ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Quaternions Interpolation & Navigation

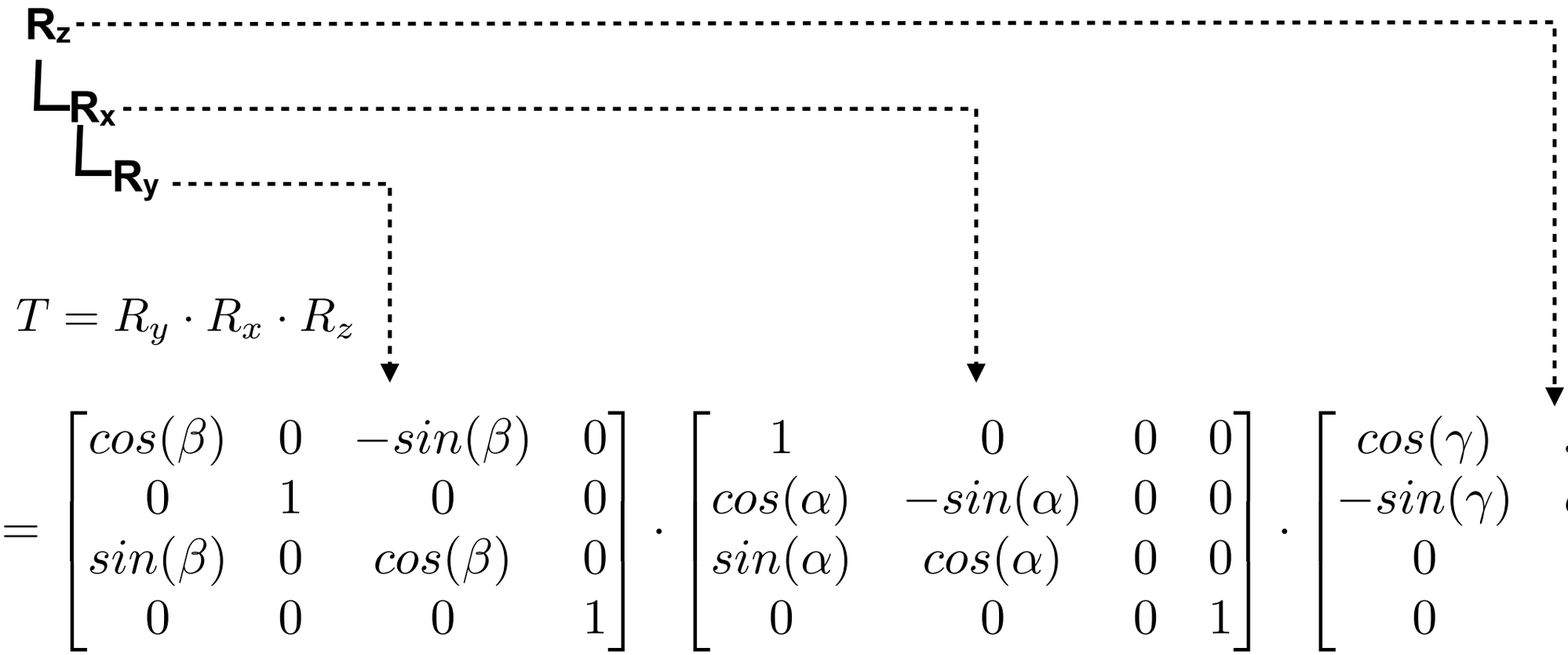
October 29th, 2015

Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Motivation

The sequence of transformations (rotations in our case) can be considered as hierarchy of transformations.



$$T = R_y \cdot R_x \cdot R_z$$

$$T = \begin{bmatrix} cos(\beta) & 0 & -sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ cos(\alpha) & -sin(\alpha) & 0 & 0 \\ sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos(\gamma) & sin(\gamma) & 0 & 0 \\ -sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Navigation is represented as a sequence of multiplication over time.

$$T = T_1\ T_2\ T_4\ \ldots\ T_n$$

Gimbal lock problem

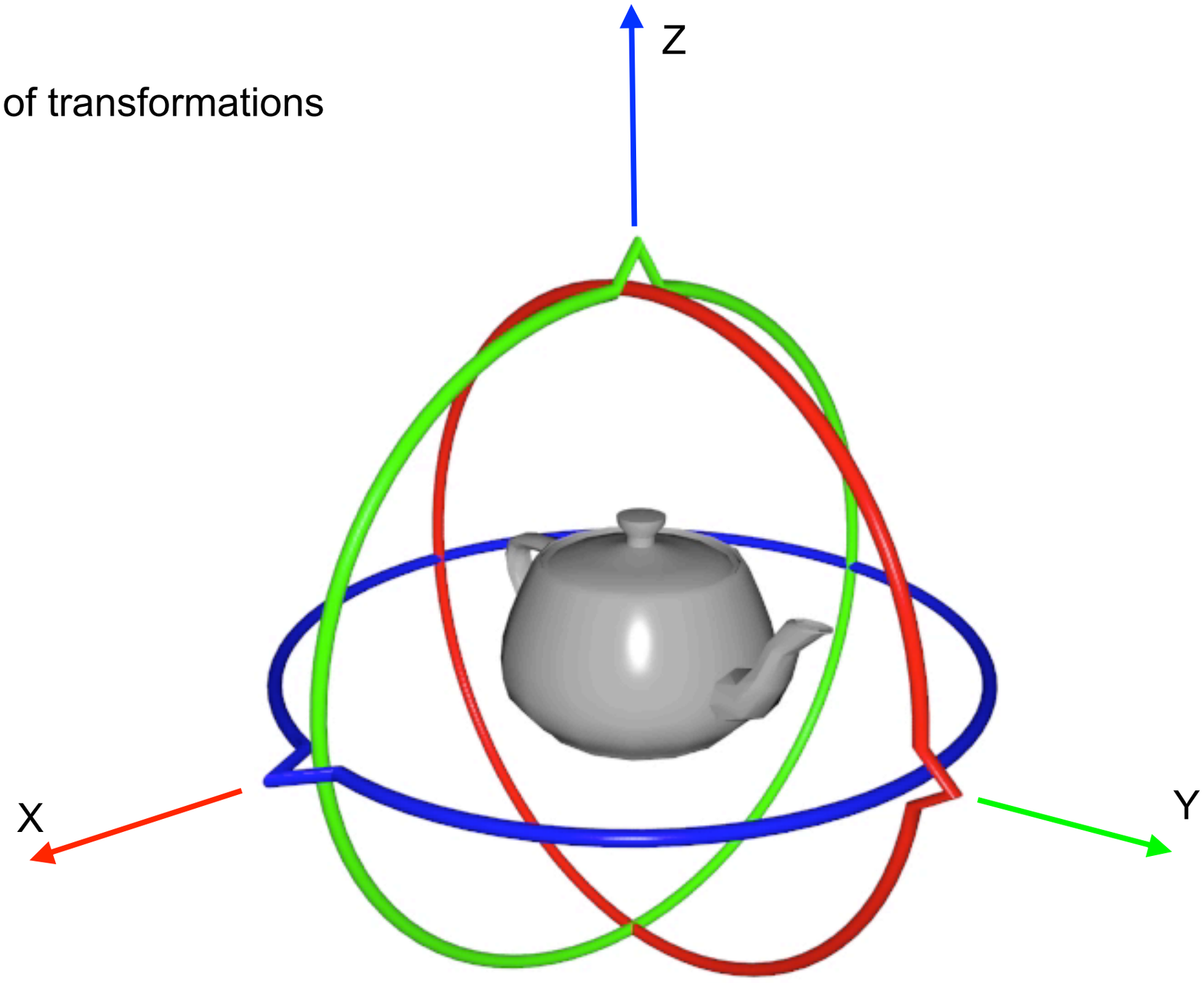
Gimbal Lock Problem

Hierarchy of transformations

Z

L_X

L_Y



Content

- SLERP Interpolation
- SQUAD Interpolation
- Quaternions in GLM
- Navigation without gimbal lock
- Translation (zoom) with Trackball

SLERP

SLERP stands for Spherical Linear Interpolation.

It provides a method to smoothly interpolate a point between two orientations.

$$q_1 = [s_1, \mathbf{a}] \quad q_2 = [s_2, \mathbf{b}]$$

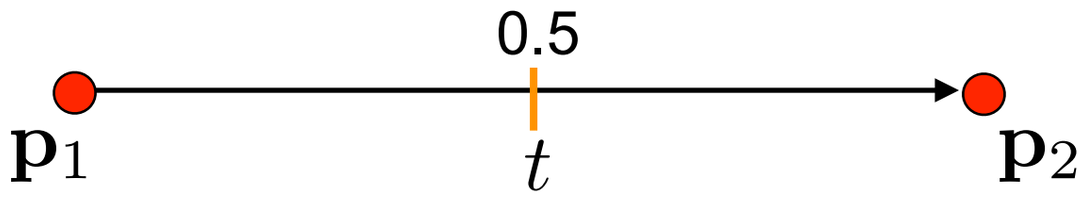
We will interpolate point **p**

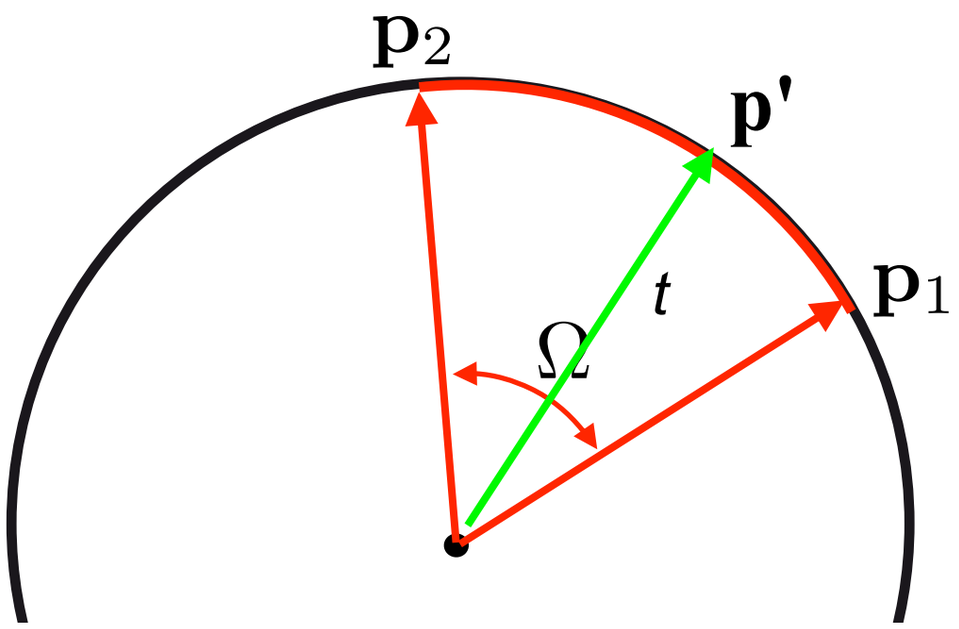
q_1 represents this point at *t=0*

q_2 represents this point at *t=1*

The standard linear interpolation equation is:

$$\mathbf{p}' = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$$





General steps:

- Compute the difference between p_2 and p_1.
- Take the fractional part of that difference.
- Adjust the original value by the fractional difference

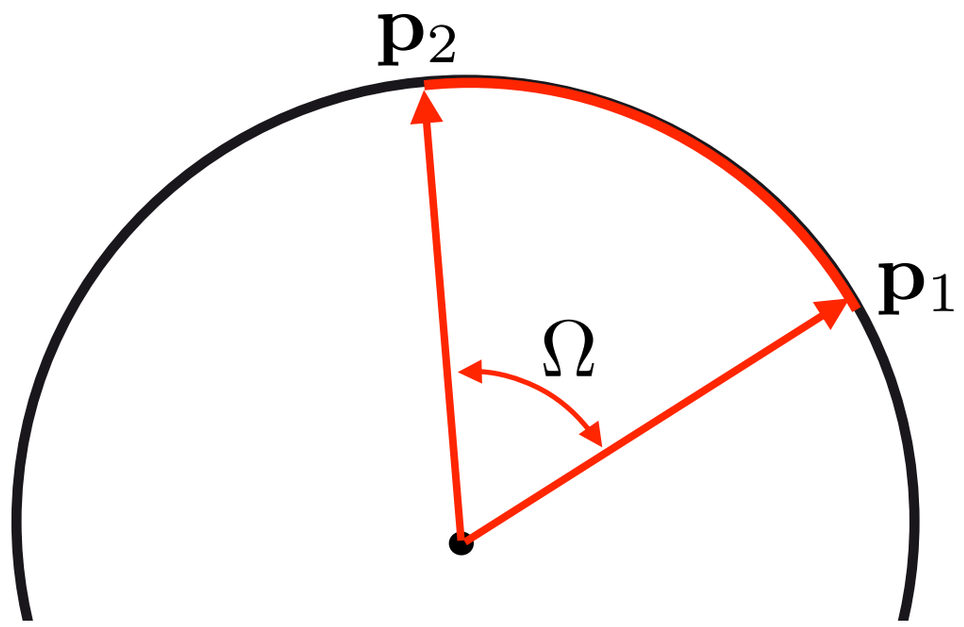
Difference between Quaternions

$$\mathbf{p}' = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$$

First, we must compute the difference between. With regards to quaternions, this is equivalent to computing the angular difference between the two quaternions.

Angular difference:

$$\Omega = q_1^{-1} q_2$$



Fractional part

$$\mathbf{p}' = \mathbf{p}_1 + \boxed{t(\mathbf{p}_2 - \mathbf{p}_1)}$$

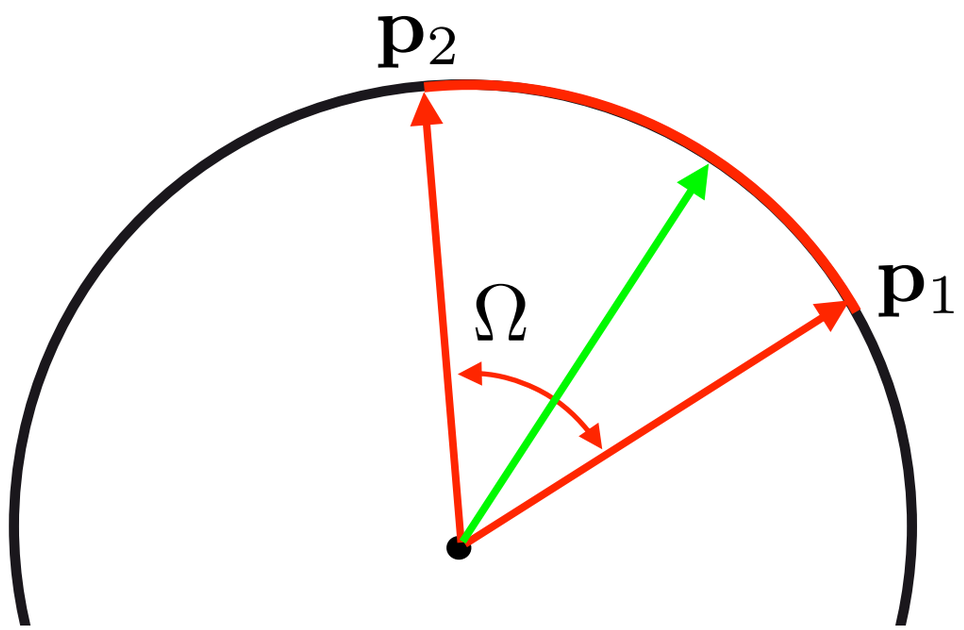
Next, we need to calculate the fractional part of the difference between both points.

We can compute the fractional part of a quaternion by raising it to a power whose value is in the range 0…1.

$$t = \Omega^t$$

which gives us:

$$= (q_1^{-1} q_2)^t$$



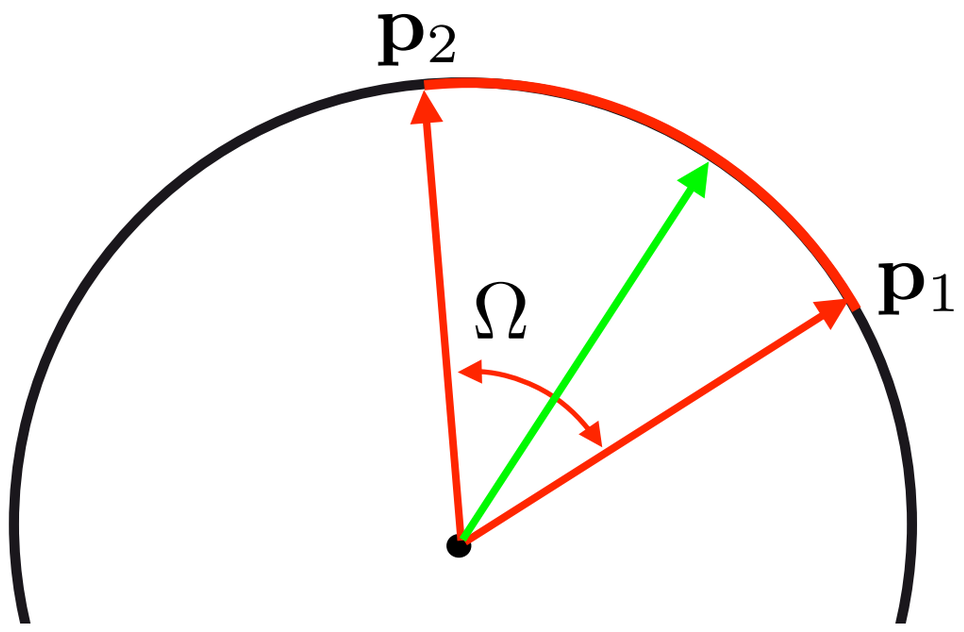
Quaternion Exponentiation

General formula for quaternion exponentiation is:

$$q^t = \exp(t \log q)$$

Where the exponential function for quaternions is given by:

$$\begin{aligned} \exp(q) &= \exp([0, \theta\hat{\mathbf{v}}]) \\ &= [\cos\theta, \sin\theta\mathbf{v}] \end{aligned}$$



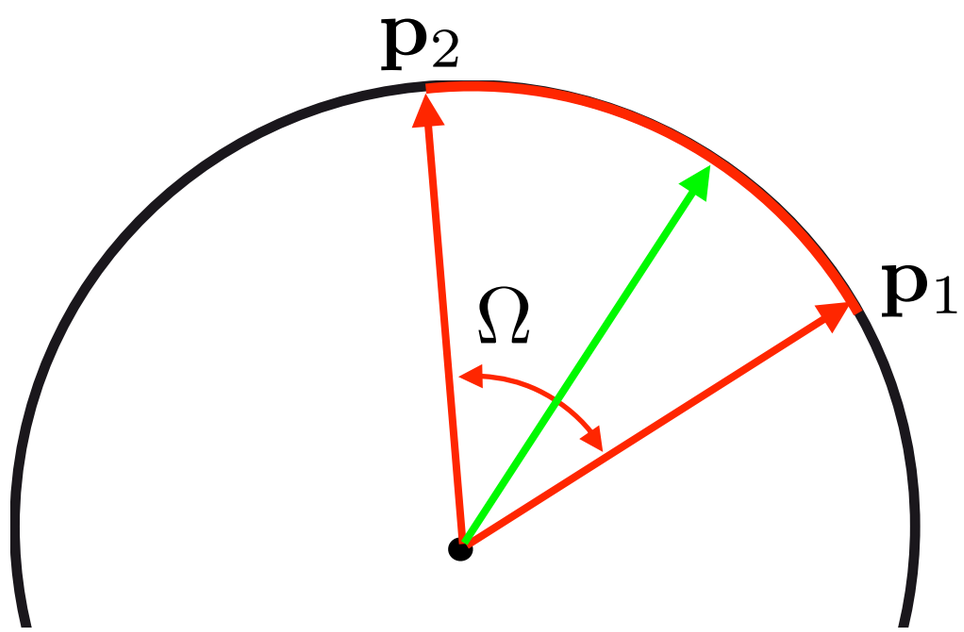
And the logarithm of a quaternion is given by:

$$\begin{aligned} \log q &= \log(\cos\theta + \sin\theta\hat{\mathbf{v}}) \\ &= \log(\exp(\theta\hat{\mathbf{v}})) \\ &= \theta\hat{\mathbf{v}} \\ &= [0, \theta\hat{\mathbf{v}}] \end{aligned}$$

Fractional part

For t=0

$$
\begin{aligned}
q^0 &= \exp(0, \log q) \\
&= \exp([\cos(0), \sin(0)\hat{\mathbf{v}}]) \\
&= \exp([1, 0]) \\
&= [1, 0]
\end{aligned}
$$



For t=1

$$
\begin{aligned}
q^1 &= \exp(\log q) \\
&= q
\end{aligned}
$$

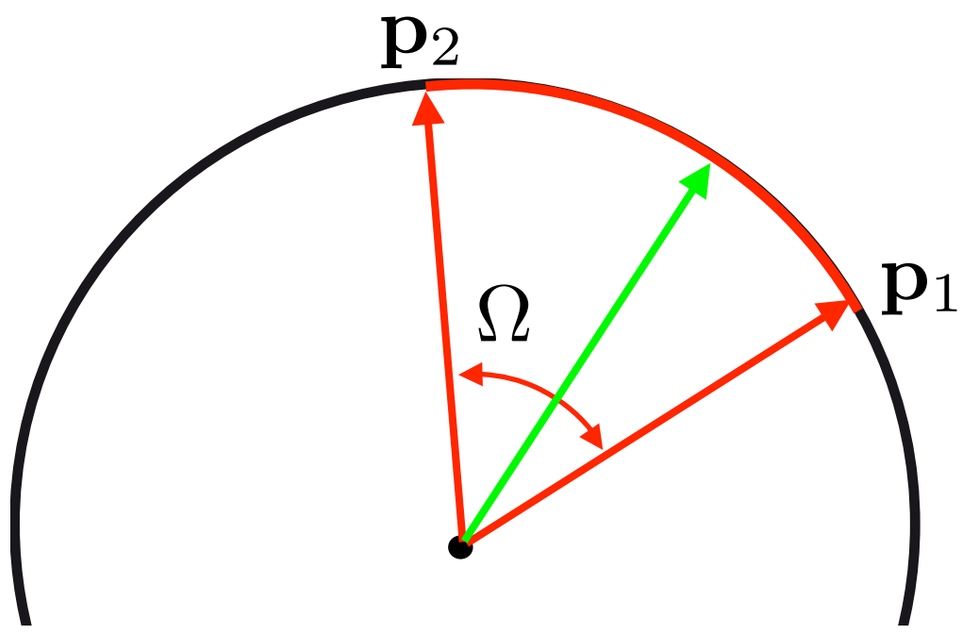
Fractional Difference

$$\mathbf{p}' = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$$

And to compute the interpolated angular rotation, we adjust the original orientation and adjust it by the fractional part

$$q' = q_1(q_1^{-1}q_2)^t$$

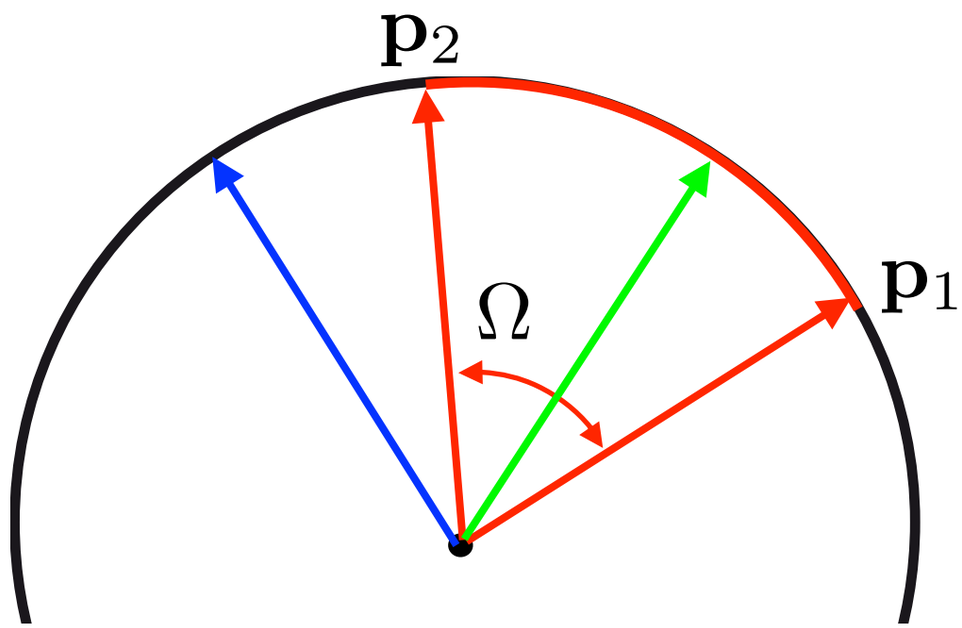
Which is the general form of spherical linear interpolation using **quaternions**.



Interpolation of points

The general form of a spherical interpolation for vectors is defined as:

$$\mathbf{p}_t = \frac{\sin(1-t)\theta}{\sin\theta}\mathbf{p}_1 + \frac{\sin t\theta}{\sin\theta}\mathbf{p}_2$$



And we can apply this also for quaternions:

$$q_t = \frac{\sin(1-t)\theta}{\sin\theta}q_1 + \frac{\sin t\theta}{\sin\theta}q_2$$

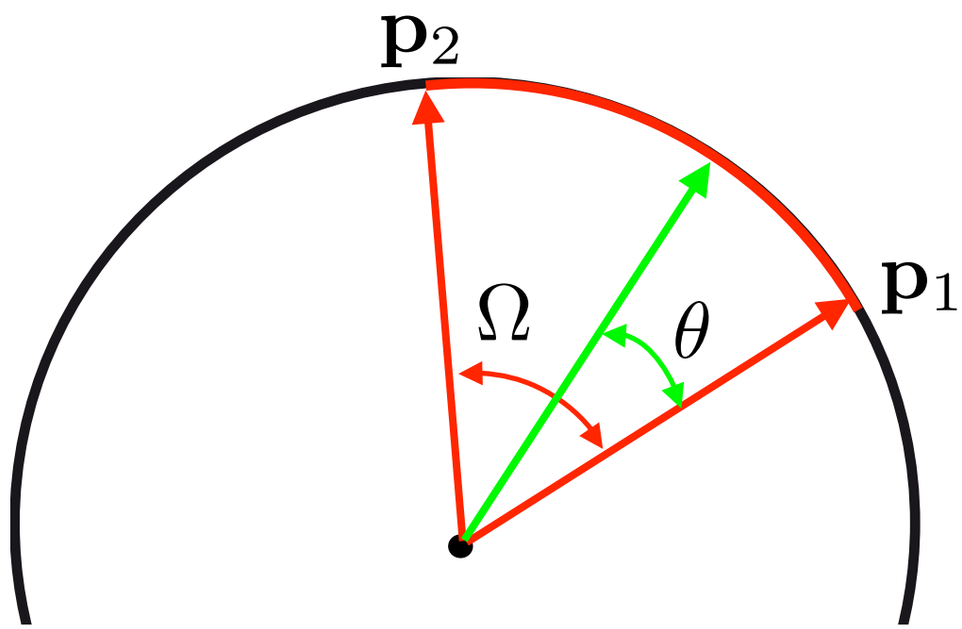
Angle

We can also obtain the angle as dot product between the two quaternions:

$$\cos\theta = \frac{q_1 \cdot q_2}{|q_1||q_2|}$$

$$= \frac{s_1 s_2 + x_1 x + 2 + y_1 y_2 + z_1 z_2}{|q_1||q_2|}$$

$$\theta = \cos^{-1}\left(\frac{s_1 s_2 + x_1 x + 2 + y_1 y_2 + z_1 z_2}{|q_1||q_2|}\right)$$



Note

- If the quaternion dot-product results in a negative value, then the resulting interpolation will travel the long-way around the 4D sphere which is not necessarily what we want.
- negate one of the orientations (negate scalar and vector part)
- If the difference between q1 and q2 is very small, the sinus is 0.
- User linear interpolation in this case.

Notation:

$$\mathrm{slerp}\{q_1, q_2, t\}$$

SQUAD

SQUAD (Spherical and Quadrangle) can be used to smoothly interpolate over a path of rotations.

Consider the sequence of quaternions:

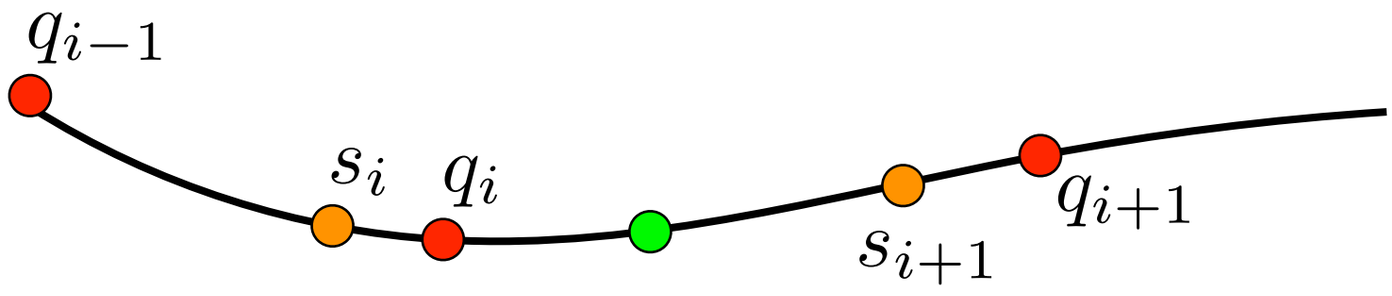
$$q_1, q_2, q_3, \ldots, q_{n-2}, q_{n-1}, q_n$$

And we also define the “helper” quaternion which we can consider an intermediate control point:

$$s_i = \exp\left(-\frac{\log(q_{i+1}q_i^{-1}) + \log(q_{i-1}q_i^{-1})}{4}\right) q_i$$

For animations

SQUAD



We can find any point in a given sequence of point by applying three SLERP interpolations.

$$\text{squad}(q_i, q_{i+1}, s_i, s_{i+1}, t) =$$
$$\text{slerp}(\text{slerp}(q_i, q_{i+1}, t), \text{slerp}(s_i, s_{i+1}, t), 2t(1-t))$$

Advantages

- Quaternion interpolation using SLERP and SQUAD provide a way to interpolate smoothly between orientations in space.
- Rotation concatenation using quaternions is faster than combining rotations expressed in matrix form.
- For unit-norm quaternions, the inverse of the rotation is taken by subtracting the vector part of the quaternion. Computing the inverse of a rotation matrix is considerably slower if the matrix is not normalized (if it is, then it's just the transpose of the matrix).
- Converting quaternions to matrices is slightly faster than for Euler angles.
- Quaternions only require 4 numbers (3 if they are normalized. The Real part can be computed at run-time) to represent a rotation where a matrix requires at least 9 values.

Disadvantages

- Quaternions can become invalid because of floating-point round-off error however this “error creep” can be resolved by re-normalizing the quaternion.
- They are very hard to understand.

But: GLM already supports quaternions

The advantages outweigh the disadvantages

ARLAB

Quaternions in GLM

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

glm::quat

ARLAB

`typedef detail::tquat< float >  quat`

Quaternion of floating-point numbers.

Functions:

- Returns the q conjugate.

  `conjugate` `(detail::tquat< T > const &q)`

- Returns the cross product of q1 and q2.

  `cross` `(detail::tquat< T > const &q1, detail::tquat< T > const &q2)`

- Returns dot product of q1 and q2

  `dot` `(detail::tquat< T > const &q1, detail::tquat< T > const &q2)`

- Returns the length of the quaternion q

  `length` `(detail::tquat< T > const &q)`

- Returns the normalized quaternion of q

  `normalize` `(detail::tquat< T > const &q)`

http://glm.g-truc.net/0.9.0/api/a00135.html

glm::quat

`typedef detail::tquat< float >  quat`

Quaternion of floating-point numbers.

Functions:

- Returns a SLERP interpolated quaternion of q1 and q2 according t.

```
detail::tquat< T >
mix ( detail::tquat< T > const &q1,
      detail::tquat< T > const &q2,
      typename detail::tquat< T >::value_type const &t)
```

all the other functions also exits, pow, log, squad, ...

http://glm.g-truc.net/0.9.0/api/a00135.html

glm::gtx::quaternion::angleAxis

Build a quaternion from an angle and an axis.

```
detail::tquat< valType >
angleAxis (valType const &angle,
           valType const &x, valType const &y, valType const &z)


detail::tquat< valType >
angleAxis (valType const &angle, detail::tvec3< valType > const &v)
```

http://glm.g-truc.net/0.9.0/api/a00184.html

ARLAB

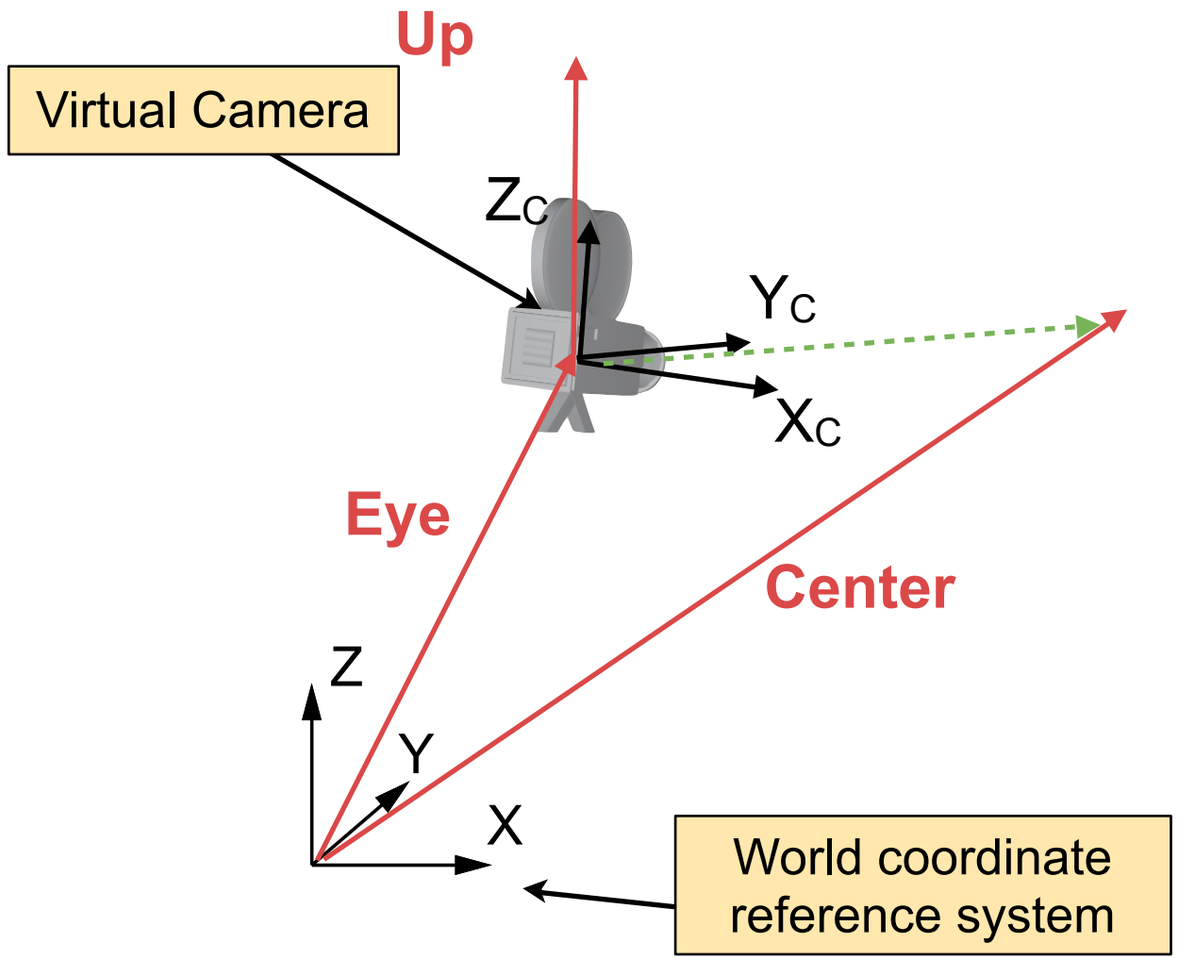
Gimbal-lock free navigation

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

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)
```

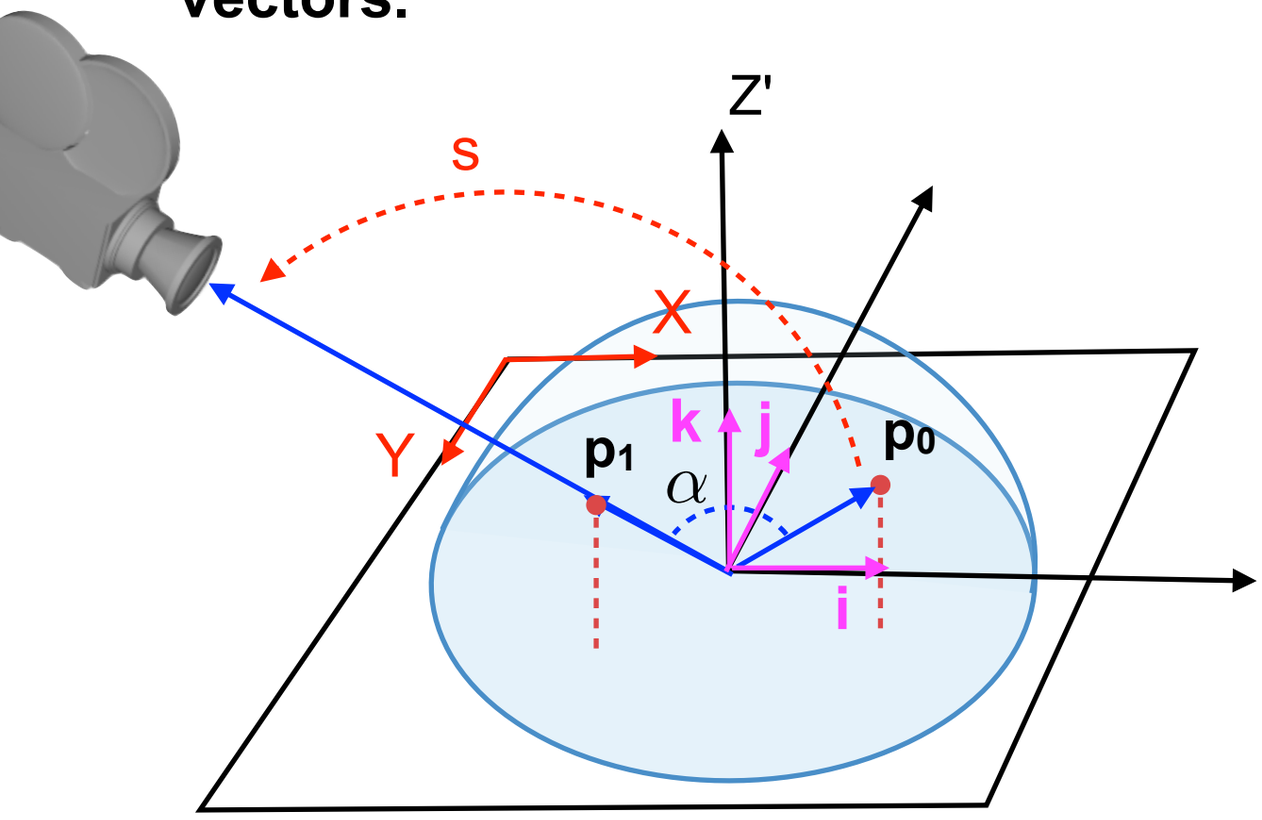
Parameters to describe a camera position and orientation:

- 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.

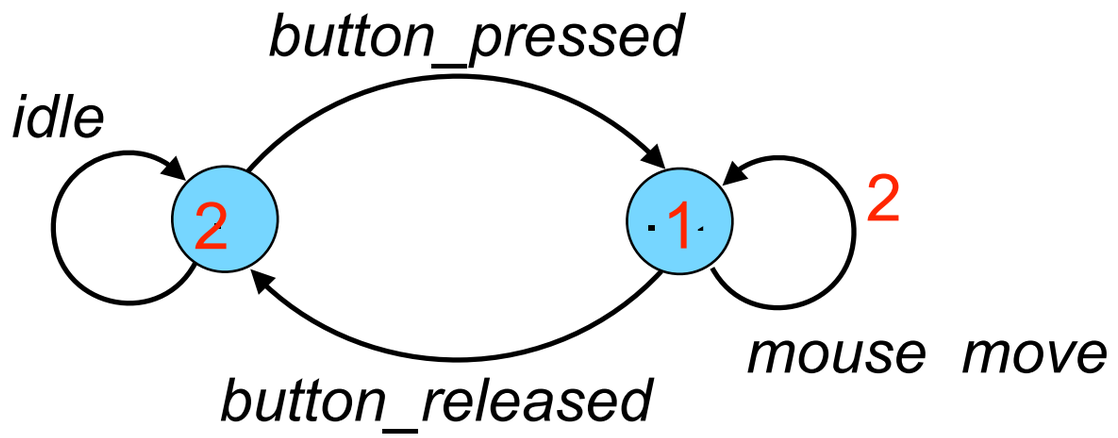
Trackball Manipulator

ARLAB

Calculate the angle between both vectors.



State Machine



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

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

- Map the angle to the complex vectors i, j, k

Code Changes

```
position = position - target;

angle_x *= dt;
angle_y *= dt;

glm::quat old_orient = orient;
glm::normalize(orient);

glm::quat q_x_axis =
        glm::gtx::quaternion::angleAxis(float(angle_y),1.0f, 0.0f, 0.0f) ;
glm::quat q_y_axis =
        glm::gtx::quaternion::angleAxis(float(angle_x), 0.0f, 1.0f, 0.0f);

orient = orient * q_x_axis * q_y_axis;

upDirection = upDirection * orient;
position = position * (orient);


orient = old_orient;

position = position + target;
```

This is our current orientation at p_0

Mapping our new orientation to x and y

No gimbal lock problem

Code Changes

```
position = position - target;

angle_x *= dt;
angle_y *= dt;

glm::quat old_orient = orient;
glm::normalize(orient);

glm::quat q_x_axis =
        glm::gtx::quaternion::angleAxis(float(angle_y),1.0f, 0.0f, 0.0f) ;
glm::quat q_y_axis =
        glm::gtx::quaternion::angleAxis(float(angle_x), 0.0f, 1.0f, 0.0f);

orient = orient * q_x_axis * q_y_axis;

upDirection = upDirection * orient;
position = position * (orient);


orient = old_orient;

position = position + target;
```

No gimbal lock problem

$$R_{old} = R_{old} \cdot R_{new}$$

It comes from the right... what does this mean?

Code Changes

```
position = position - target;

angle_x *= dt;
angle_y *= dt;

glm::quat old_orient = orient;
glm::normalize(orient);

glm::quat q_x_axis =
        glm::gtx::quaternion::angleAxis(float(angle_y),1.0f, 0.0f, 0.0f) ;
glm::quat q_y_axis =
        glm::gtx::quaternion::angleAxis(float(angle_x), 0.0f, 1.0f, 0.0f);

orient = orient * q_x_axis * q_y_axis;

upDirection = upDirection * orient;
position = position * (orient);

orient = old_orient;

position = position + target;
```

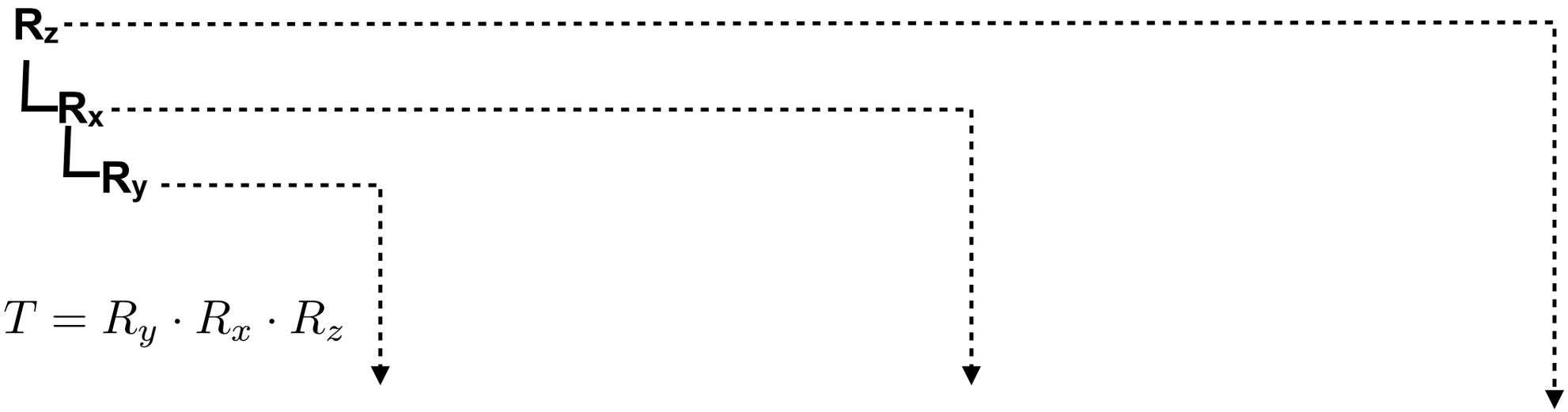
No gimbal lock problem

Calculate a new position and a new up vector

We calculate the position frame by frame, between to mouse clicks, thus, remember the origin at click 1.

In Comparison, slide from the beginning

The sequence of transformations (rotations in our case) can be considered as hierarchy of transformations.



$$T = R_y \cdot R_x \cdot R_z$$

$$T = \begin{bmatrix} cos(\beta) & 0 & -sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ cos(\alpha) & -sin(\alpha) & 0 & 0 \\ sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos(\gamma) & sin(\gamma) & 0 & 0 \\ -sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Navigation is represented as a sequence of multiplication over time.

$$T = T_1\ T_2\ T_4\ \ldots\ T_n$$

Gimbal lock problem

ARLAB

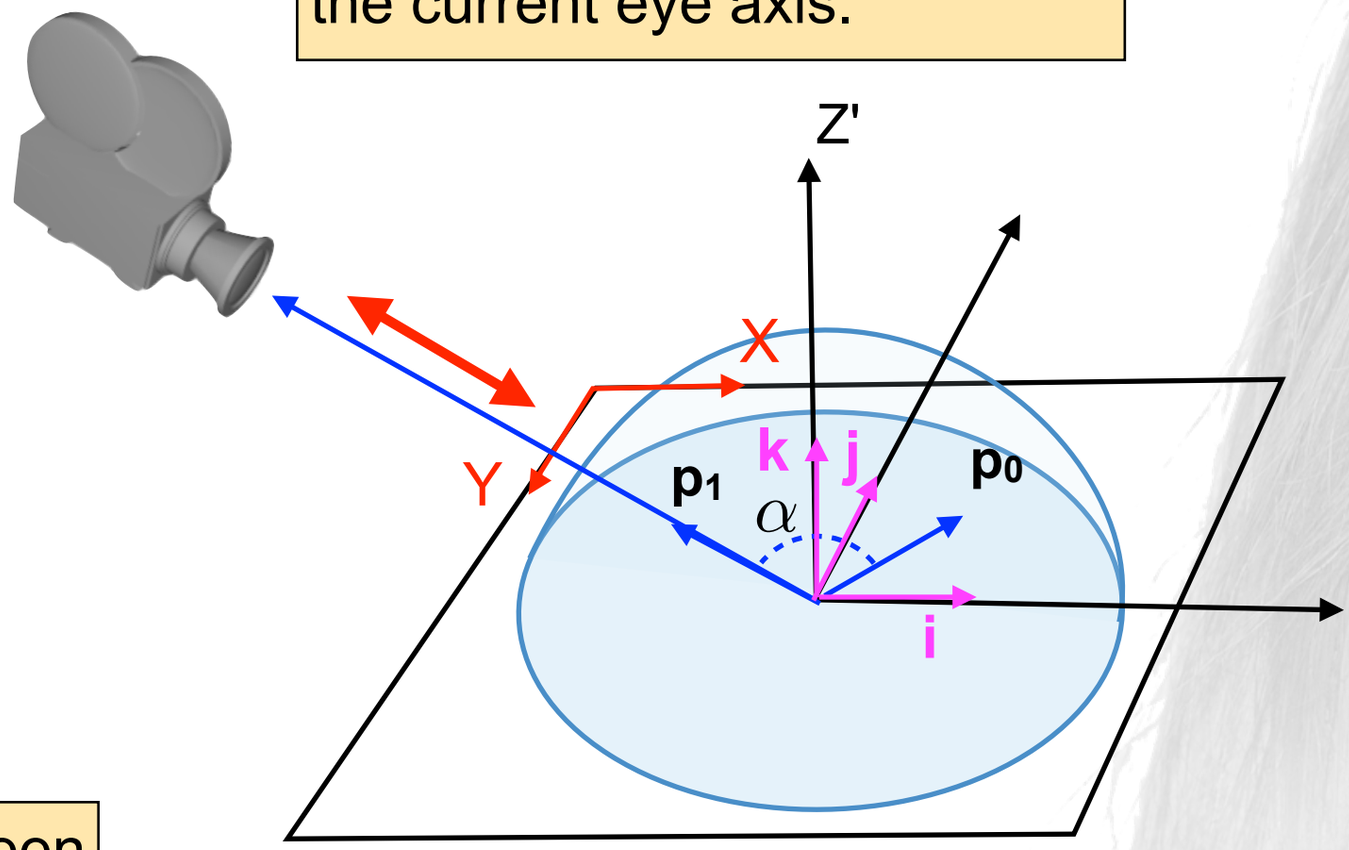
Zoom with a Trackball

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Trackball

Zoom = moving forward on the current eye axis.



Representation as matrix:

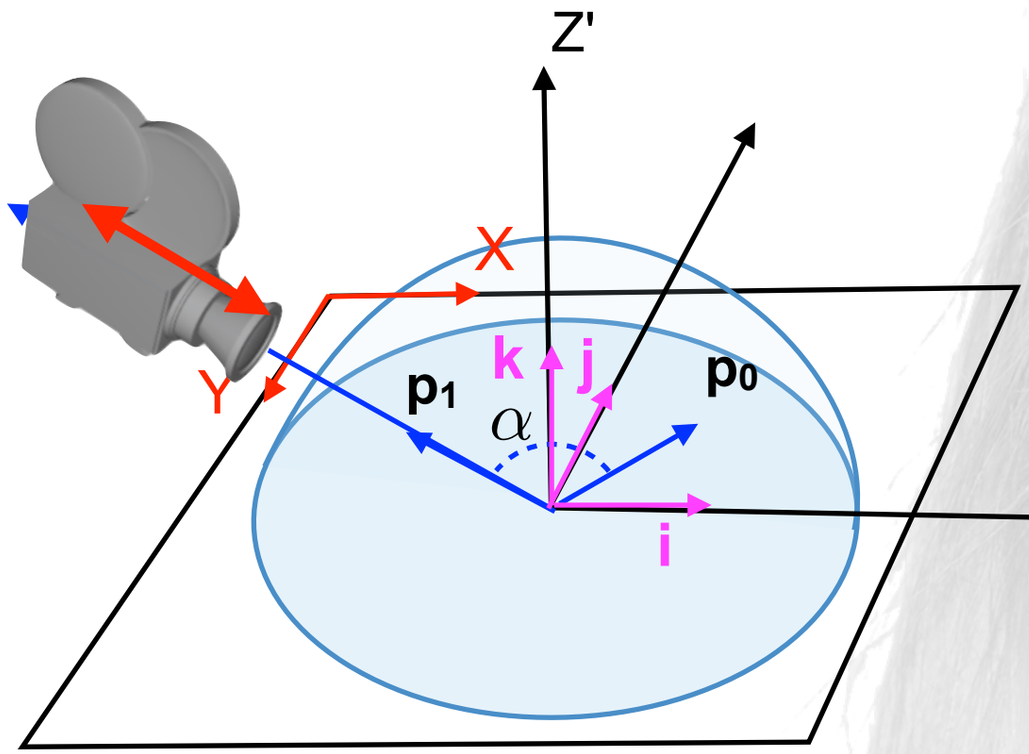
$$R = VM \cdot R_{\text{trackball}}$$

screen

We observe the right mouse button click and one axis.

Trackball

Zoom = moving forward on the current eye axis.



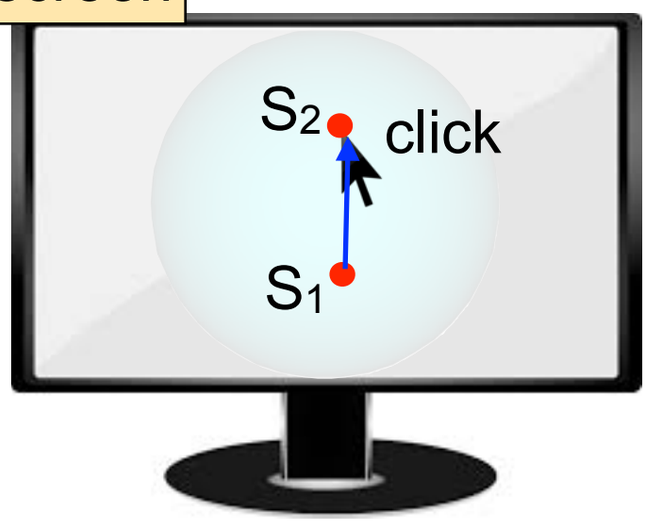
Representation as matrix:

$$R = VM \cdot R_{\text{trackball}}$$

VM: view matrix

Rotation from the trackball

screen

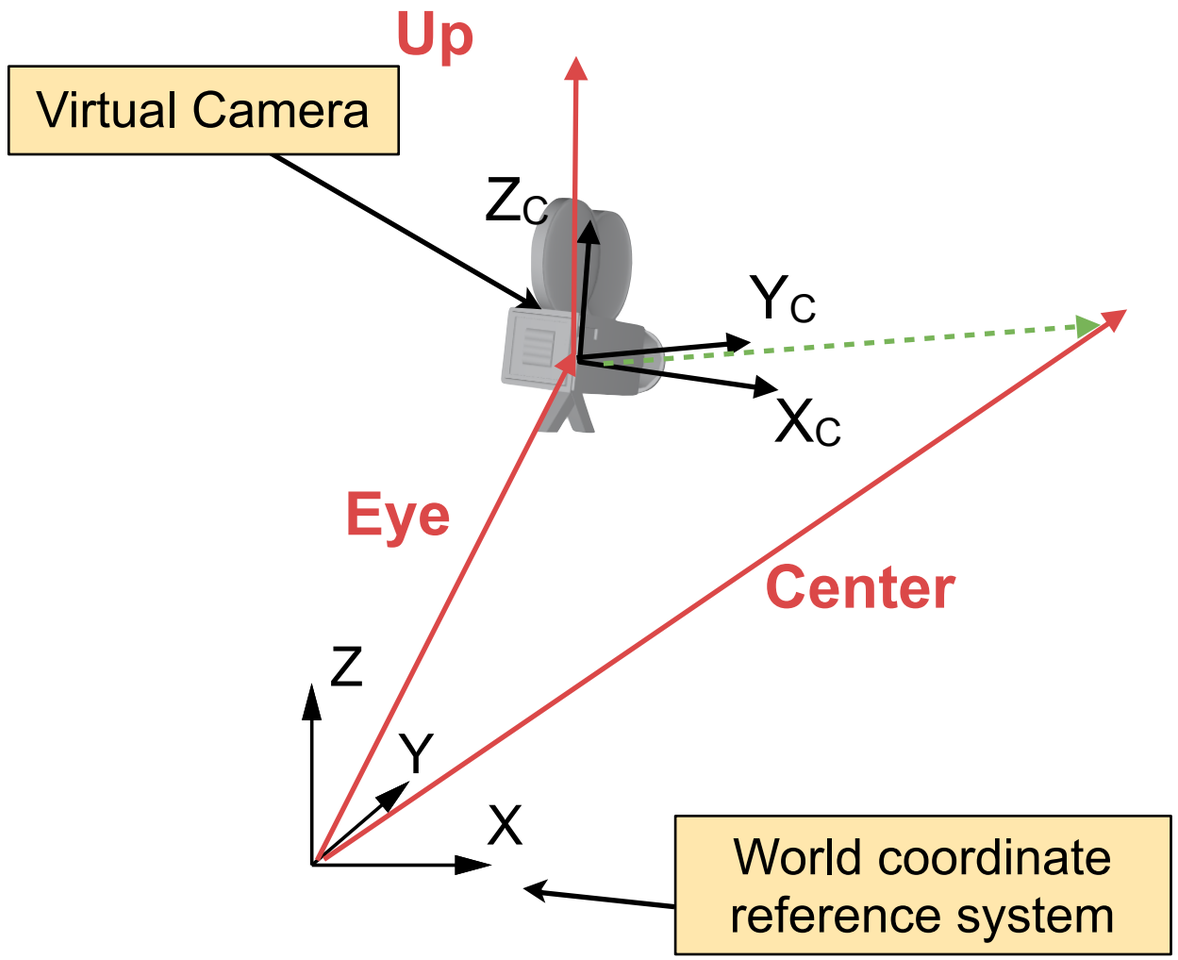


We observe the right mouse button click and one axis.

and use the distance between both points to move the camera.

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)
```

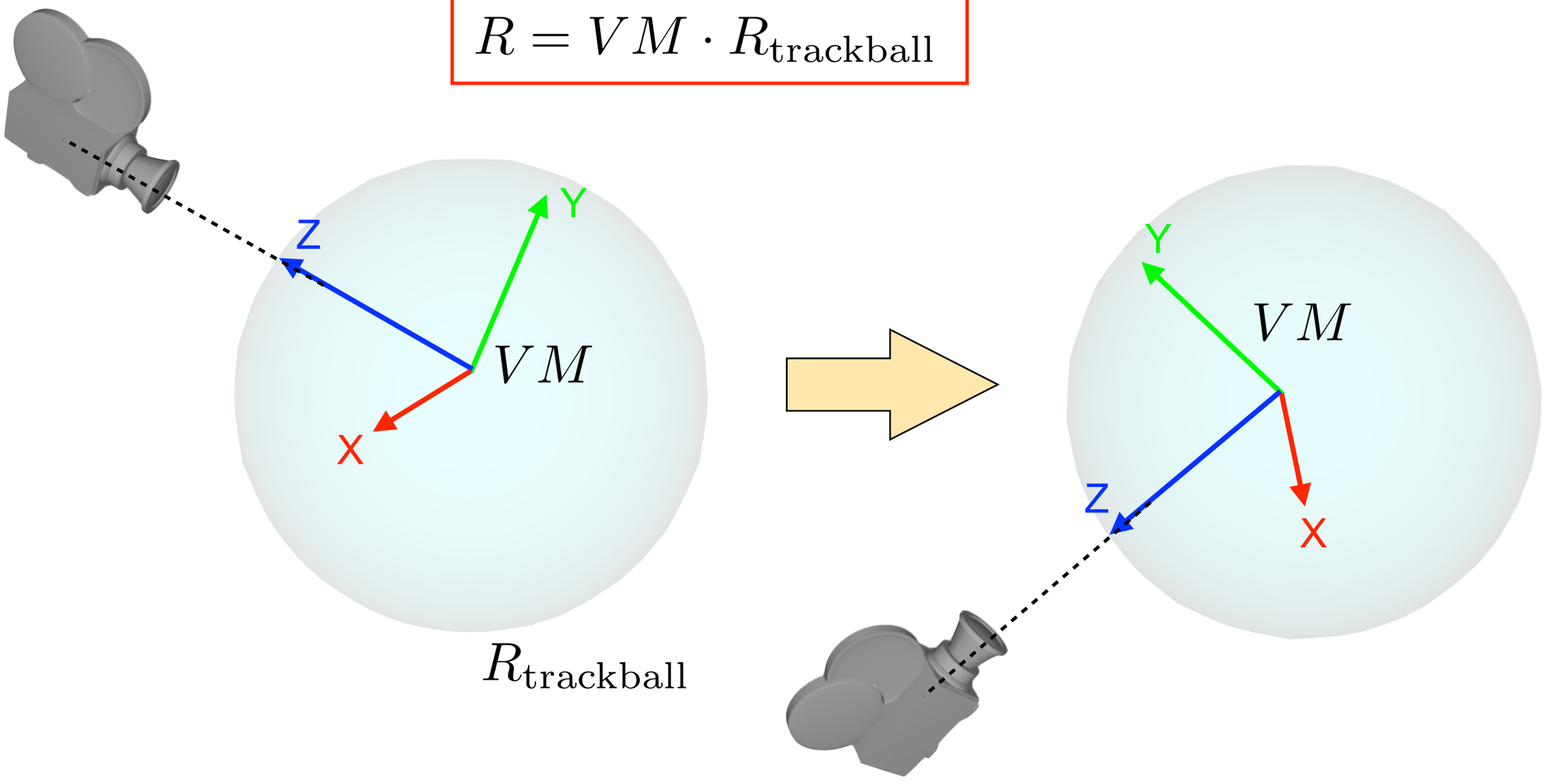
Parameters to describe a camera position and orientation:

- 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.

Trackball

Representation as matrix:

$$R = VM \cdot R_{\text{trackball}}$$



We always rotate our view matrix and move our camera by changing the eye value.

Example Code

ARLAB

Trackball parameter in *CameraManipulator*

```
glm::vec3 translation(0.0f, 0.0f, distance),
```

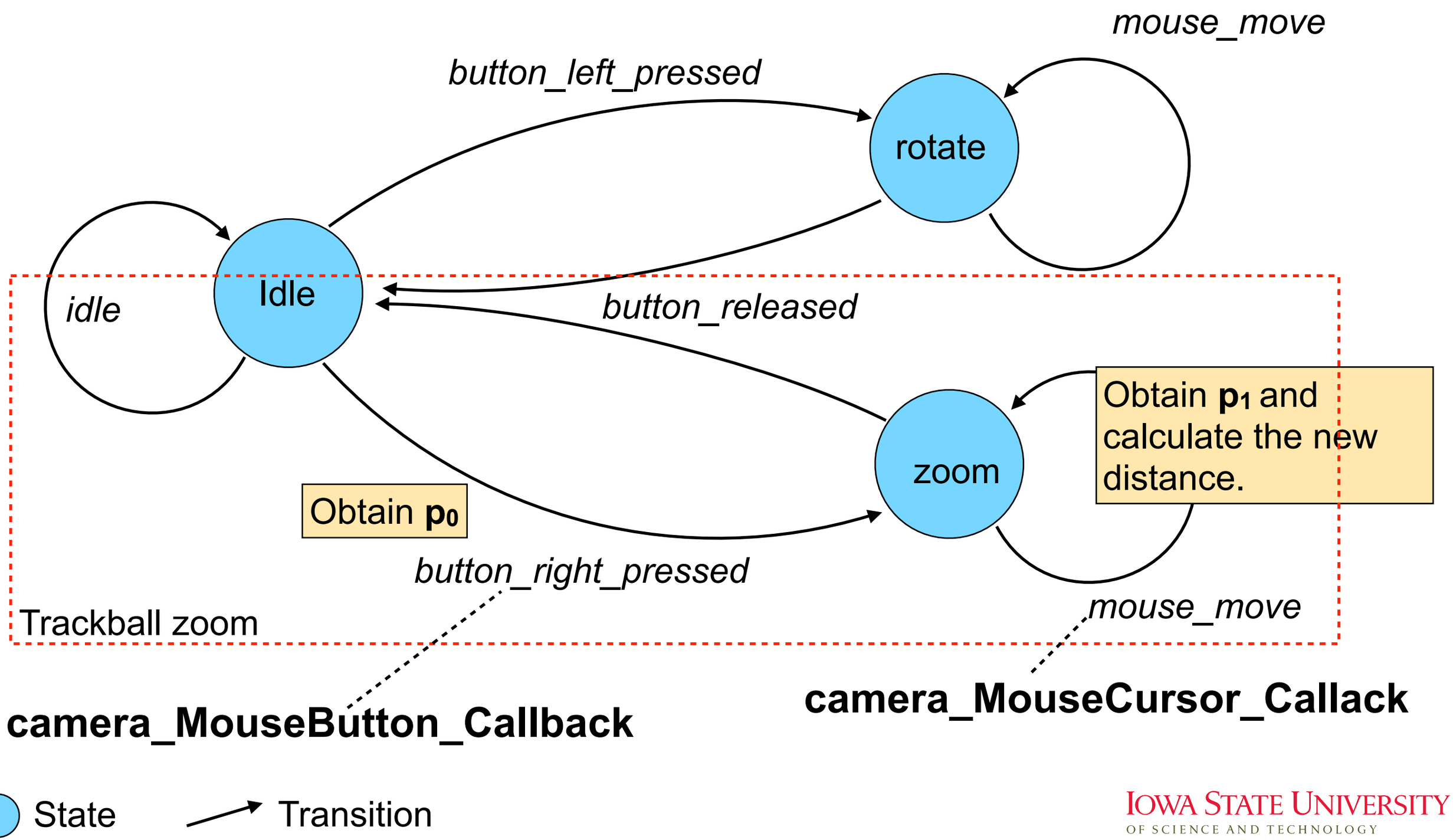
Function in GLObject.cpp

```
/*!
 Global function to set the trackball data
 */
void SetTrackballLocation(glm::mat4 orientation, glm::vec3 translation)
{
    g_viewMatrix = glm::lookAt(translation,
                               glm::vec3(0.0f, 0.0f, 0.0f),
                               glm::vec3(0.0f, 1.0f, 0.0f));

    g_trackball = orientation;
    g_rotated_view = g_viewMatrix * g_trackball;
    g_inv_rotated_view = glm::inverse(g_rotated_view);
}
```

Are passed into our shader program.

State Machines

The typical processes for an interaction process:



CameraManipulator

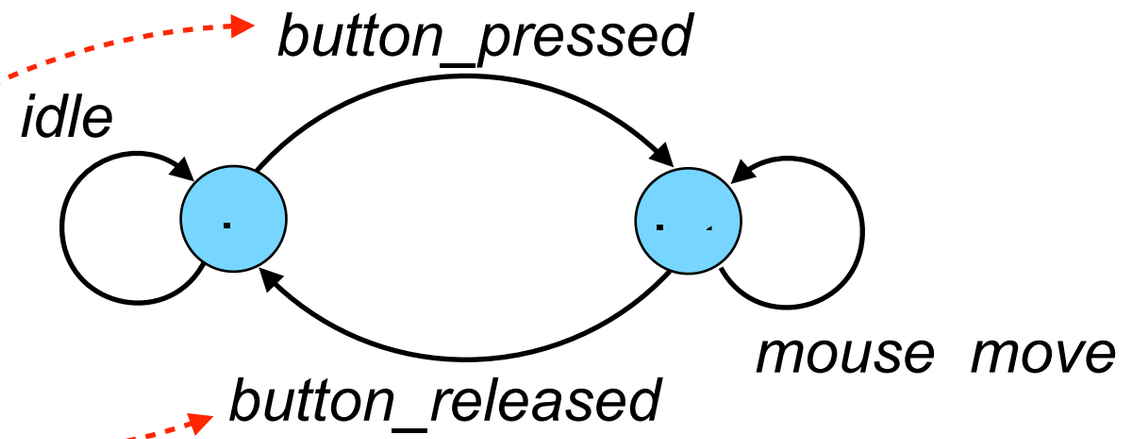
ARLAB

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 and the position in the previous frame.

```
glm::vec3 _lastframePosDistance
```

Current position when moving the mouse

```
glm::vec3 _currentPosDistance
```

The distance to the camera

```
float _camera_distance
```

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

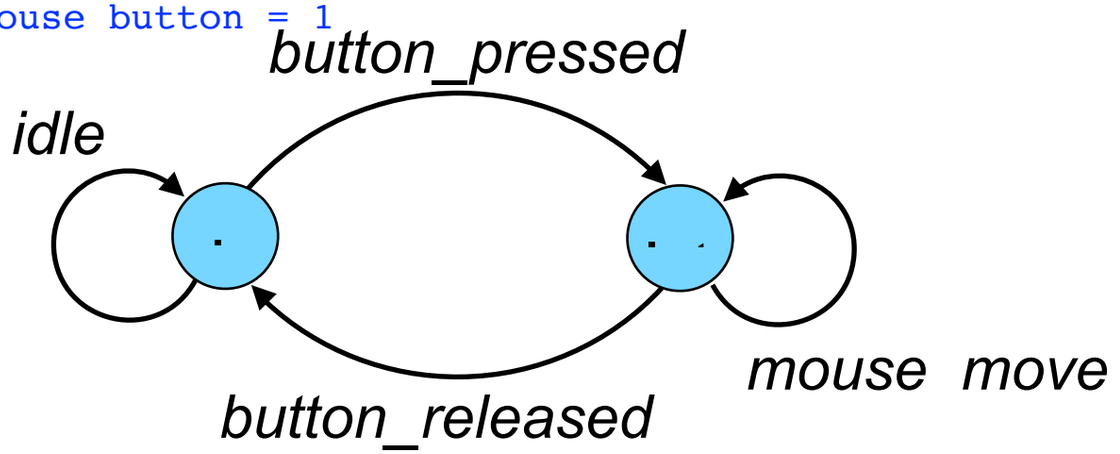
Left button

Right button

Button released

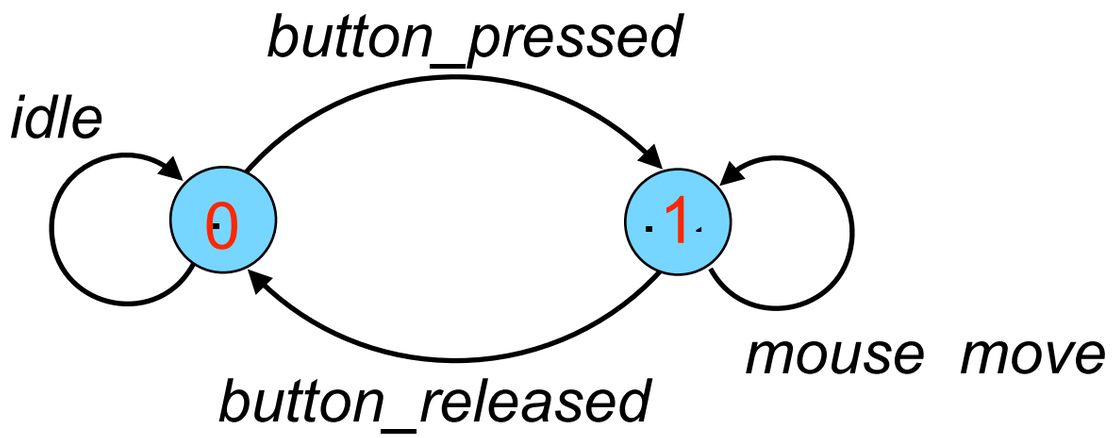
Left button

Right button



camera_MouseButton_Callback

```
switch (action) {
        case GLFW_PRESS: // 1
            [...]
            else if(button == GLFW_MOUSE_BUTTON_2) // Right mouse button = 1
            {
                // switch mouse button event to 1 when the mouse button is pressed
                _rightMouseButtonEvent = 1;

            }
            break;
        case GLFW_RELEASE: // 0

            [...]
            else if(button == GLFW_MOUSE_BUTTON_2) // Right mouse button = 0
            {
                // switch mouse button event to 0 when the mouse button is release
                _rightMouseButtonEvent = 0;

            }

            break;

}
```



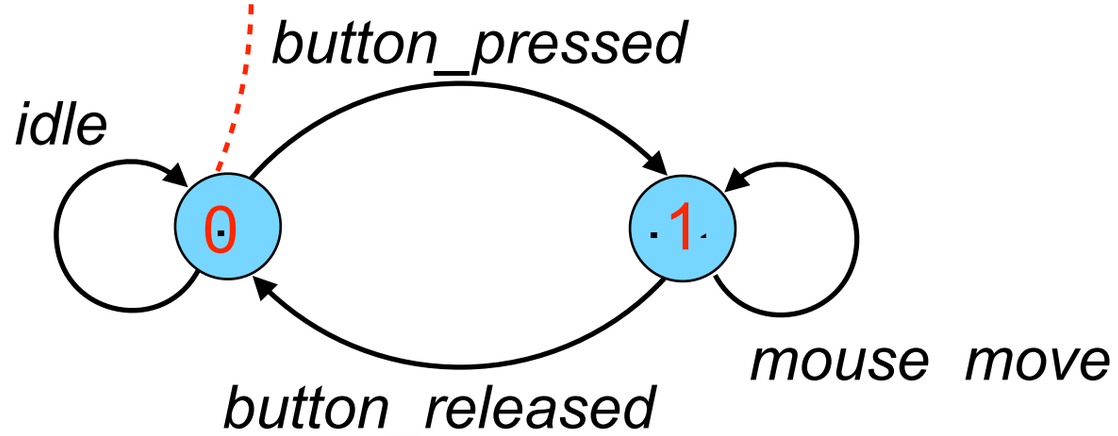
camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
      ///-----------------------------
      // Camera distance starts here
      switch (_rightMouseButtonEvent)
      {
       case 0:
            break;

     case 1:
          //  remember the first position
          _rightMouseButtonEvent = 2;
          _lastframePosDistance = toWindowCoord( x, y );
          return;
          break;
}
```

This function gets called every frame!

State 0: we do nothing and return / idle

continue here

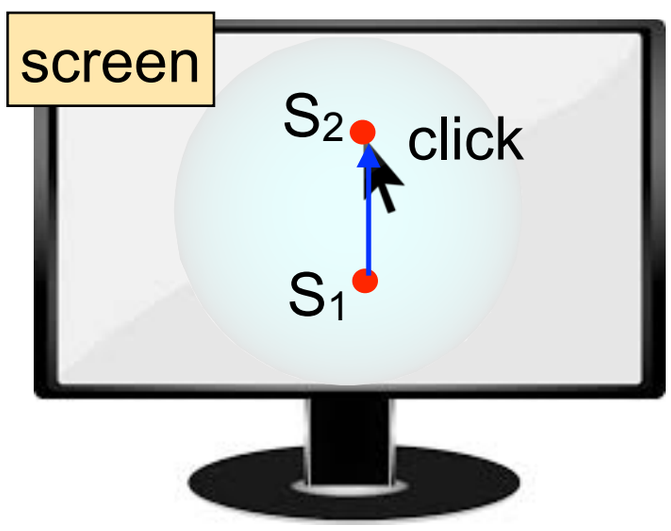


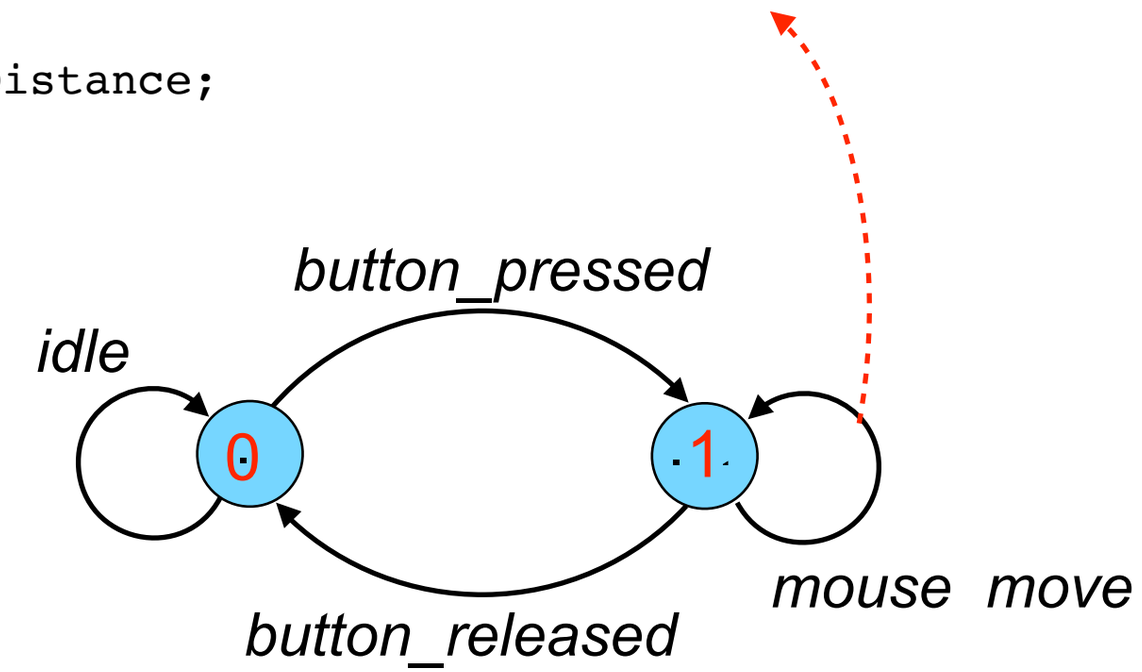
camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
   if(_rightMouseButtonEvent == 2)
   {
        // Tracking the current mouse cursor
        _currentPosDistance  = toWindowCoord( x, y )
        float dir =  _lastframePosDistance.y - _currentPosDistance.y;
              // this order to invert the mouse motion

        float length = sqrt( (_currentPosDistance.y - _lastframePosDistance.y)*
                             (_currentPosDistance.y - _lastframePosDistance.y));

        _camera_distance = _camera_distance + length * dir * _traveling_speed;
        _camera_distance =  glm::clamp(_camera_distance, 0.0f, 600.0f);

        _lastframePosDistance = _currentPosDistance;
   }
```

This function gets called every frame!

up or down = zoom in / zoom out



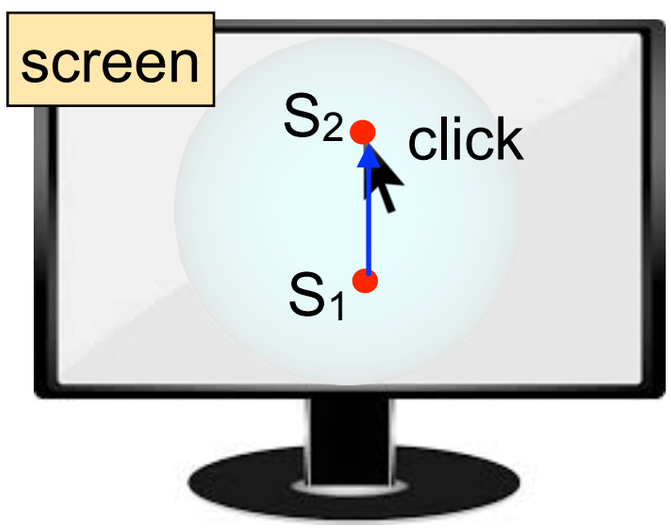


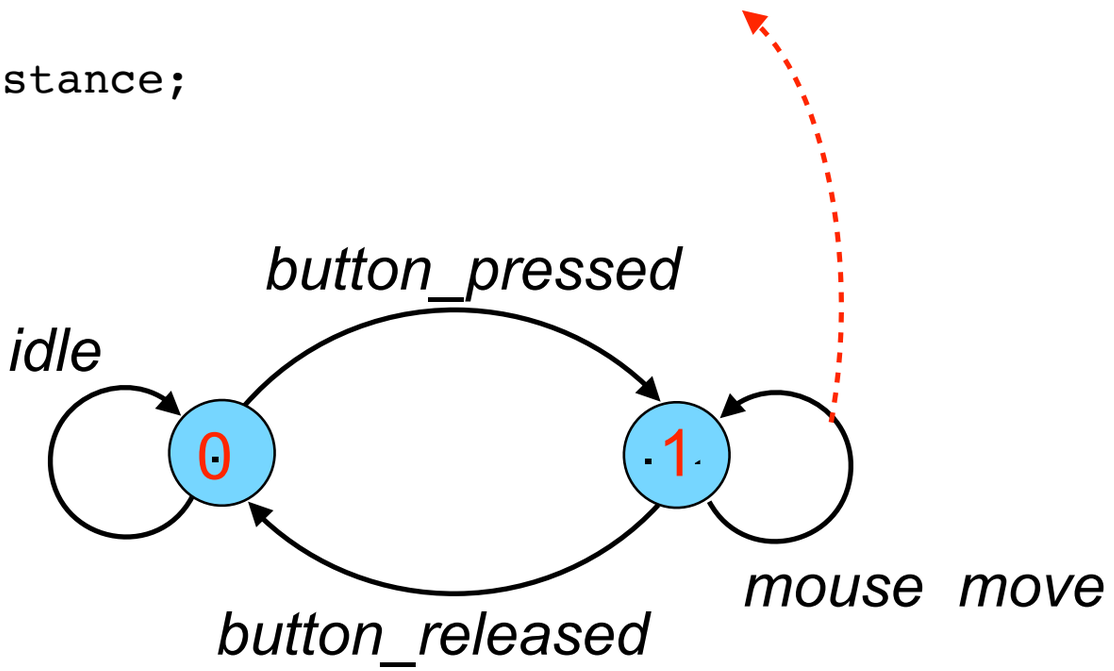
camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
   if(_rightMouseButtonEvent == 2)
   {
        // Tracking the current mouse cursor
        _currentPosDistance  = toWindowCoord( x, y )
        float dir =  _lastframePosDistance.y
             // this order to invert the mou

      float length = sqrt( (_currentPosDistance.y - _lastframePosDistance.y)*
                           (_currentPosDistance.y - _lastframePosDistance.y));

        _camera_distance = _camera_distance + length * dir * _traveling_speed;
        _camera_distance =  glm::clamp(_camera_distance, 0.0f, 600.0f);

        _lastframePosDistance = _currentPosDistance;
   }
```

This function gets called every frame!

Distance in screen coordinates.
Note, it runs from 0 to 1



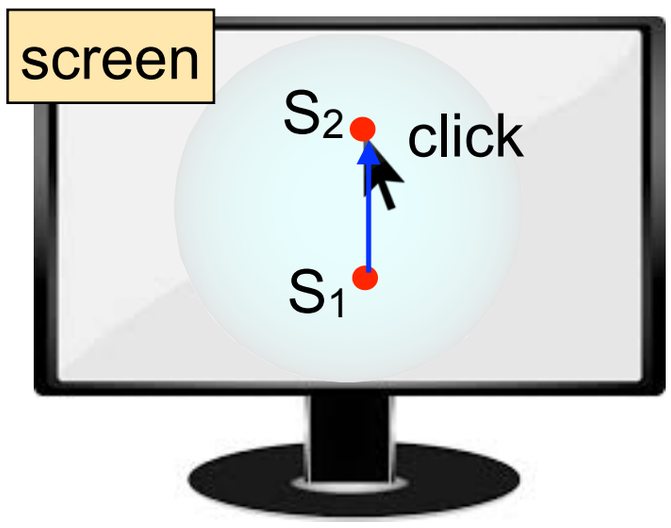


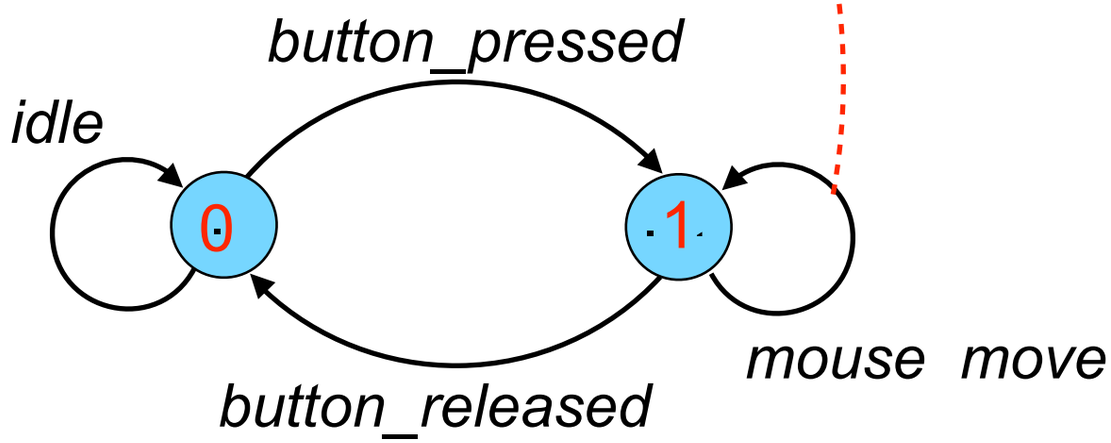
camera_MouseCursor_Callack

```
void CameraManipulator::camera_MouseCursor_Callack( GLFWwindow *window, double x, double y)
{
   [...]
   if(_rightMouseButtonEvent == 2)
   {
        // Tracking the current mouse cursor
        _currentPosDistance  = toWindowCoord( x, y )
        float dir =  _lastframePosDistance.y - _currentPosDistance.y;
             // this order to invert the mouse motion

       float length = sqrt( (_currentPosDistance.y - _lastframePosDistance.y)*
                            (_current                                    sDistance.y));
        _camera_distance = _camera_distance + length * dir * _traveling_speed;
        _camera_distance =  glm::clamp(_camera_distance, 0.0f, 600.0f);

        _lastframePosDistance = _currentPosDistance;
   }
```

This function gets called every frame!

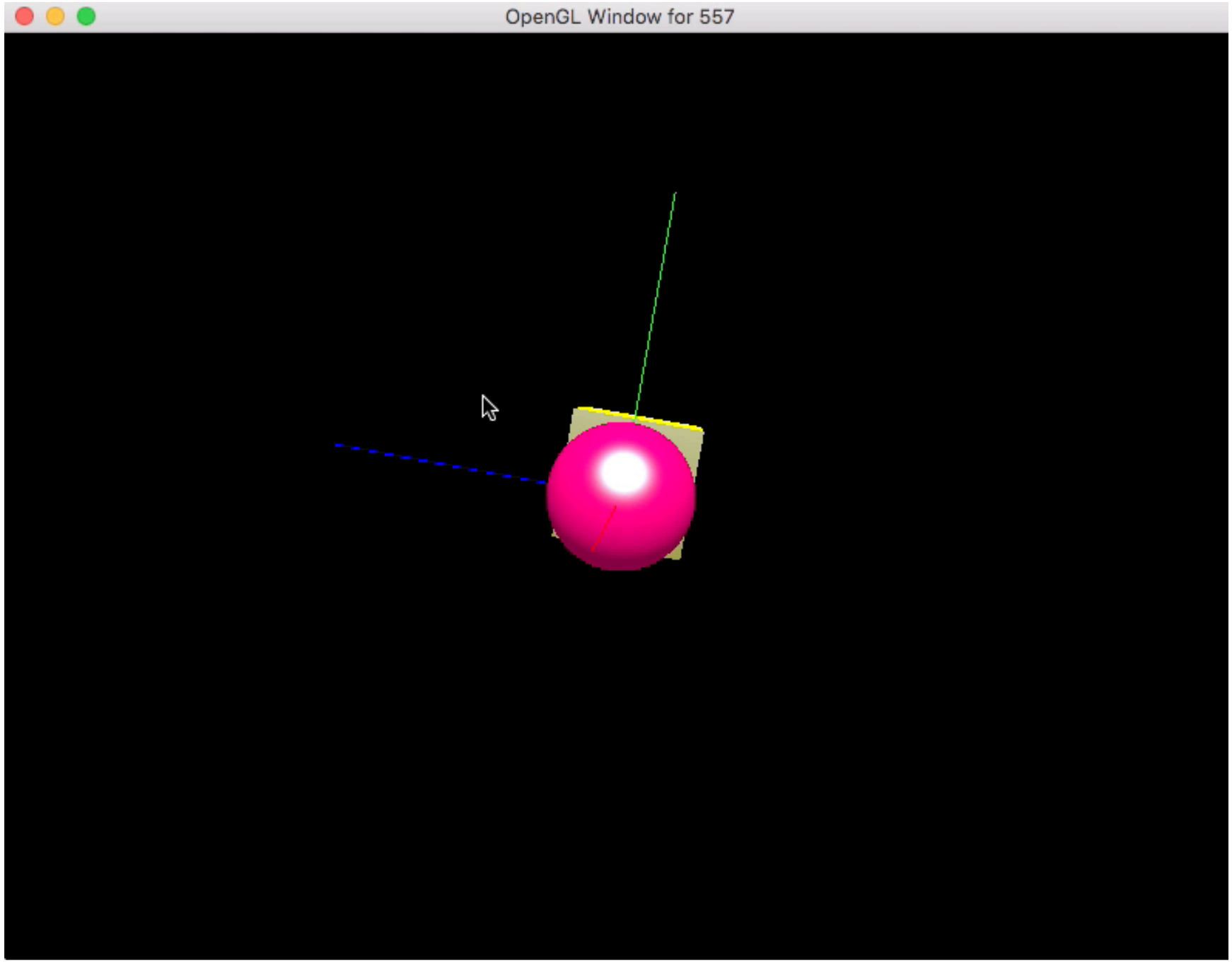
calculate the new distance.





Video

ARLAB



Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 50011, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu
http://arlab.me.iastate.edu

www.linkedin.com/in/rradkowski

ARLAB

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY