** to **v'**
- It is an arbitrary point, the rotation is not axis aligned.

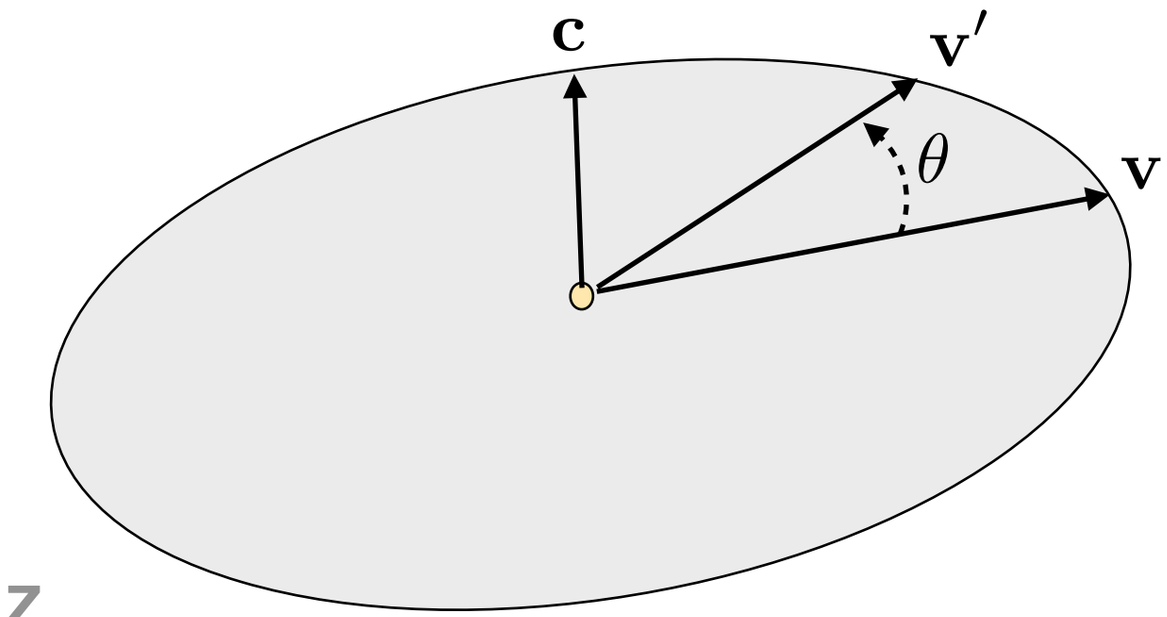


world coordinate system

Axis-Angle Description or Rodrigues Rotation

ARLAB

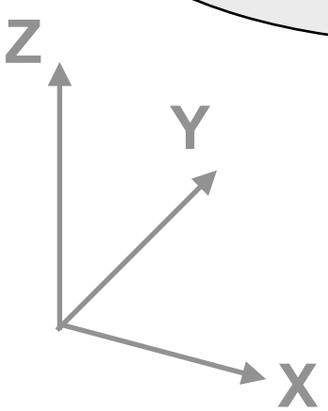
Every rotation in 3d-space can be represented as a rotation about some axis by some angle.



- Assume a vector **c** perpendicular to **v**
- We can show that

$$\mathbf{v}' = \cos(\theta)\mathbf{v} + \sin(\theta)\mathbf{c}$$

- How do we get **c** ?

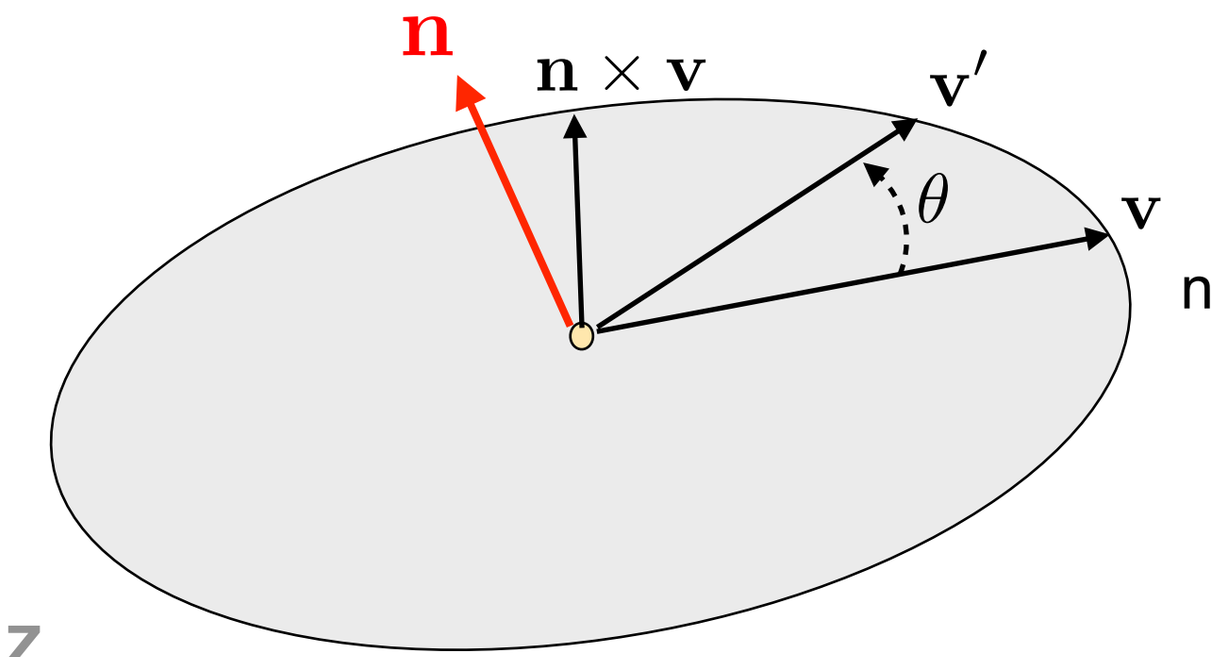


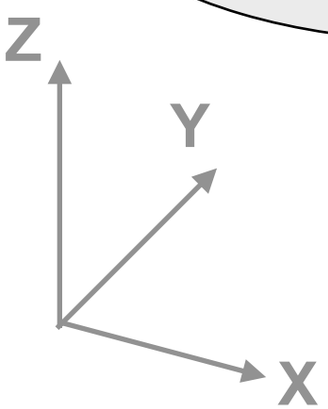
world coordinate system

Axis-Angle Description or Rodrigues Rotation

ARLAB

Every rotation in 3d-space can be represented as a rotation about some axis by some angle.





world coordinate system

- **n** is our rotation axis:

$$\mathbf{n} = \{n_x, n_y, n_z\}$$

- **c** = (**n** x **v**)
- **v'** is

$$\mathbf{v}' = \cos(\theta)\mathbf{v} + \sin(\theta)(\mathbf{n} \times \mathbf{v})$$

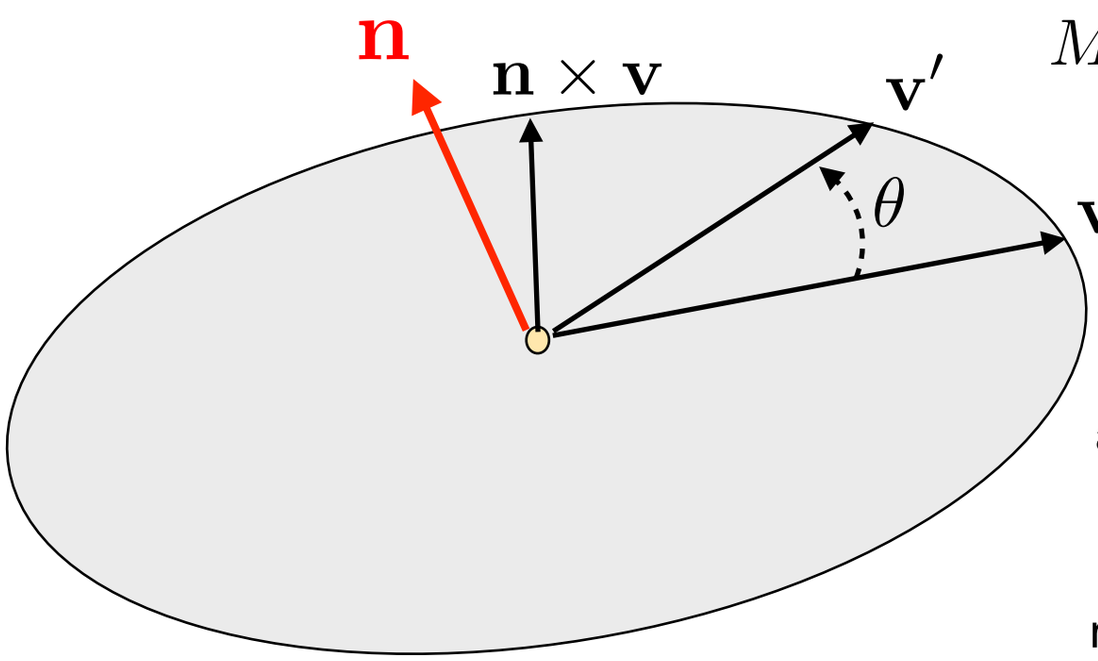
$$M\ \mathbf{v} = \cos(\theta)\mathbf{v} + \sin(\theta)(\mathbf{n} \times \mathbf{v})$$

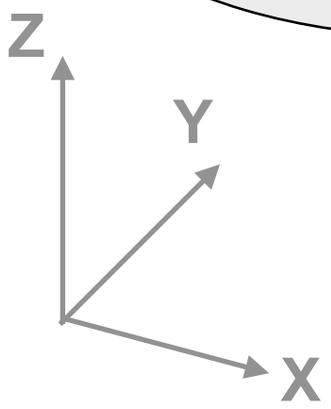
with *M*, a transformation matrix for:

$$\mathbf{v}' = M\ \mathbf{v}$$

Axis-Angle Description or Rodrigues Rotation

Every rotation in 3d-space can be represented as a rotation about some axis by some angle.





world coordinate system

$$M\ \mathbf{v} = \cos(\theta)\mathbf{v} + \sin(\theta)(\mathbf{n} \times \mathbf{v})$$

geometric re-interpretation

$$= \mathbf{v} + (1 - \cos(\theta))(-\mathbf{v}) + \sin(\theta)(\mathbf{n} \times \mathbf{v})$$

$$= \mathbf{v} + (1 - \cos(\theta))(\mathbf{n} \times (\mathbf{n} \times \mathbf{v})) + \sin(\theta)(\mathbf{n} \times \mathbf{v})$$

anti-symmetric matrix

$$= I\mathbf{v} + (1 - \cos(\theta))W^2\mathbf{v} + \sin(\theta)W\mathbf{v}$$

re-arranged

$$= I\mathbf{v} + \sin(\theta)W\mathbf{v} + (1 - \cos(\theta))W^2\mathbf{v}$$

remove **v**

$$M = I + (\sin\alpha)W + (1 - \cos\alpha)\,W^2$$

with *M*, the rotation matrix

Anti-symmetric Matrix / Skew-symmetric Matrix

ARLAB

Symmetric matrix

$$M = M^T$$

$$\begin{pmatrix} 1 & 5 & 7 \\ 5 & 2 & -6 \\ 7 & -6 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 5 & 7 \\ 5 & 2 & -6 \\ 7 & -6 & 8 \end{pmatrix}$$

Anti-symmetric matrix

$$A = -A^T$$

$$\begin{pmatrix} 0 & 5 & -7 \\ -5 & 0 & 3 \\ 7 & -3 & 0 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 0 & -5 & 7 \\ 5 & 0 & -3 \\ -7 & 3 & 0 \end{pmatrix} \rightarrow -A^T = \begin{pmatrix} 0 & 5 & -7 \\ -5 & 0 & 3 \\ 7 & -3 & 0 \end{pmatrix}$$

Vector to matrix

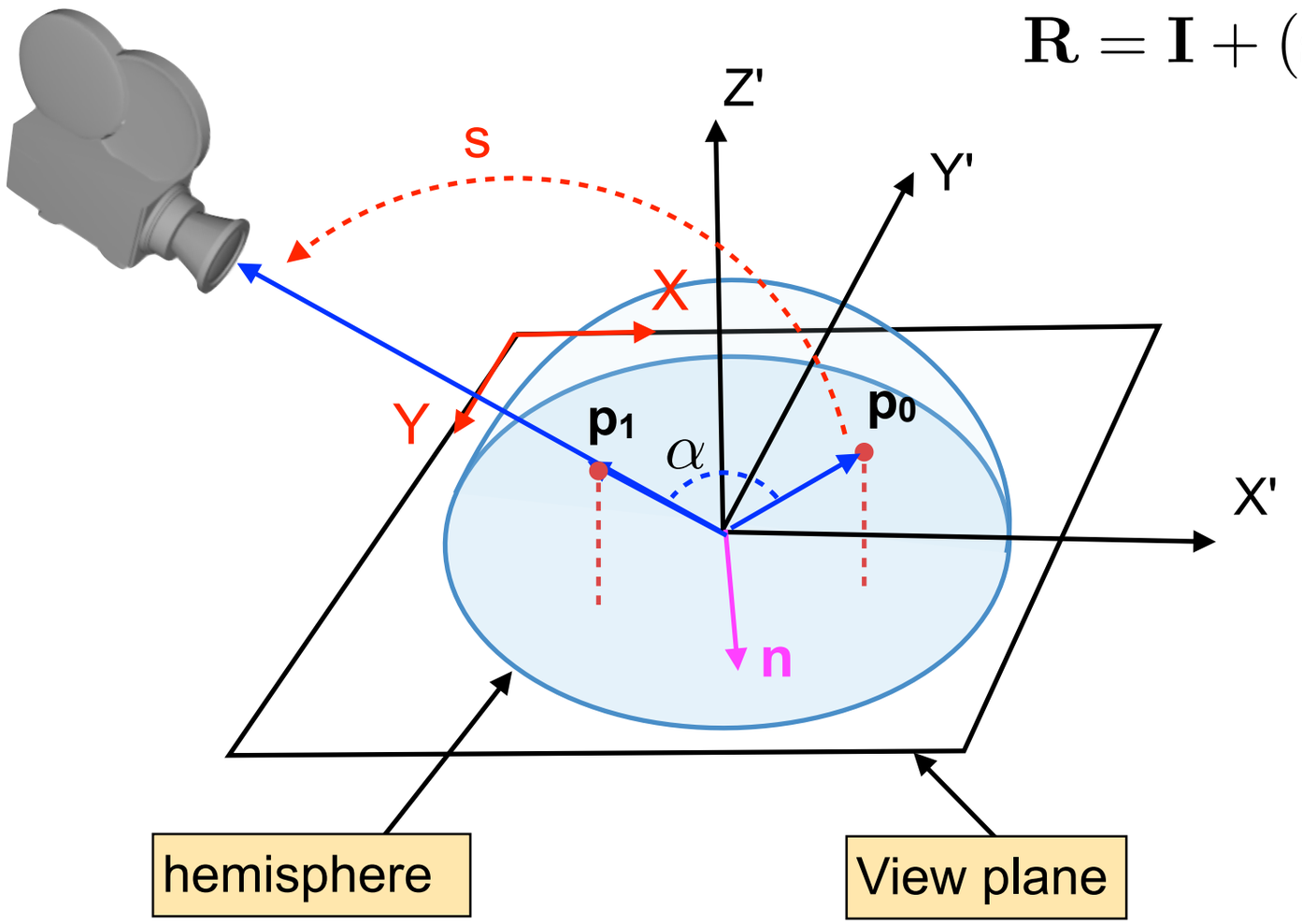
$$skew \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} = \begin{pmatrix} 0 & -e_z & e_y \\ e_z & 0 & -e_x \\ -e_y & e_x & 0 \end{pmatrix}$$

The cross product of two vectors can be represented as a multiplication

$$\mathbf{a} \times \mathbf{b} = skew(\mathbf{a}) \cdot \mathbf{b}$$

Trackball Manipulator

5. Calculate a rotation matrix + SPEED



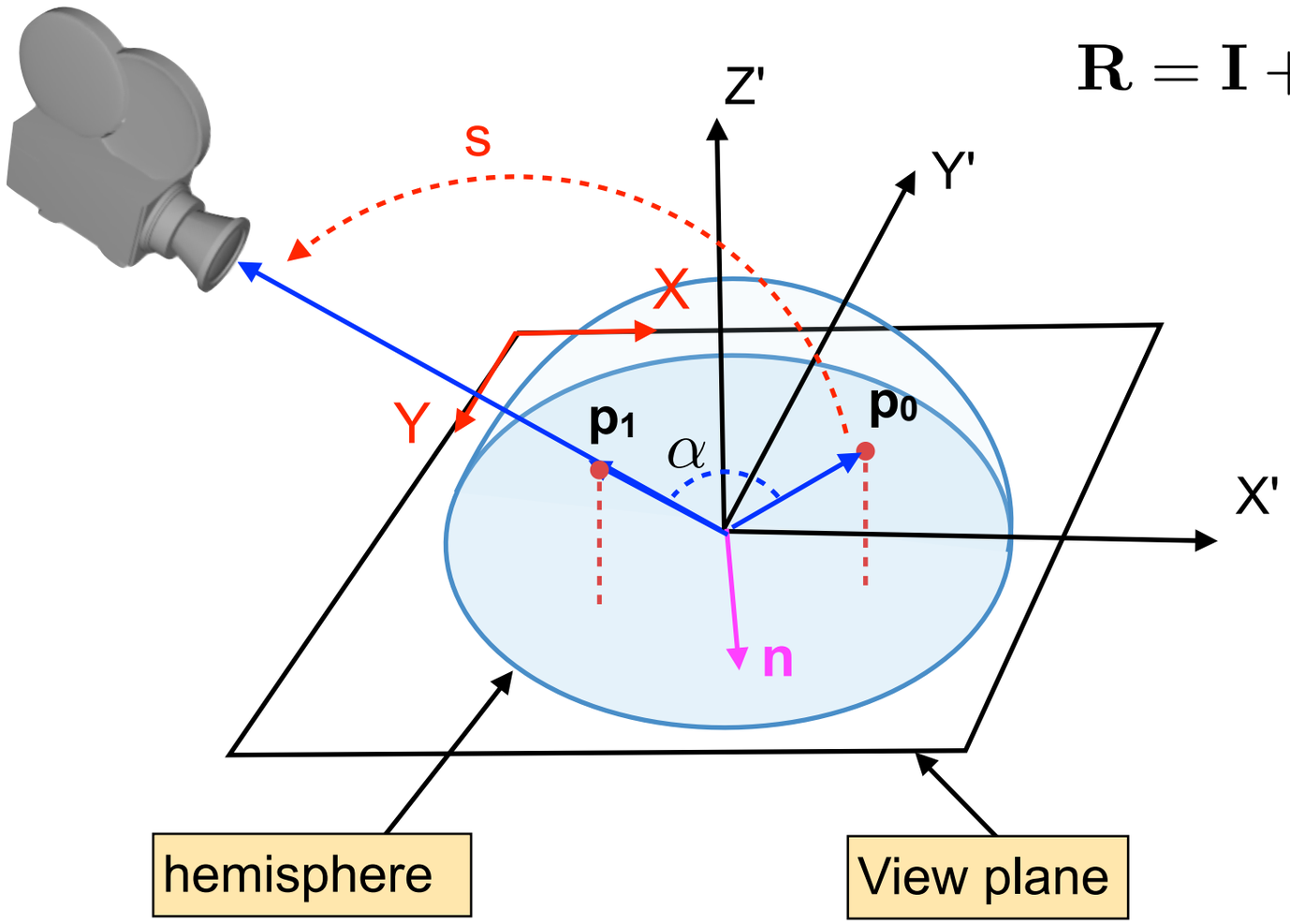
$$\mathbf{R} = \mathbf{I} + (\sin(\alpha \cdot s))\mathbf{W} + (1 - \cos(\alpha \cdot s))\,\mathbf{W}^2$$

- Multiply the angle with a constant values s, s for speed, to accelerate the rotation.
- The constant gives you better motion control.
- note R is the rotation matrix

Trackball Manipulator

ARLAB

6. Multiply R with the view matrix



$$\mathbf{R} = \mathbf{I} + (\sin(\alpha \cdot s))\mathbf{W} + \left(2\sin^2\frac{\alpha \cdot s}{2}\right)\mathbf{W}^2$$

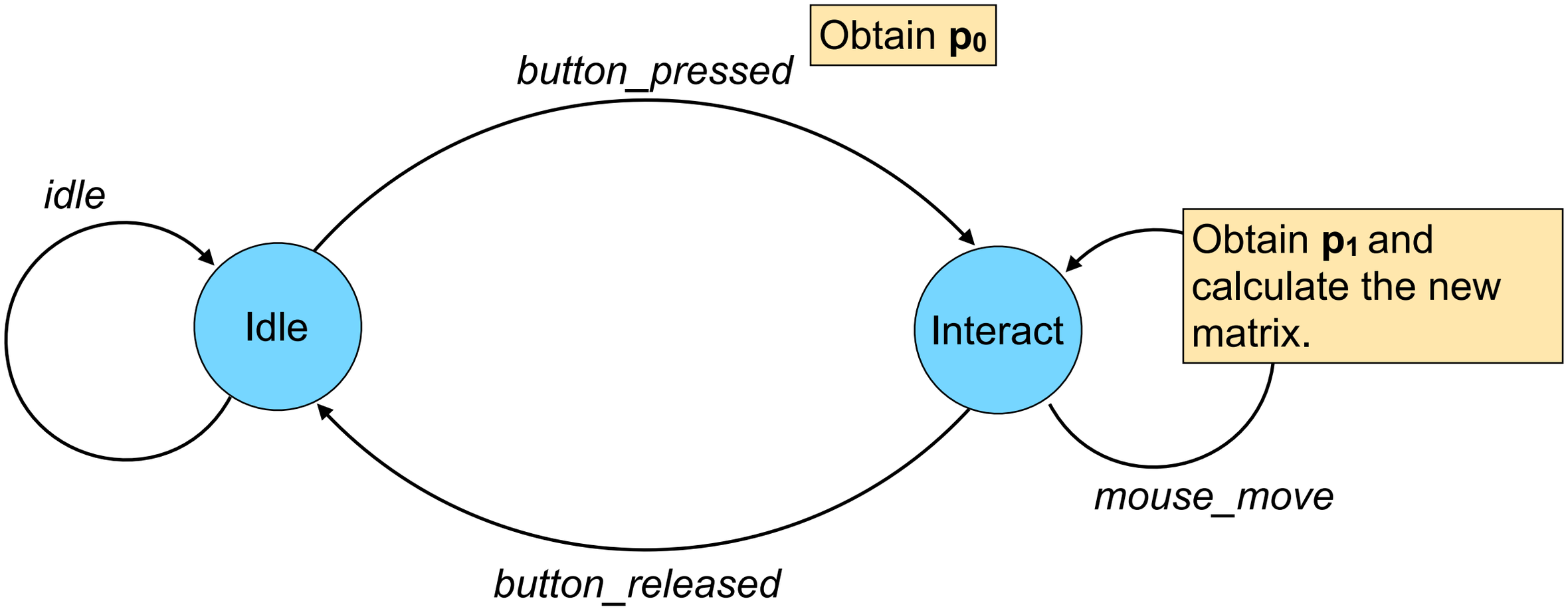
- Multiply the angle with a constant values s, s for speed, to accelerate the rotation.
- The constant gives you better motion control.

- The matrix R represents the delta between your original camera position and the new one. Multiply it with the original matrix and send it to your graphics program.

$$\mathbf{VM}_{new} = \mathbf{R} * \mathbf{VM}_{old}$$

State Machines

The typical processes for an interaction process:



State Transition

ARLAB

Code Example

CameraManipulator

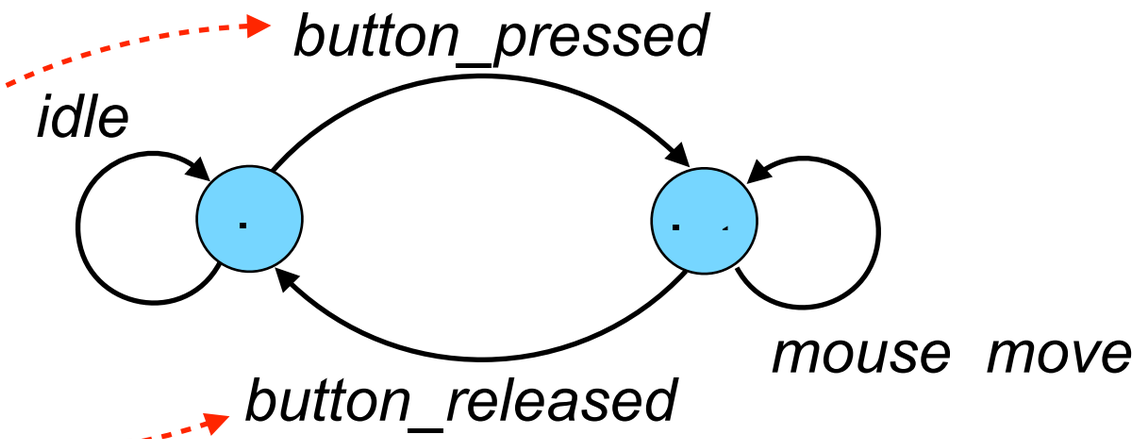
ARLAB

The name of the class is *CameraManipulator*:

Important variables:

Keep the state of the buttons;
they are our State Machine

```
int _leftMouseButtonEvent
int _rightMouseButtonEvent
```



Start position when pressing the button

```
glm::vec3 _startPos
```

Current position when moving the mouse

```
glm::vec3 _currentPos
```

camera_MouseButton_Callback

```
void CameraManipulator::camera_MouseButton_Callback(GLFWwindow * window, int button, int action, int mods)
{
    switch (action) {
      case GLFW_PRESS: // 1
        if(button == GLFW_MOUSE_BUTTON_1) // Left mouse button = 0
        {
            // switch mouse button event to 1 when the mouse button is pressed
            _leftMouseButtonEvent = 1;

        }
        else if(button == GLFW_MOUSE_BUTTON_2) // Right mouse button = 1
        {
            _rightMouseButtonEvent = 1;
        }

        break;
      case GLFW_RELEASE: // 0
        if(button == GLFW_MOUSE_BUTTON_1) // Left mouse button = 0
        {
            // switch mouse button event to 0 when the mouse button is release
            _leftMouseButtonEvent = 0;

        }
        else if(button == GLFW_MOUSE_BUTTON_2) // Right mouse button = 1
        {
            _rightMouseButtonEvent = 0;
        }

        break;
    }
}
```

Button pressed

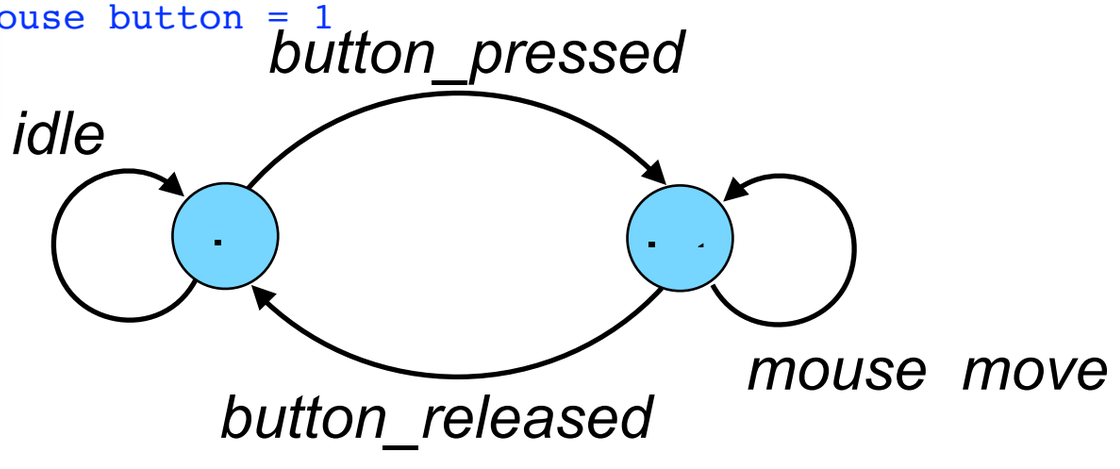
Left button

Right button

Button released

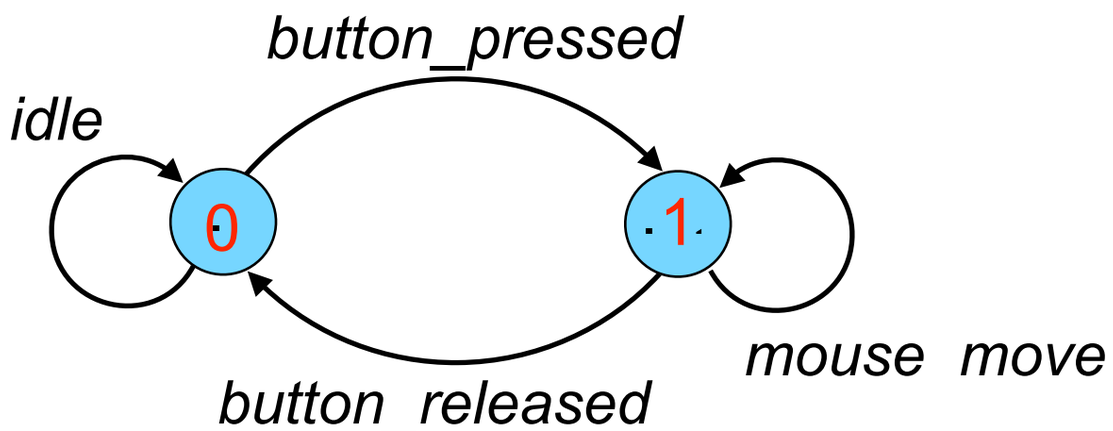
Left button

Right button



camera_MouseButton_Callback

```
switch (action) {
        case GLFW_PRESS: // 1
            if(button == GLFW_MOUSE_BUTTON_1) // Left mouse button = 0
            {
                // switch mouse button event to 1 when the mouse button is pressed
                _leftMouseButtonEvent = 1;

            }
            [...]
            break;
        case GLFW_RELEASE: // 0

            if(button == GLFW_MOUSE_BUTTON_1) // Left mouse button = 0
            {
                // switch mouse button event to 0 when the mouse button is release
                _leftMouseButtonEvent = 0;

            }
            [...]
            break;

}
```



C++ switch / case

The switch - case function is a control structure that uses constant expressions to switch between several possible values or statements.

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
  .
  .
  default:
     default group of statements
}
```

It works as follows:

1. switch evaluates expression and checks if it is equivalent to constant1. If it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.
2. If expression was not equal to constant1 it will be checked the next constants until it reaches the default statement or the switch statement ends

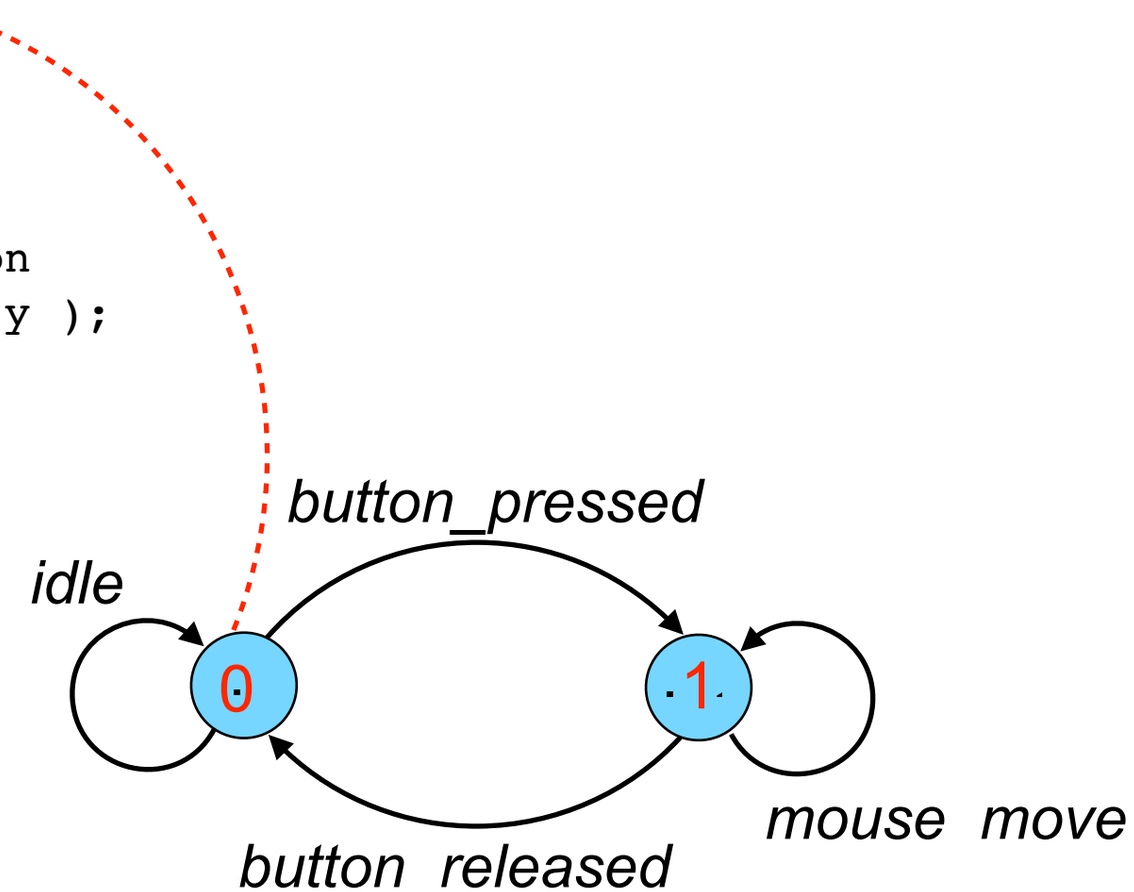
Never forget a break! C++ will continue to execute the code, if no break has been set.

camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
      ///-----------------------------
       // Camera orientation starts here
      switch (_leftMouseButtonEvent) {
          case 0:
              return;
              break;
      case 1:
          //  remember the first position
          _startPos = toWindowCoord( x, y );

          // switch to state 2
          _leftMouseButtonEvent = 2;
          return;
          break;
 }
```

This function gets called every frame!

State 0: we do nothing and return / idle

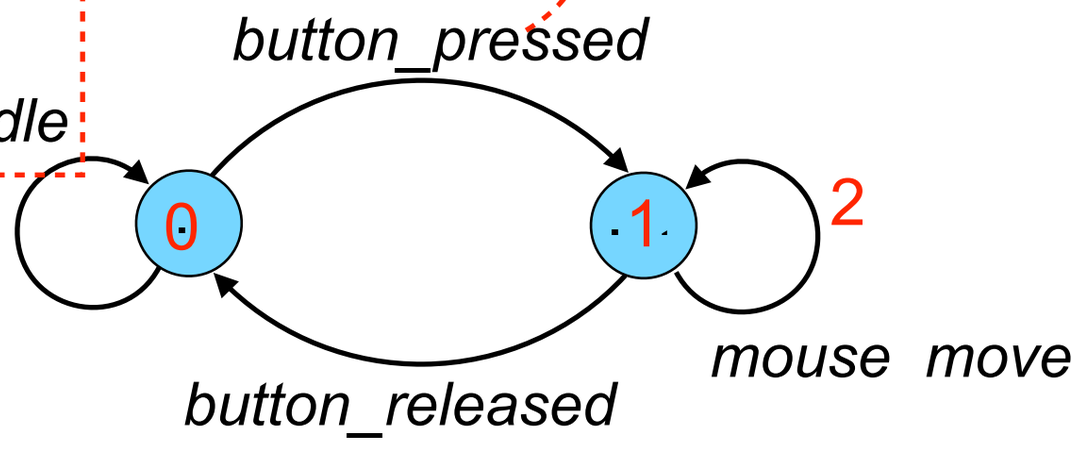


camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
      ///-----------------------------
       // Camera orientation starts here

       switch (_leftMouseButtonEvent) {
           case 0:
               return;
               break;

       case 1:
           //  remember the first position
           _startPos = toWindowCoord( x, y );

           // switch to state 2
           _leftMouseButtonEvent = 2;
           return;
           break;
}
```

This function gets called every frame!

State 1: button pressed
We remember the start position, change to state 2, and return.



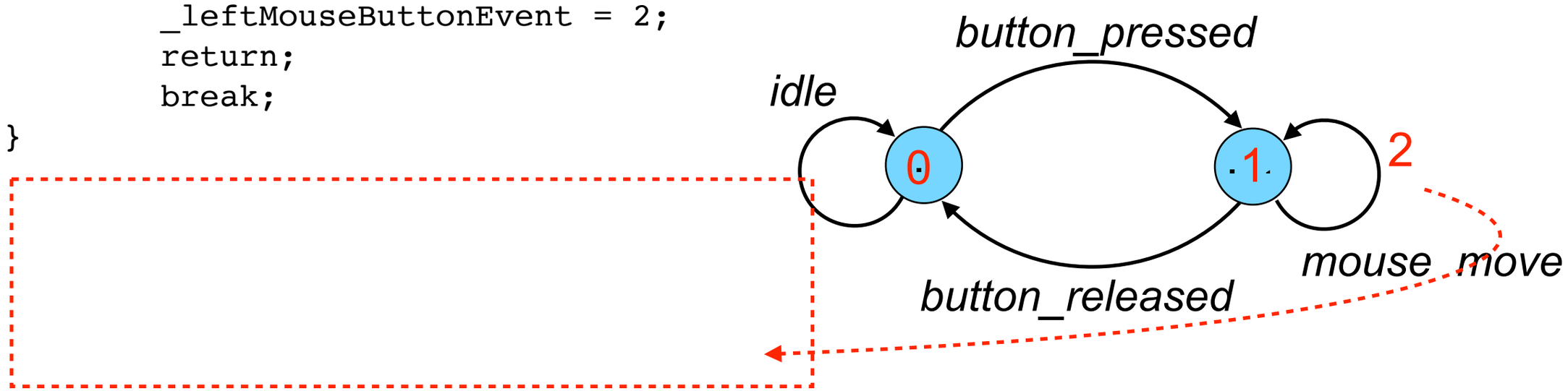
To distinguish state 1 and state 2 prevents that the application will store the start position an second time.

camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
      ///-----------------------------
       // Camera orientation starts here

       switch (_leftMouseButtonEvent) {
           case 0:
               return;
               break;

       case 1:
           //  remember the first position
           _startPos = toWindowCoord( x, y );

           // switch to state 2
           _leftMouseButtonEvent = 2;
           return;
           break;
}
```

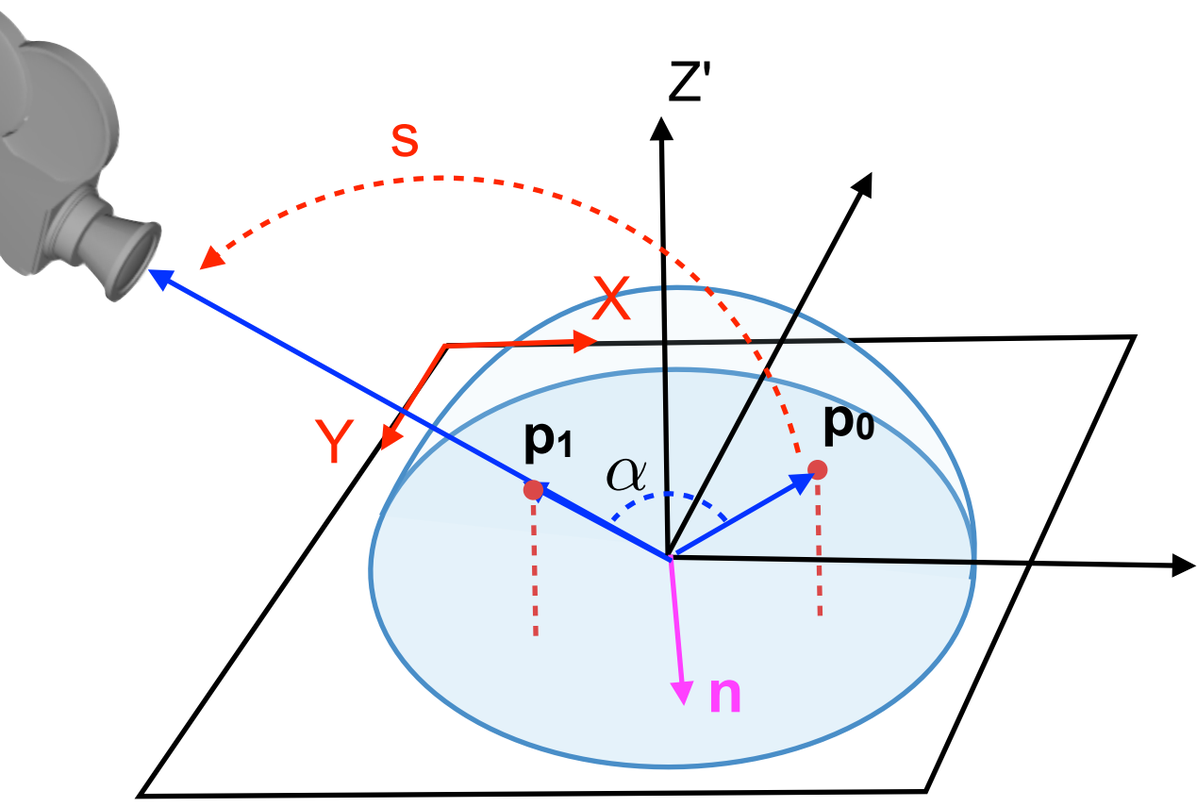
This function gets called every frame!

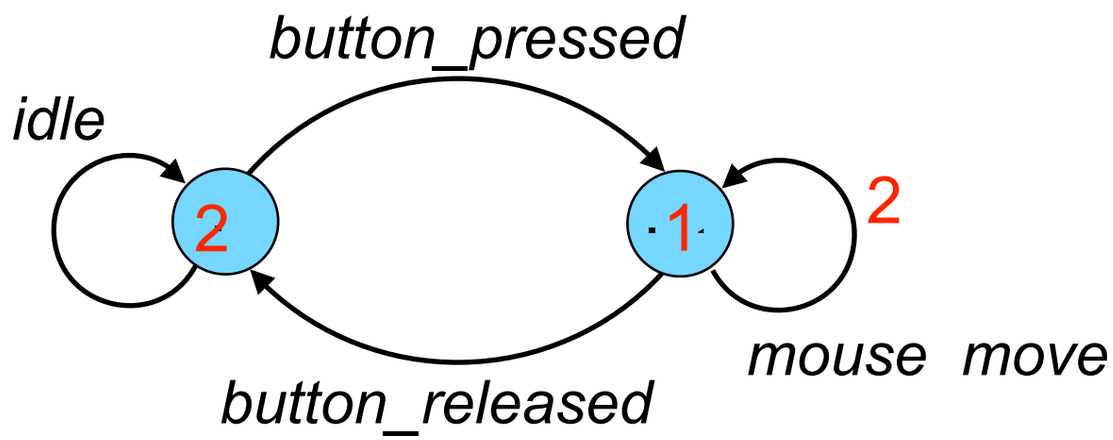
State 2: the switch / case control does not answer to state 2, we continue the code.



camera_MouseCursor_Callack

```
// state 2 is processed here

// Tracking the current mouse cursor
_currentPos  = toWindowCoord( x, y );

// Calculate the angle in radians, and clamp it between 0 and 90 degrees
_currentAngle    = acos( std::min(1.0f, glm::dot(_startPos, _currentPos) ));

// Cross product to get the rotation axis,
// but it's still in camera coordinate
 _cameraRotationAxis  = glm::cross( _startPos, _currentPos );
```



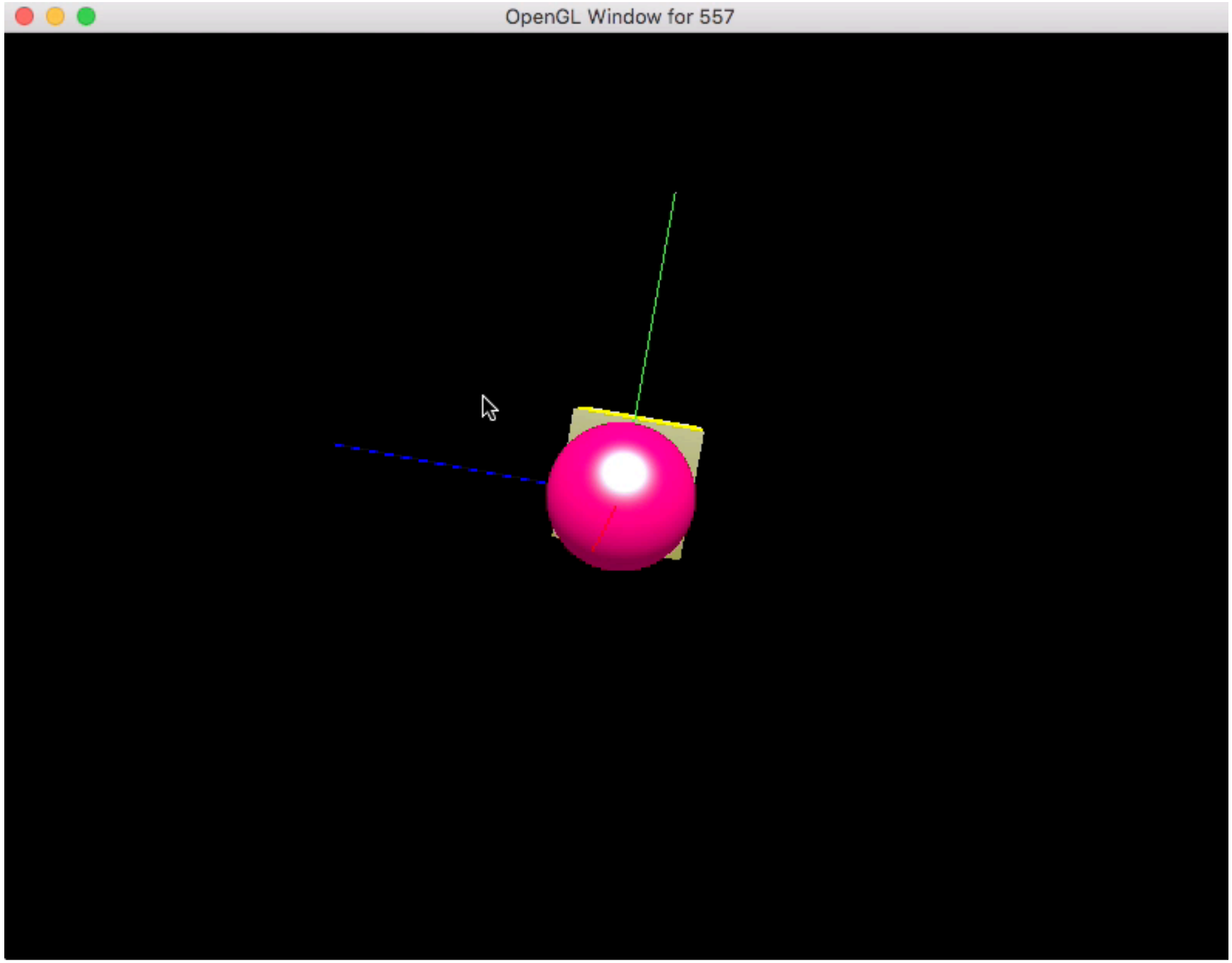


toWindowCoord

Take the coordinates in screen pixels and transfer them to sphere coordinates

```
glm::vec3 CameraManipulator::toWindowCoord( double x, double y )
{
    glm::vec3 coord(0.0f);

    coord.x =  (2 * x - _window_width ) / _window_width;
    coord.y = -(2 * y - _window_height) / _window_height;

    /* Clamp it to border of the windows */
    coord.x = glm::clamp( coord.x, -1.0f, 1.0f );
    coord.y = glm::clamp( coord.y, -1.0f, 1.0f );


    float length_squared = coord.x * coord.x + coord.y * coord.y;

    if( length_squared <= 1.0 )
        coord.z = sqrt( 1.0 - length_squared );
    else
        coord = glm::normalize( coord );

    return coord;
}
```

Video



Translations ... next week

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 5001, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY