ME/CprE/ComS 557

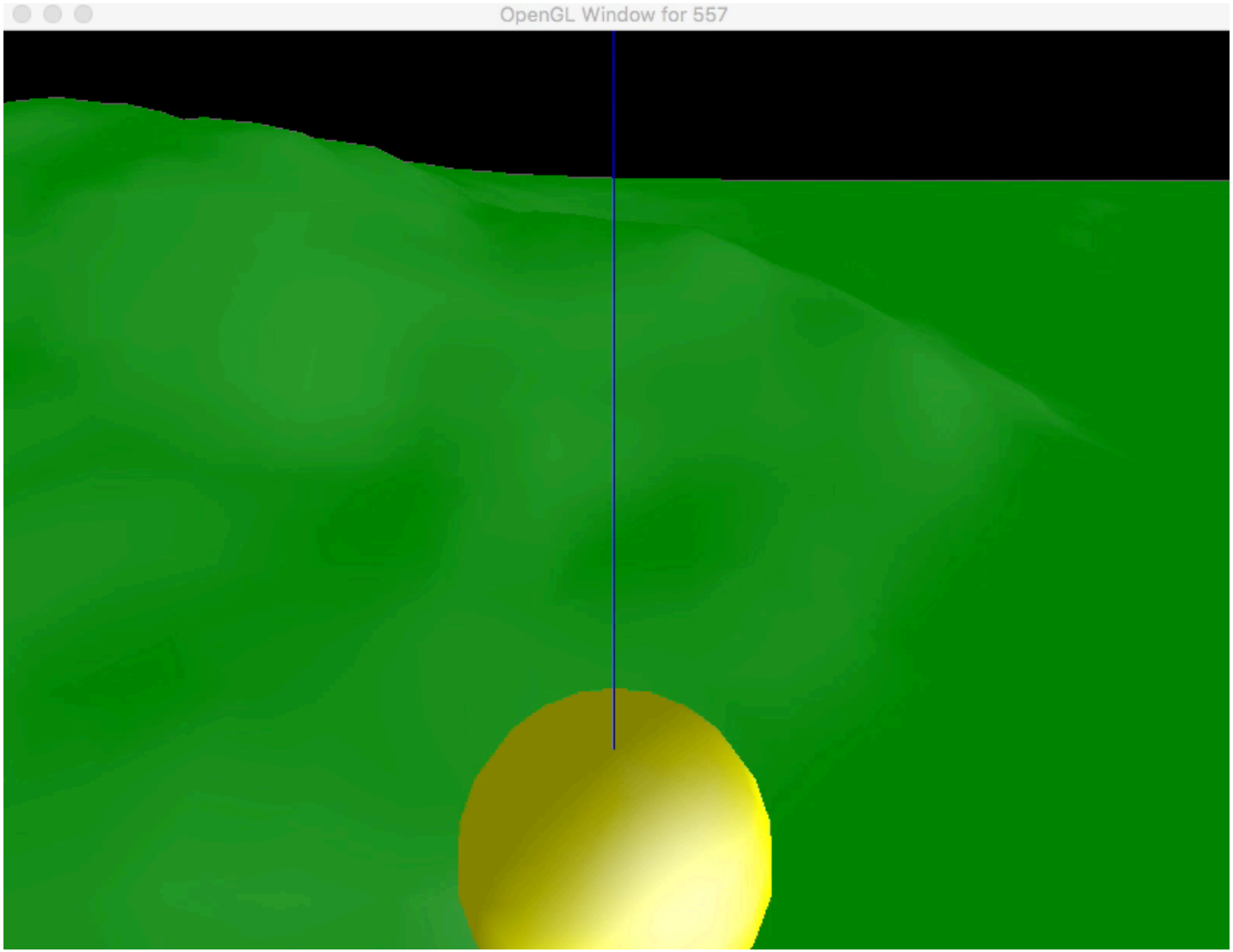
Computer Graphics and Geometric Modeling

Locomotion on a Surface

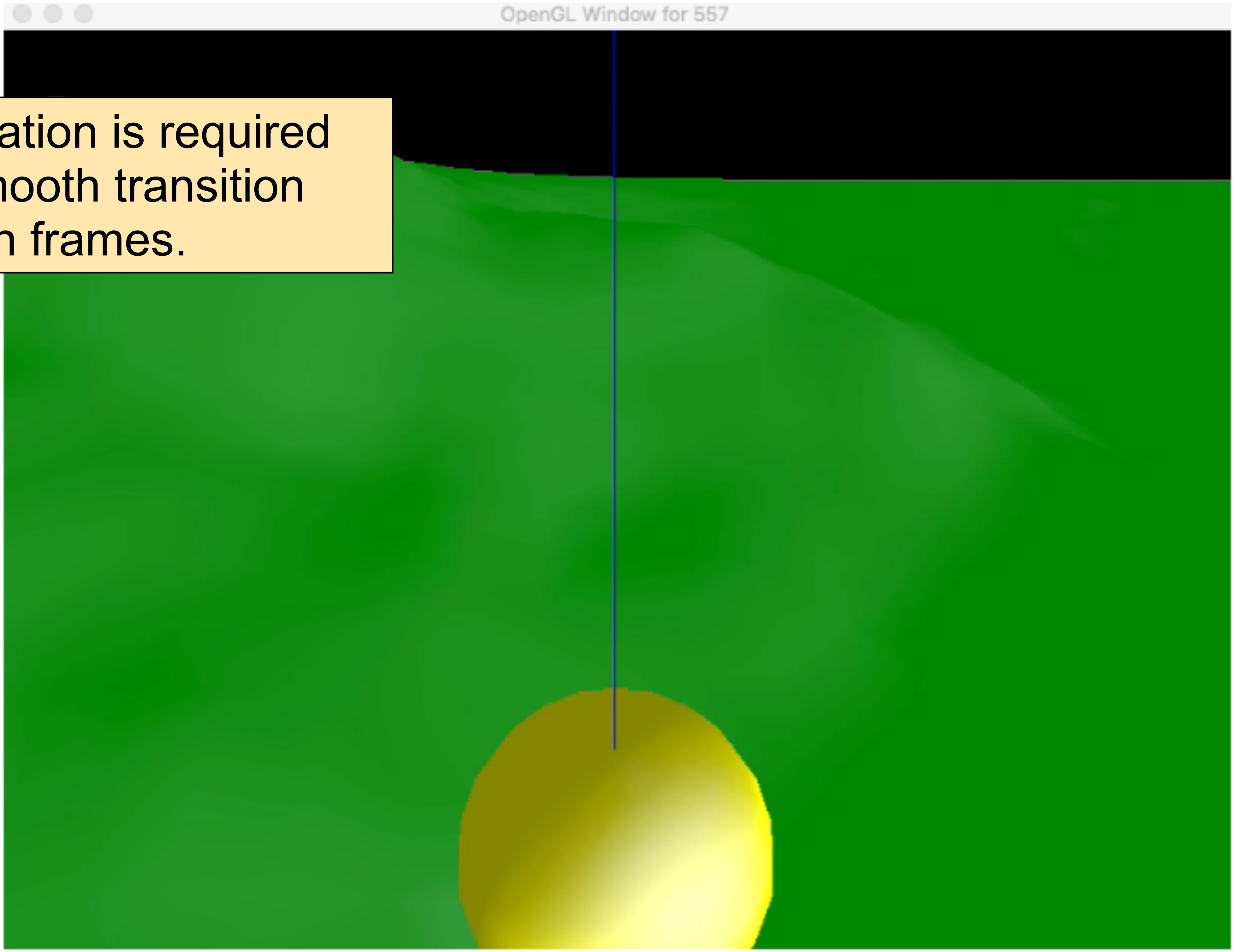
November 10th, 2015
Rafael Radkowski

Iowa State University
of Science and Technology

Video

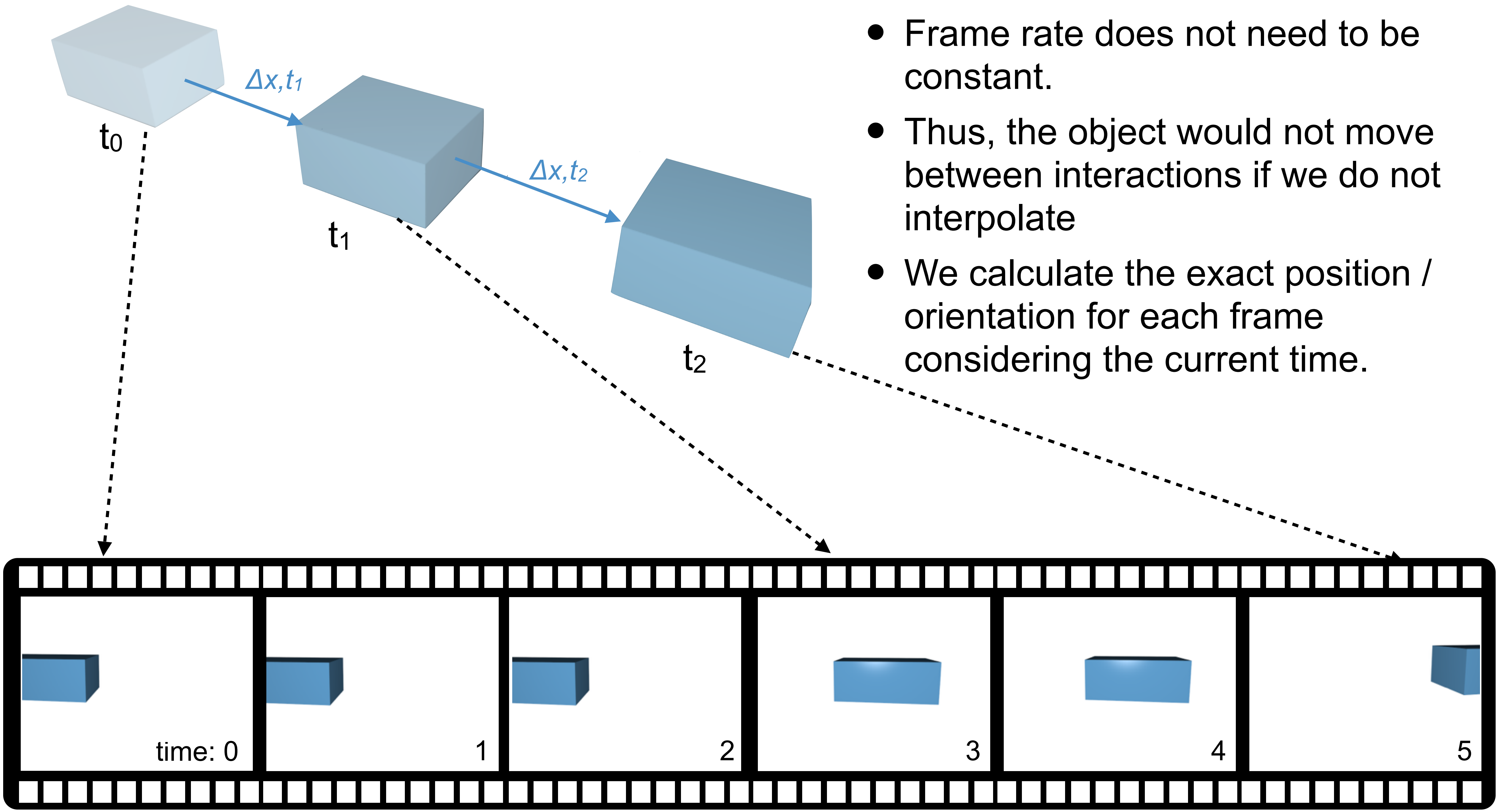


Video



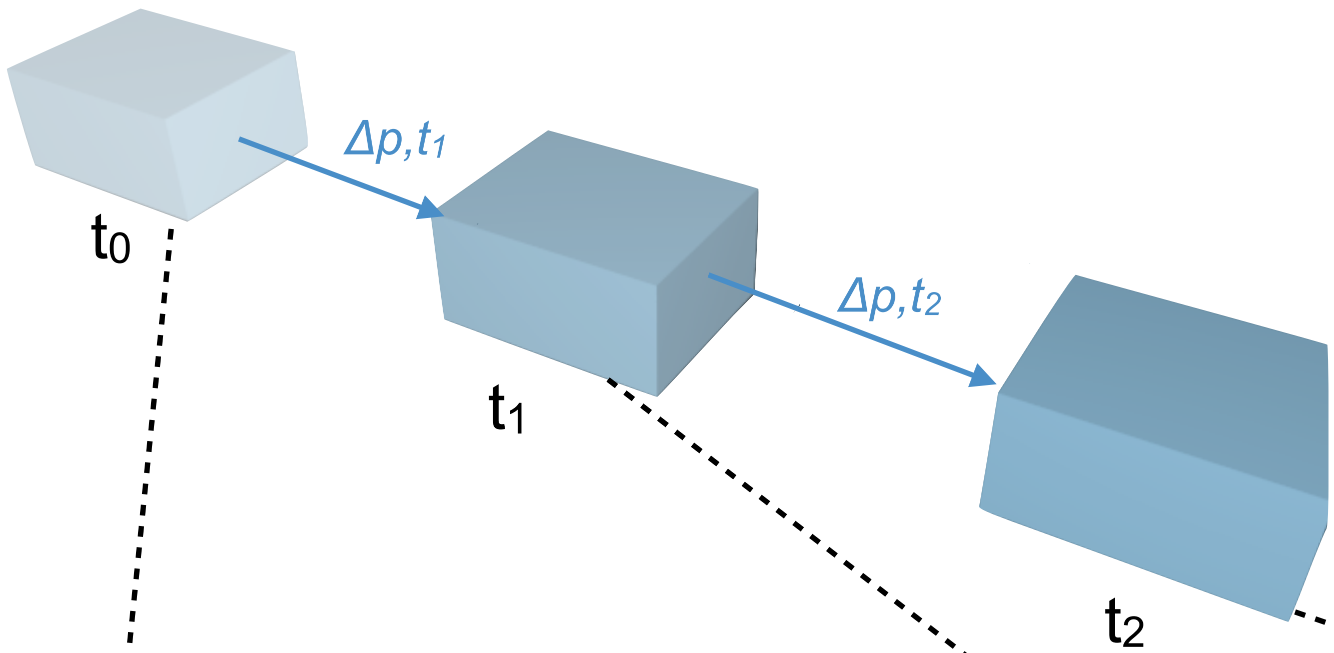
Interpolation is required for a smooth transition between frames.

NO Interpolation between Points

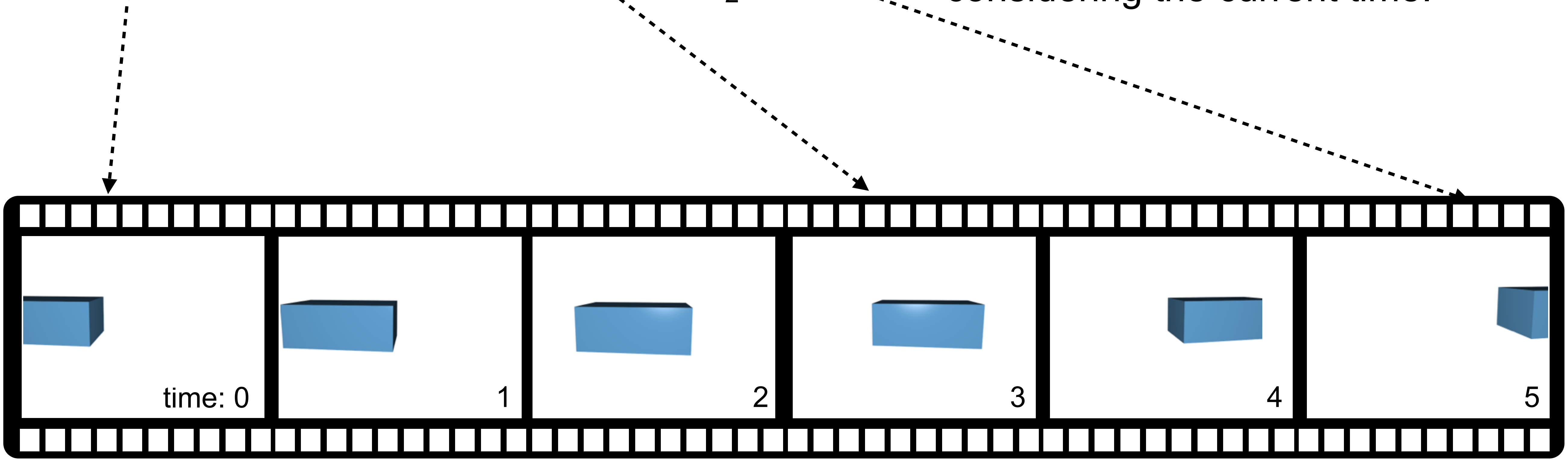


- Frame rate does not need to be constant.
- Thus, the object would not move between interactions if we do not interpolate
- We calculate the exact position / orientation for each frame considering the current time.

Interpolation between Points

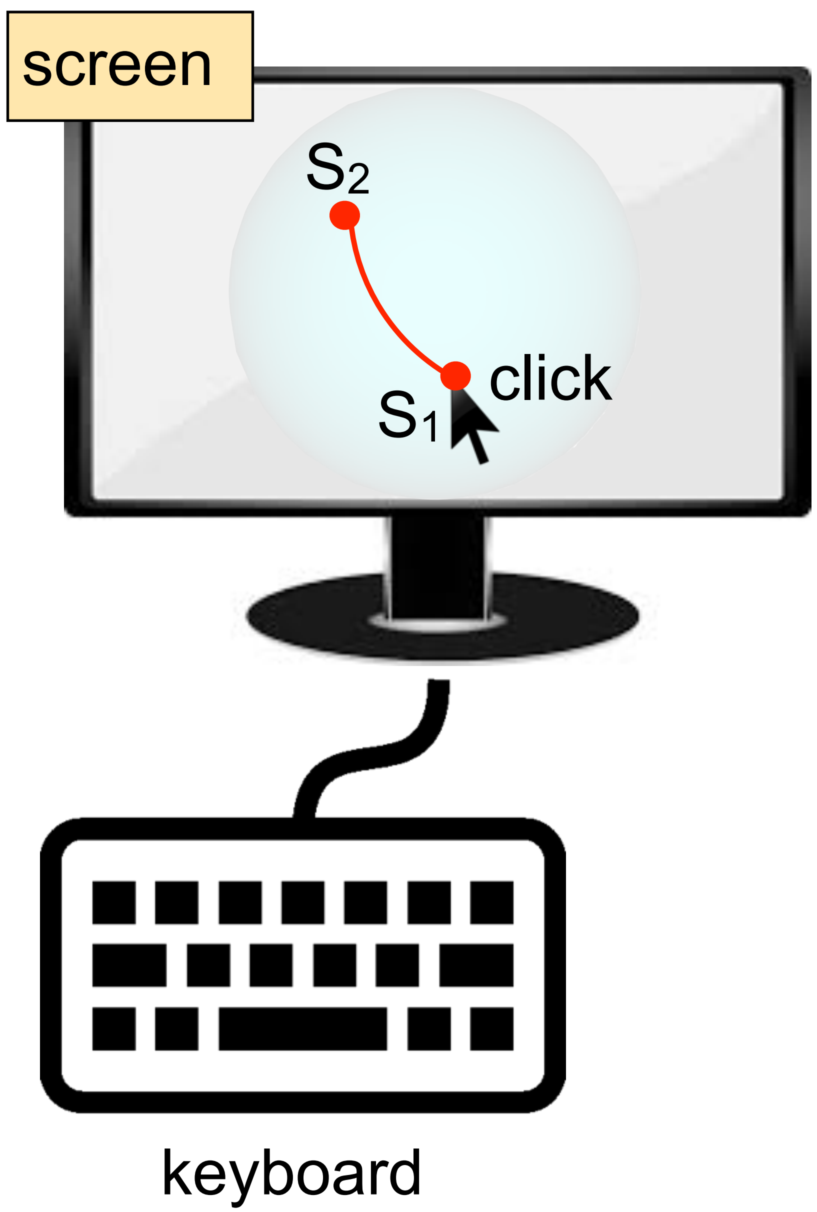


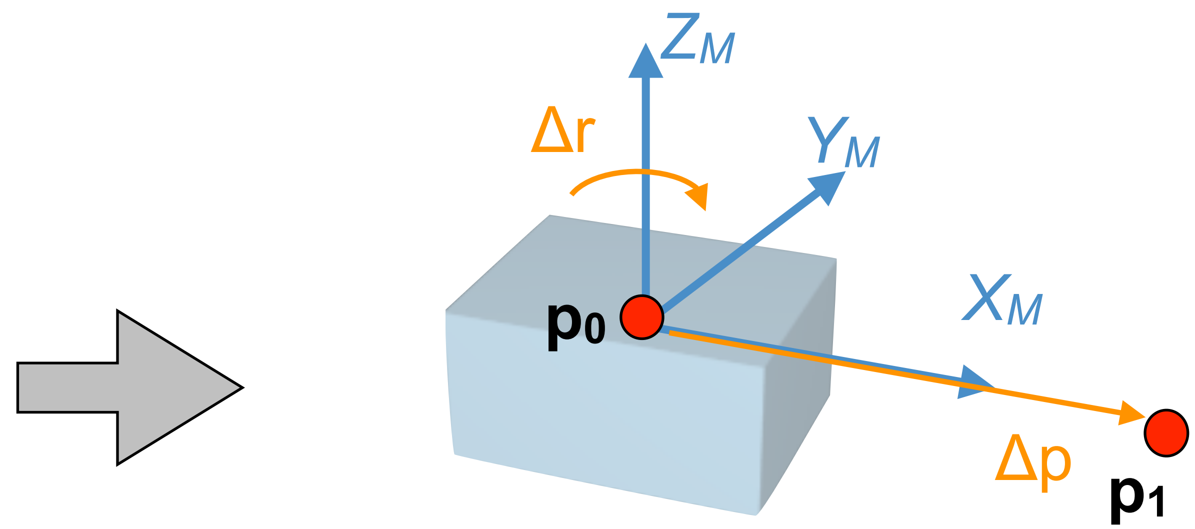
- Frame rate does not need to be constant.
- Thus, the object would not move between interactions if we do not interpolate
- We calculate the exact position / orientation for each frame considering the current time.



Relation input and action

ARLAB

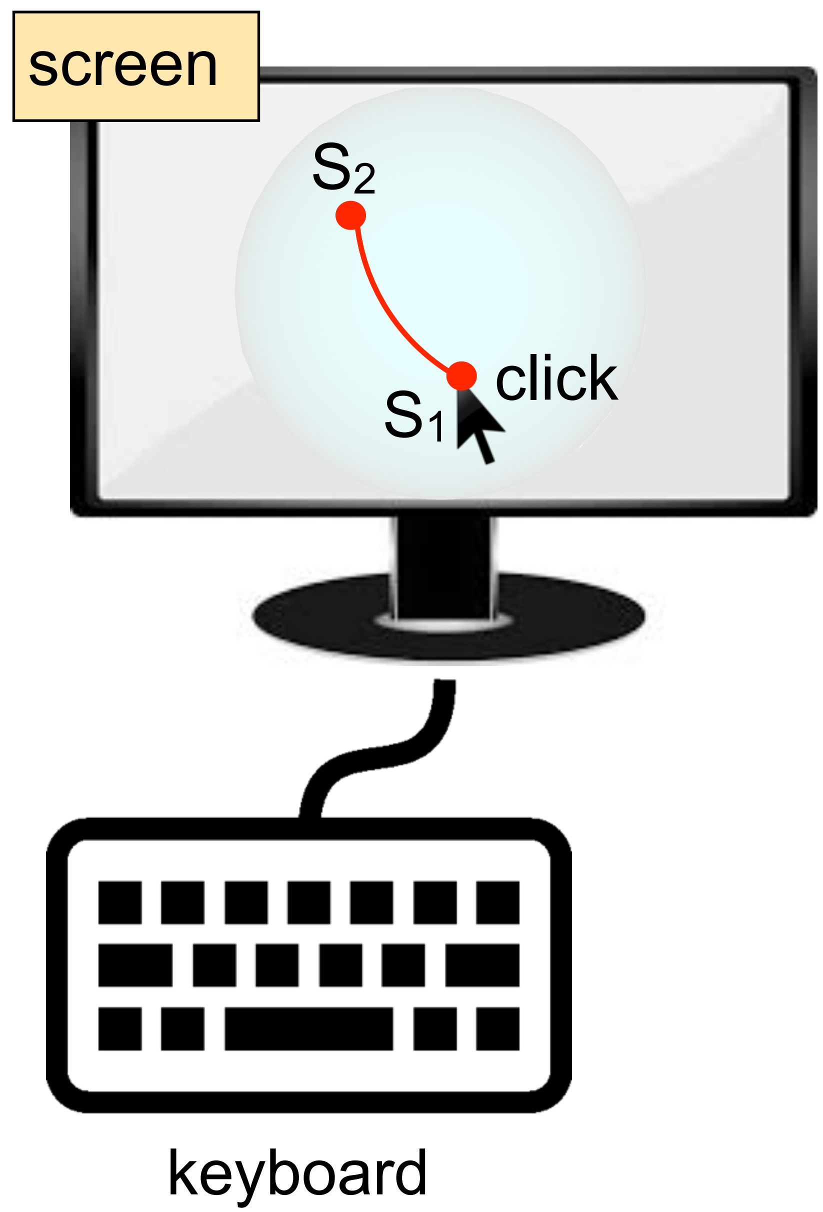


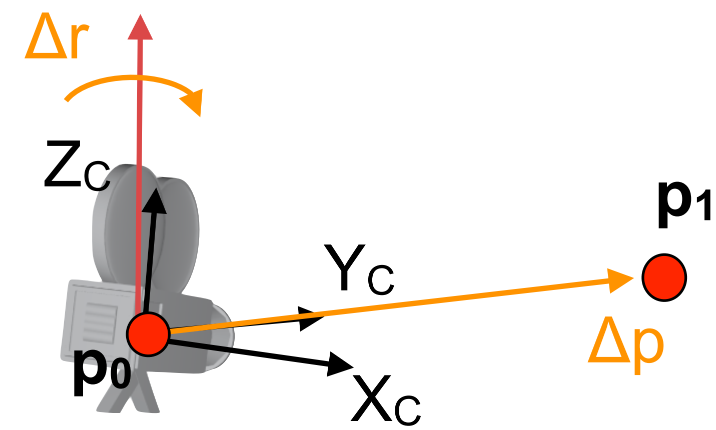


Mouse button pressed & repeated:

- *w* or *s* - adds a Δp into the current moving direction, represented by the model matrix, X_M, Y_M, Z_M.
- *a* or *d* - rotates the object around Z_M.

Relation input and action





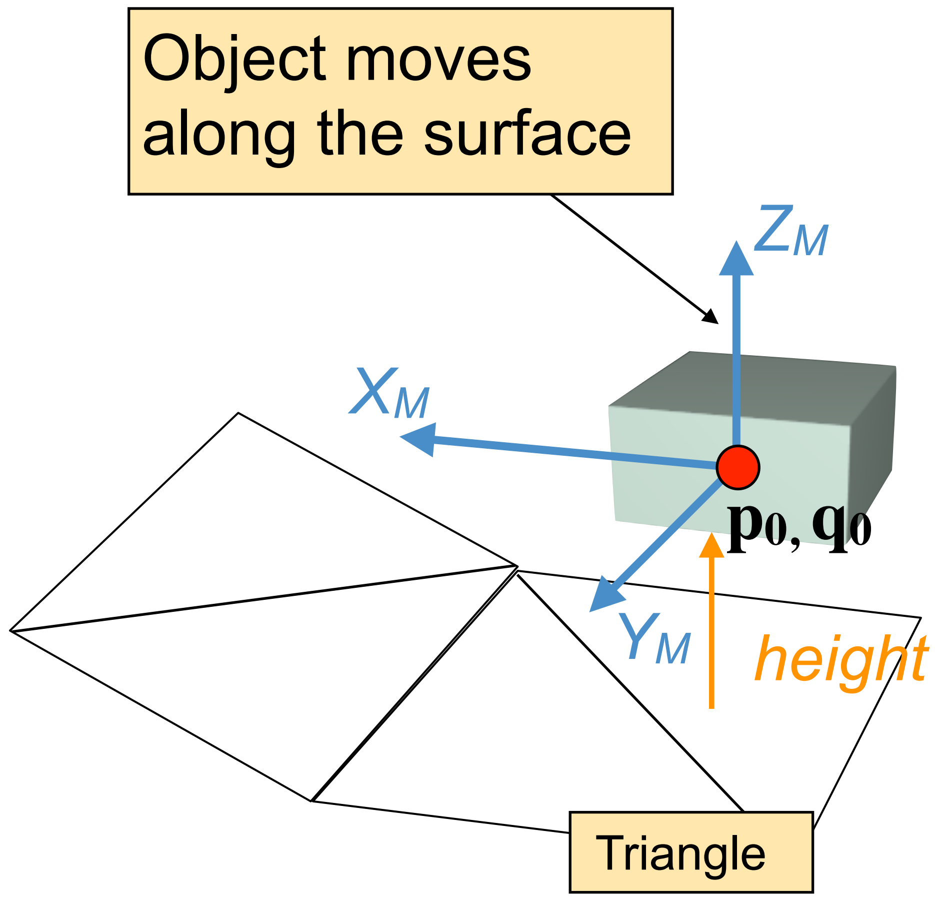
We use the model matrix in this example.

Instead, you can use the view matrix and move the camera around.

Mouse button pressed & repeated:

- *w* or *s* - adds a Δp into the current moving direction, represented by the model matrix, X_M, Y_M, Z_M.
- *a* or *d* - rotates the object around Z_M.

Object motion



The transformation of this object is represented as matrix:

$$\mathbf{M} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & x \\ r_{10} & r_{11} & r_{12} & y \\ r_{20} & r_{21} & r_{22} & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and as position (vector):

$$\mathbf{p} = \{x, y, z\}$$

and rotation (quaternion):

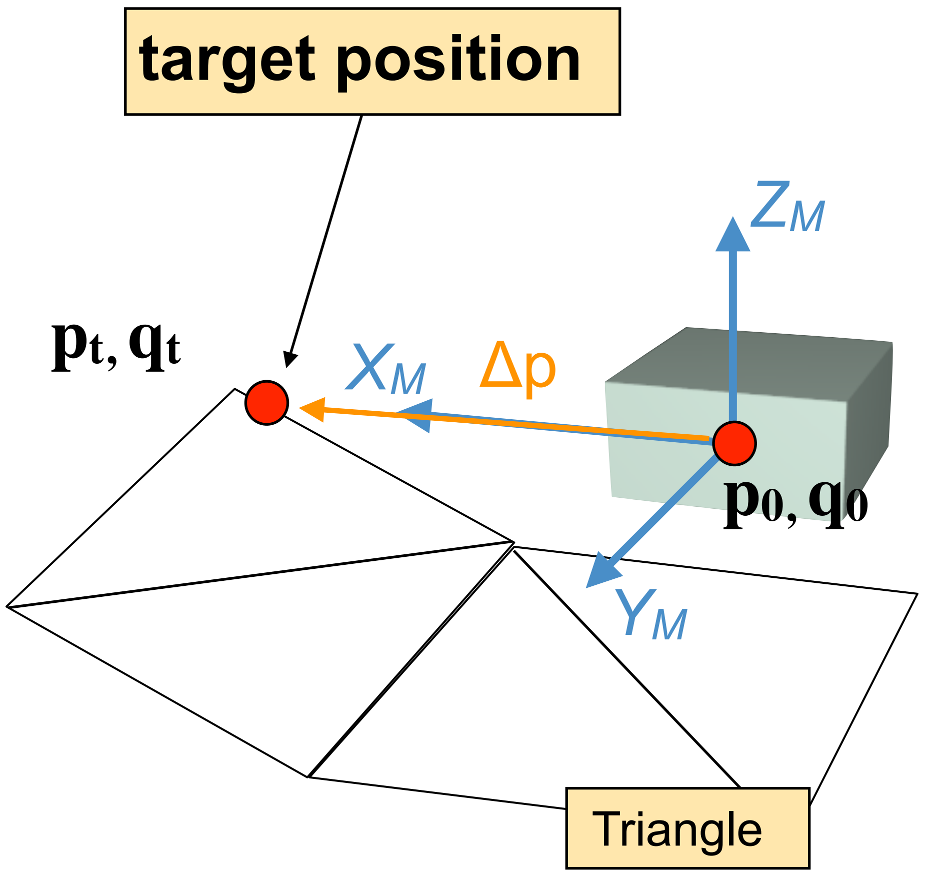
$$\mathbf{q} = \{s, x_i, y_i, z_i\}$$

$\mathbf{q}_i,\ \mathbf{p}_i,\ \mathbf{M}_i$ an index i indicate the frame number

Note, we can always switch between both representations

We also like to maintain a distance *height* over ground.

Object motion



Button pressed:

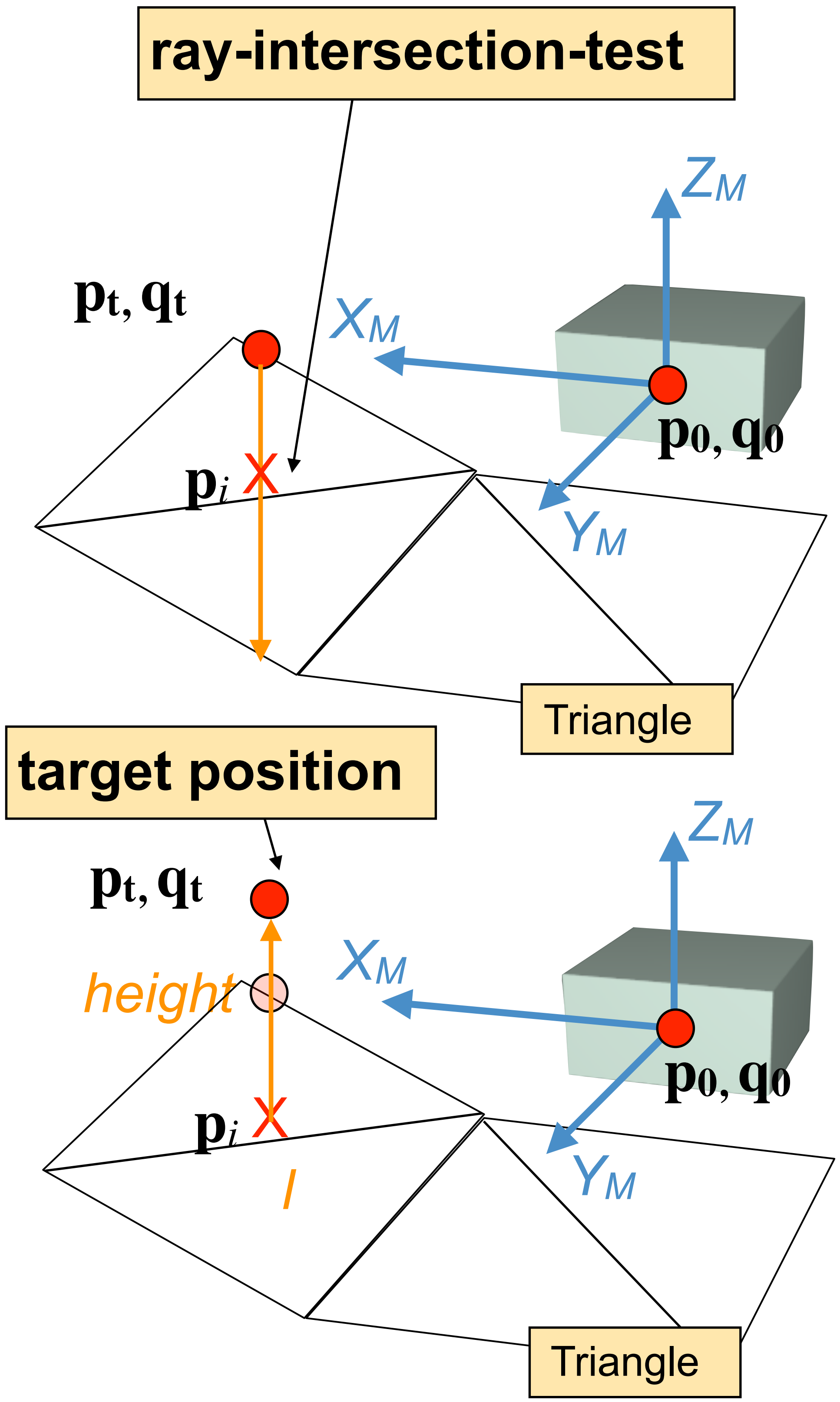
we add a value Δp to our current position and denote this position as **target position $\mathbf{p}_t$, $\mathbf{q}_t$**

$$\mathbf{p}_t = \mathbf{p}_0 + \Delta\mathbf{p}$$

or as matrix:

$$\mathbf{M}_t = \mathbf{M}_0 * \Delta\mathbf{M}$$

Object motion



Ray Intersection Test:

at position pt. perform a ray-intersection-test to find the triangle underneath your object.

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{l}$$

The intersection is given as

$$\mathbf{p}_i = \mathbf{p}_a + (\mathbf{p}_b - \mathbf{p}_a)t$$

Move the z-component (in this example, it is z.) to the height over ground you like to have

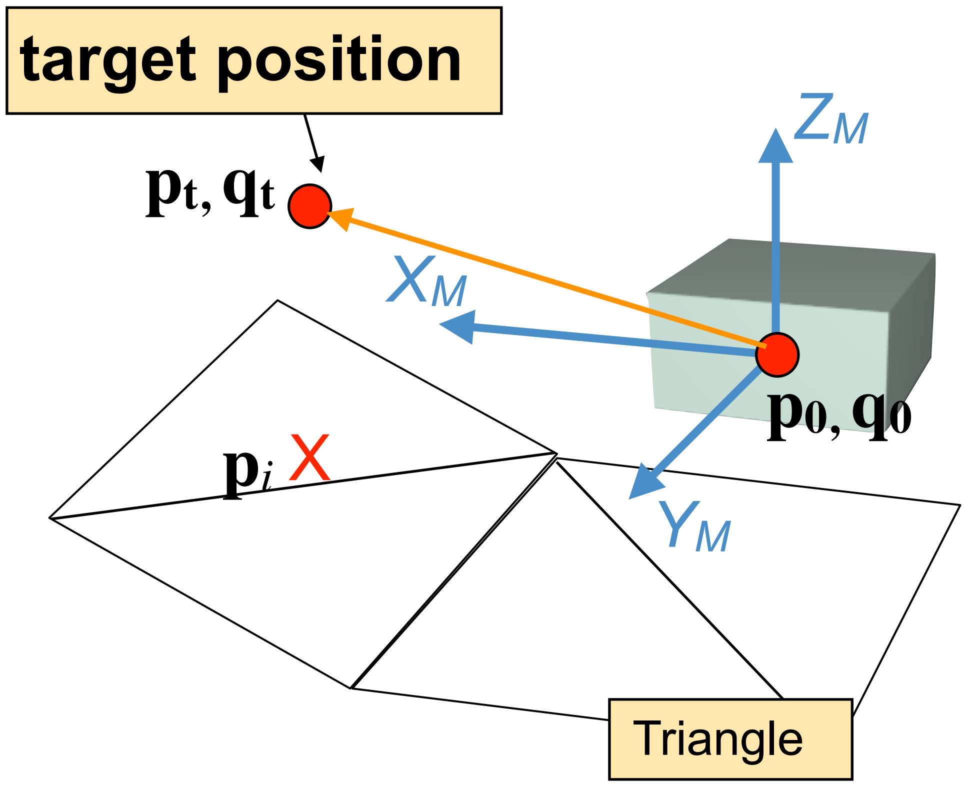
$$\mathbf{p}_t = \mathbf{p}_0 + \Delta\mathbf{p}$$

$$\mathbf{p}_t = \{x, y, p_{i,z} + height\}$$

Set the position as new target position for your object.

VRAC | HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

Object motion



Ray Intersection Test:

at position pt. perform a ray-intersection-test to find the triangle underneath your object.

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{l}$$

The intersection is given as

$$\mathbf{p}_i = \mathbf{p}_a + (\mathbf{p}_b - \mathbf{p}_a)t$$

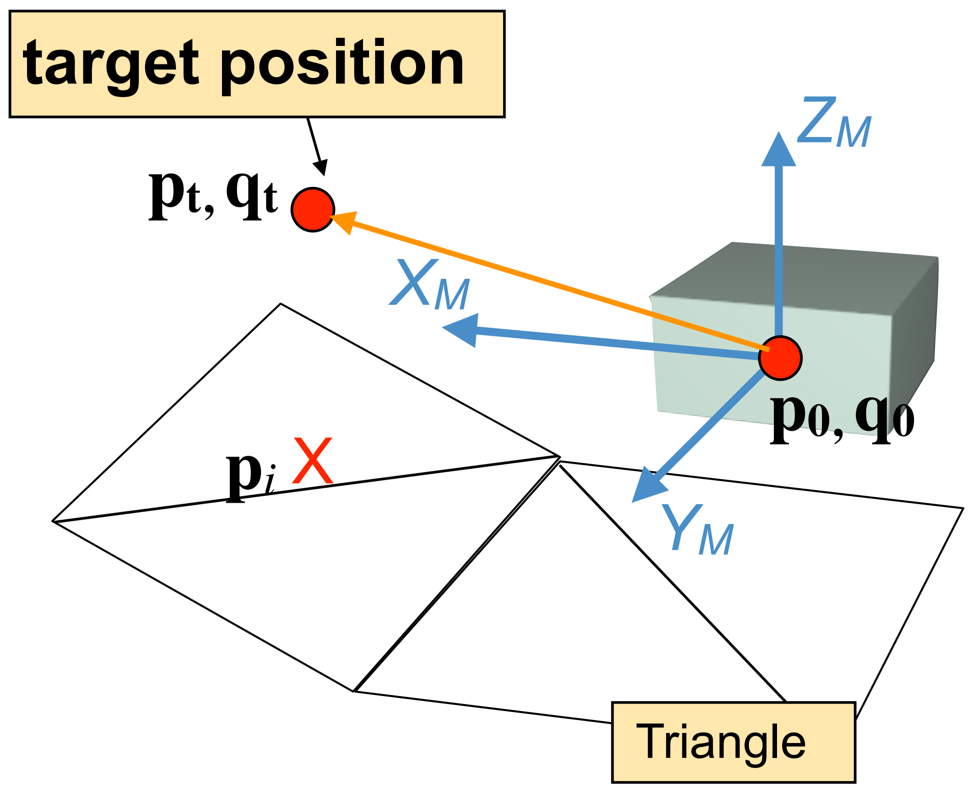
Move the z-component (in this example, it is z.) to the height over ground you like to have

$$\mathbf{p}_t = \mathbf{p}_0 + \Delta\mathbf{p}$$

$$\mathbf{p}_t = \{x, y, p_{i,z} + height\}$$

Set the position as new target position for your object.

Object motion



The current position

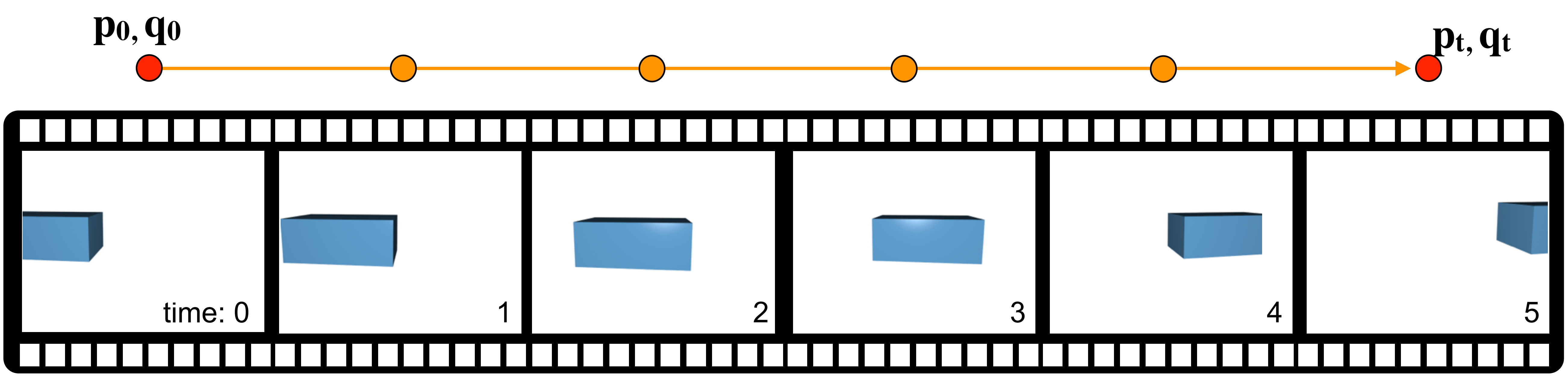
$$\mathbf{p}_0 = \{x, y, z\}$$

The target position:

$$\mathbf{p}_t = \{x, y, p_{i,z} + height\}$$

We split the motion vector into multiple position realize a smooth transition.

Keep in mind: the distance $\mathbf{p}_t$-$\mathbf{p}_0$ is const



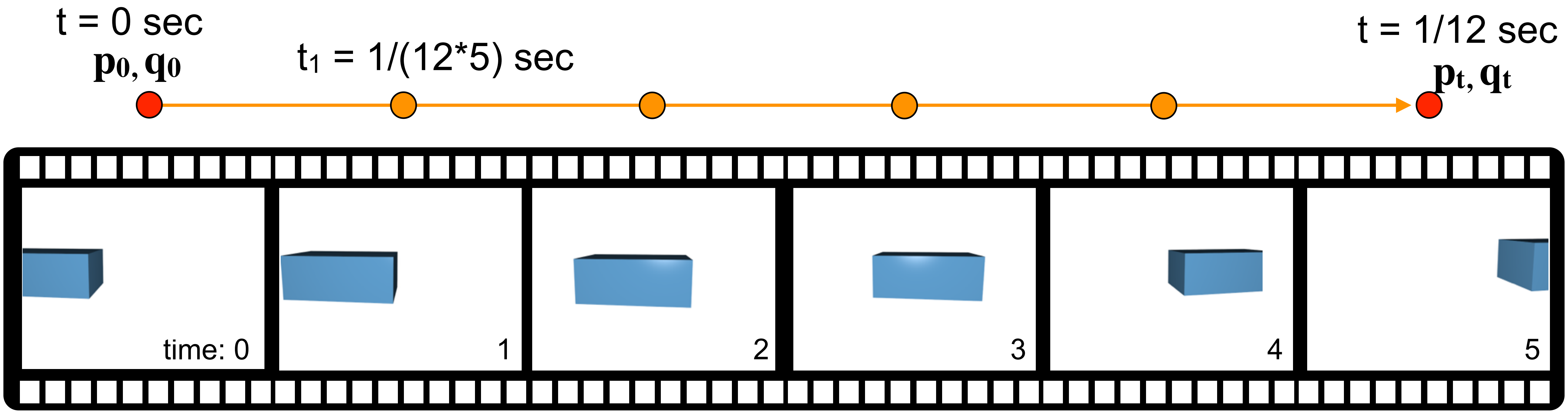
Linear Interpolation

$$\mathbf{p}_i = \mathbf{p}_0 + (\mathbf{p}_t - \mathbf{p}_0) \cdot v$$

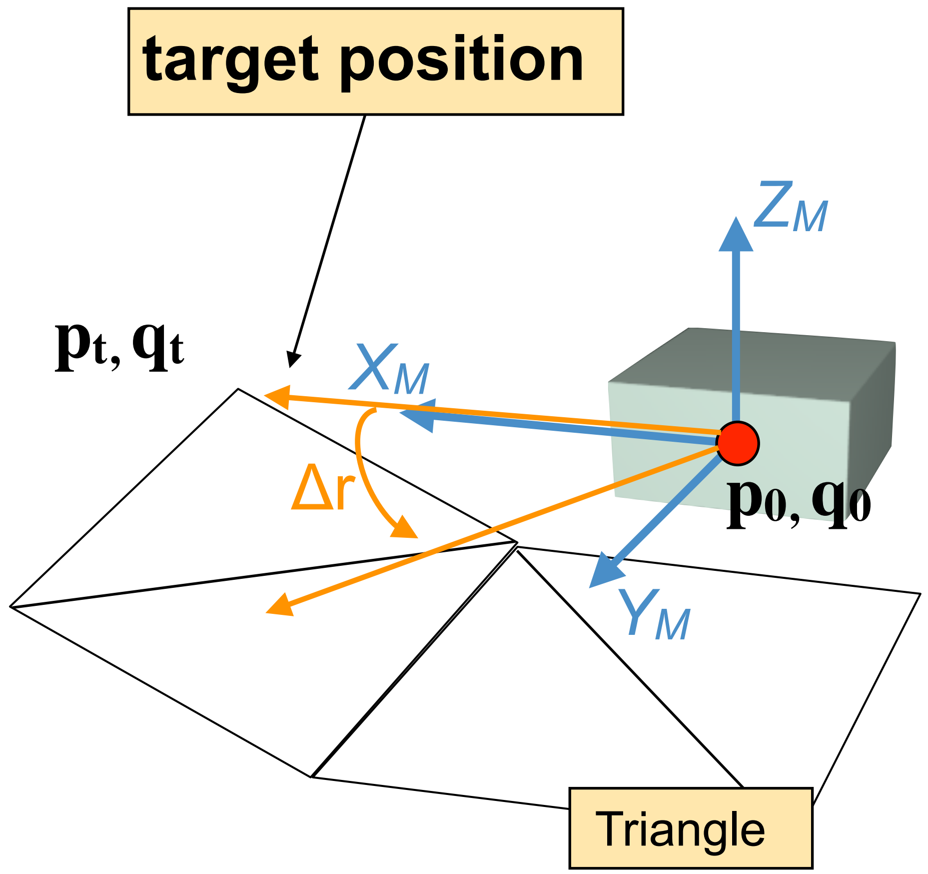
The factor v:

$$v = \frac{\text{events per sec}}{\text{framerate}} \cdot i = \frac{12}{60} \cdot i = \frac{1}{5} \cdot i$$

- 60 frames per seconds
- 12 key events per second
- 1/12 = time between two events
- 60/12 = 5 = frames between two events



For Object Rotation



Button pressed:

we add a value Δr to our current rotation and denote this position as **target rotation $\mathbf{q}_t$**

$$\mathbf{p}_t = \mathbf{p}_0 + \Delta\mathbf{p}$$

or as matrix:

$$\mathbf{M}_t = \mathbf{M}_0 * \Delta\mathbf{M}$$

Keep in mind: the rotation delta Δr is const

Quaternion Interpolation

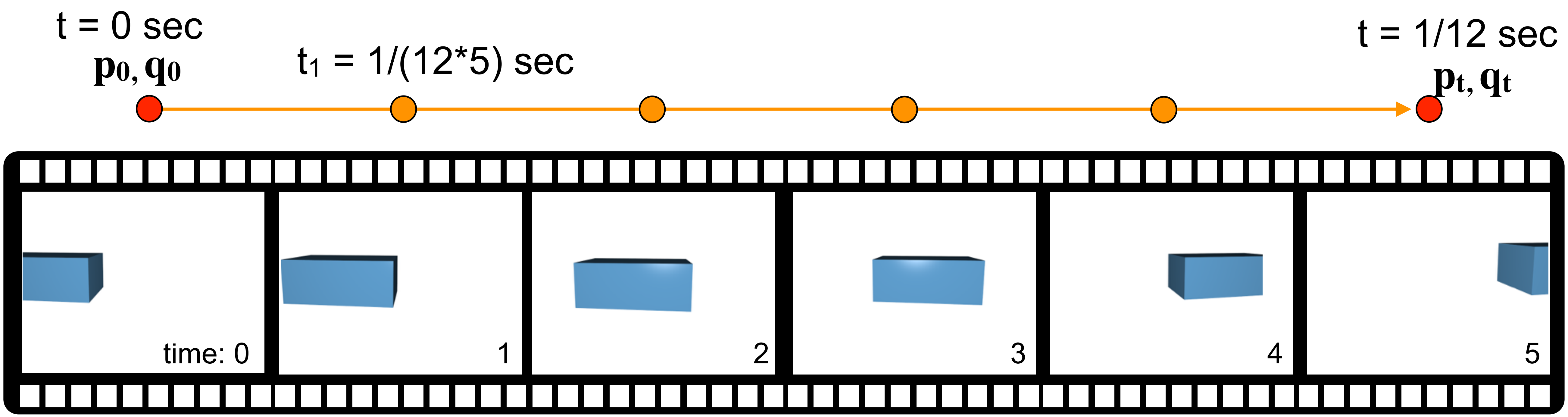
ARLAB

$$q_i = \mathrm{slerp}(\mathbf{q}_0, \mathbf{q}_t, v)$$

The factor v:

$$v = \frac{\Delta r}{5} = \frac{\Delta r}{5} = \frac{\Delta r}{\text{events}}$$

- 60 frames per seconds
- 12 key events per second
- 1/12 = time between two events
- 60/12 = 5 = frames between two events



VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Notes

- For translation and rotation, use the Δ**p** or Δ**r** to adjust the speed.
- For a very smooth transition, measure the frames per sec, typically 60, but it can deviate.
  - Note, the vertical synchronization (vsync) can be disabled with glfwSwapInterval(0) - renders as fast as possible
  - glfwSwapInterval(1) enables vsync
  - But you will not see more than you screen can refresh (display refresh rate typically 60 Hz ~ 75 Hz)
- The number of mouse events (60 per sec mouse motion and approx. 15 mouse button per sec) and keyboard events (12 per sec) is constant.

Timing

ARLAB

```
int main(int argc, const char * argv[])
{

        [....]

   while
   {   60 Hz         60

        Render the object
        draw object
            interpolate

        [....]
}   }
```

Program init

main loop

- The interpolation process belongs to the rendering process.
- The target values are set asynchronously when they arrive.

Mouse motion event
60 Hz

Mouse button event
15 Hz

Keyboard event
12 Hz

VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

SphereInt3D

Code example: 22_Navigation

SphereInt3D - implements a sphere with interpolated motion.

Parameters:

```
glm::mat4    _target_matrix;
glm::mat4    _current_matrix;
```

Stores the position / orientation as matrix

```
glm::vec3    _target_position;
glm::vec3    _current_position;
```

Stores the position as vector

```
glm::quat    _current_quat;
glm::quat    _target_quat;
```

Stores the orientation as quaternion

Function:

Set a model matrix to move the object around

```
void setMatrix(glm::mat4& matrix);
```

param matrix - the model matrix for this object.

setMatrix

```
void GLSphereInt3D::setMatrix(glm::mat4& matrix)
{
    ////////////////////////////////////////////
    // With interpolation
    if(_with_interpolation)
    {
        // with interpolation, remember the target location that you like to find
        _target_matrix = matrix;
        _target_quat = glm::quat_cast(matrix);
        _target_position[0] = _target_matrix[3][0];
        _target_position[1] = _target_matrix[3][1];
        _target_position[2] = _target_matrix[3][2];

        // Just for initialization. Move your object immediately to the location
        if(!_position_initialized)
        {
            _current_quat = glm::quat_cast(matrix);
            _current_position = _target_position;

            // update the matrix.
            _modelMatrix =  matrix;
            glUniformMatrix4fv(_modelMatrixLocation, 1, GL_FALSE, &_modelMatrix[0][0
            _position_initialized = true;
        }
    }
```

Here, we set the target position.

draw

```
void GLSphereInt3D::draw(void)
{
    ////////////////////////////////////////////////////
    // Renders the sphere

    // Enable the shader program
    glUseProgram(_program);

    // this changes the camera location
    glm::mat4 rotated_view =  rotatedViewMatrix();
    glUniformMatrix4fv(_viewMatrixLocation, 1, GL_FALSE, &rotated_view[0][0]); // se
    glUniformMatrix4fv(_inverseViewMatrixLocation, 1, GL_FALSE, &invRotatedViewMatri
    glUniformMatrix4fv(_modelMatrixLocation, 1, GL_FALSE, &_modelMatrix[0][0]); //

    // Bind the buffer and switch it to an active buffer
    glBindVertexArray(_vaoID[0]);

    ////////////////////////////
    // Call the interpolation function
    if(_with_interpolation) interpolateMat();

    // Draw the triangles
    glDrawArrays(GL_TRIANGLE_STRIP, 0, _num_vertices);
```

here we call the interpolation function

interpolateMat

```
/*!
 Interpolation function for the matrix /
 the position of the object
 */
void GLSphereInt3D::interpolateMat(void)
{
   // Calculate the distance between the target position and the current position.
   glm::vec3 temp = _target_position - _current_position;
   float distance = sqrt(dot(temp, temp));


   glm::quat temp_quad;
   temp_quad.x = _target_quat.x - _current_quat.x;
   temp_quad.y = _target_quat.y - _current_quat.y;
   temp_quad.z = _target_quat.z - _current_quat.z;
   temp_quad.w = _target_quat.w - _current_quat.w;

   // Calculate the distance between the target angle and the current angle.
   float delta_angle = sqrt( (_target_quat.x - _current_quat.x)*(_target_quat.x - _current_quat.x)  +
                       (_target_quat.y - _current_quat.y)*(_target_quat.y - _current_quat.y)  +
                       (_target_quat.z - _current_quat.z)*(_target_quat.z - _current_quat.z)  +
                      (_target_quat.w - _current_quat.w)*(_target_quat.w - _current_quat.w));

   // If the distance is too large, find the next step
   if (distance > 0.01 || delta_angle > 0.01) {

       // Linear interpolation of the position
       _current_position = _current_position + temp *  glm::vec3(0.08);

       // Linear interpolation of the rotation using slerp
       _current_quat = glm::slerp(_current_quat, _target_quat, 0.25f);

       // convert the quaternion to a matrix
       _target_matrix = glm::mat4_cast(_current_quat);

       // write the position back.
       _target_matrix[3][0] = _current_position[0];
       _target_matrix[3][1] = _current_position[1];
       _target_matrix[3][2] = _current_position[2];

       // update the model matrix.
       _modelMatrix = _target_matrix;
       glUniformMatrix4fv(_modelMatrixLocation, 1, GL_FALSE, &_modelMatrix[0][0]);
   }
}
```

Check the current distance

Interpolate if required

interpolateMat

```
// Calculate the distance between the target position and the current position.
glm::vec3 temp = _target_position - _current_position;
float distance = sqrt(dot(temp, temp));

glm::quat temp_quad;
temp_quad.x = _target_quat.x - _current_quat.x;
temp_quad.y = _target_quat.y - _current_quat.y;
temp_quad.z = _target_quat.z - _current_quat.z;
temp_quad.w = _target_quat.w - _current_quat.w;

// Calculate the distance between the target angle and the current angle.
float delta_angle = sqrt( (_target_quat.x - _current_quat.x)*(_target_quat.x -
_current_quat.x)  + (_target_quat.y - _current_quat.y)*(_target_quat.y -
_current_quat.y)  + (_target_quat.z - _current_quat.z)*(_target_quat.z -
_current_quat.z)  + (_target_quat.w - _current_quat.w)*(_target_quat.w -
_current_quat.w));
```

interpolateMat

ARLAB

```
// If the distance is too large, find the next step
if (distance > 0.01 || delta_angle > 0.01) {

   // Linear interpolation of the position
  _current_position = _current_position + temp *  glm::vec3(0.08);

   // Linear interpolation of the rotation using slerp
  _current_quat = glm::slerp(_current_quat, _target_quat, 0.25f);

   // convert the quaternion to a matrix
  _target_matrix = glm::mat4_cast(_current_quat);

   // write the position back.
  _target_matrix[3][0] = _current_position[0];
  _target_matrix[3][1] = _current_position[1];
  _target_matrix[3][2] = _current_position[2];

 // update the model matrix.
 _modelMatrix = _target_matrix;
 glUniformMatrix4fv(_modelMatrixLocation, 1, GL_FALSE, &_modelMatrix[0][0]);

}
```

glm::quat_cast, mat4_cast

The function allows us to convert a matrix into a quaternion:

```
template<typename T > detail::tquat< T >
quat_cast (detail::tmat4x4< T > const &x)
```

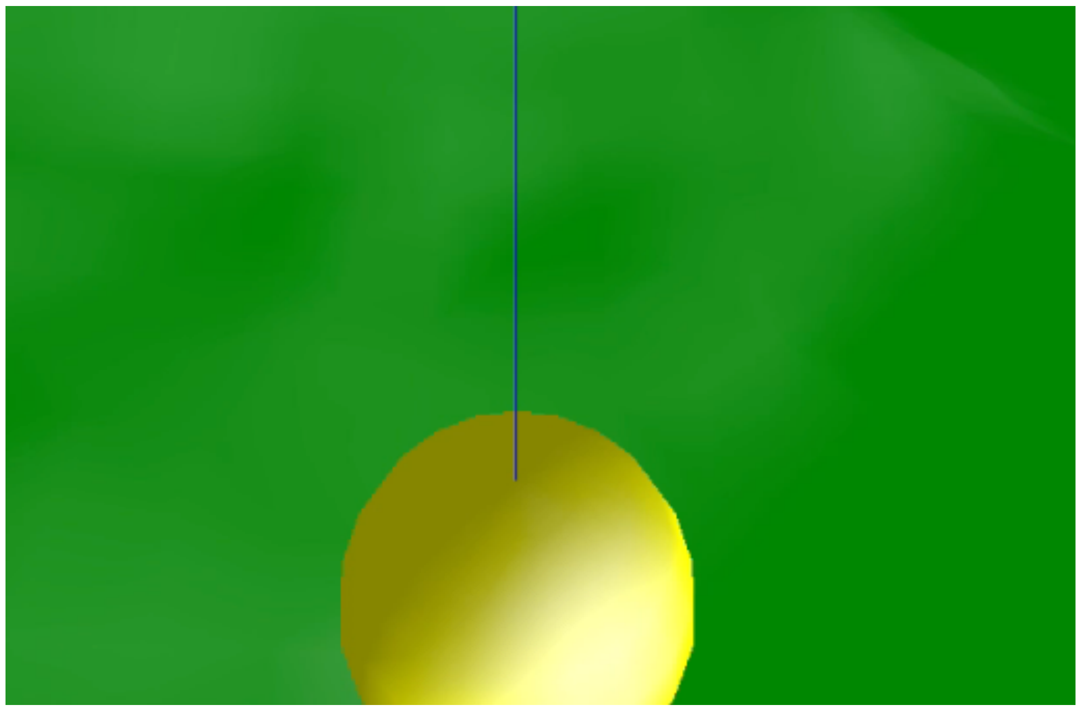
The function allows us to convert a quaternion into a matrix:

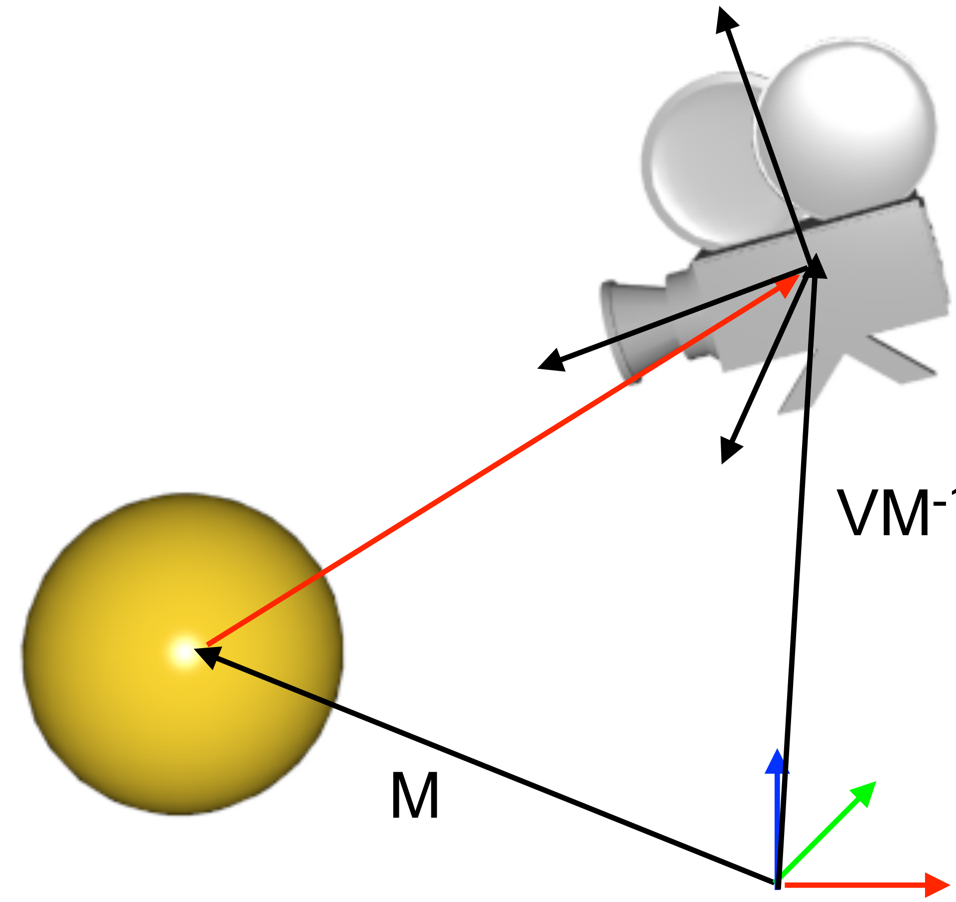
```
template<typename T > detail::tmat4x4< T >
mat4_cast (detail::tquat< T > const &x)
```

Note, only the orientation is transformed. The quaternion does not store a position.

Camera Motion

Camera is attached to the 3D model

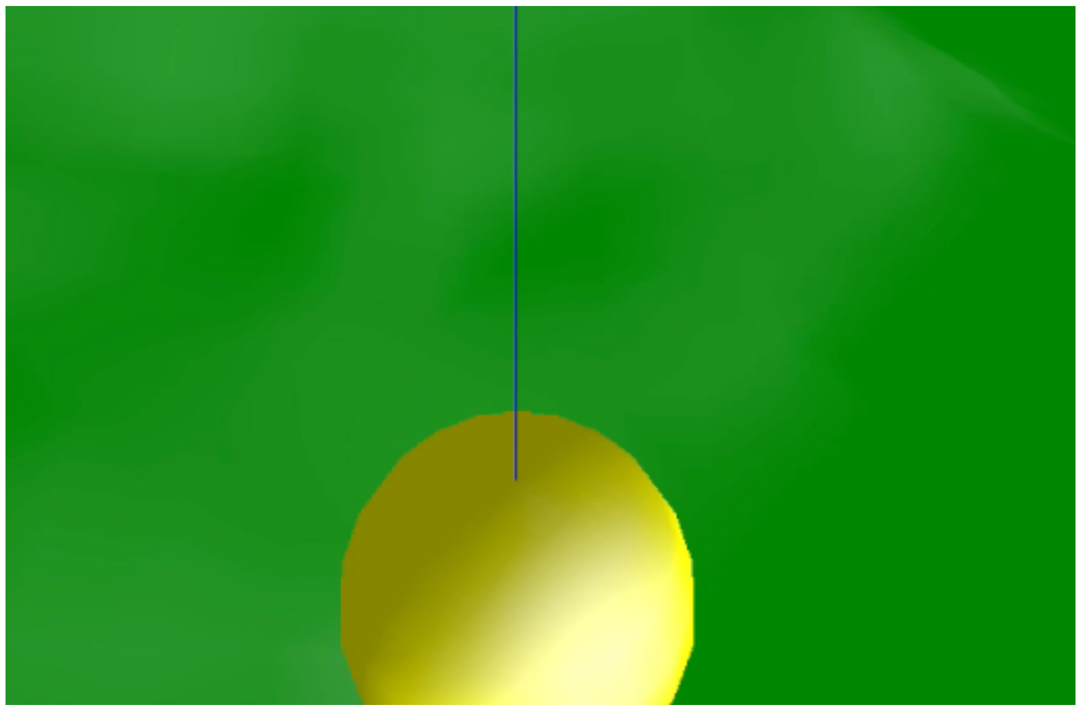


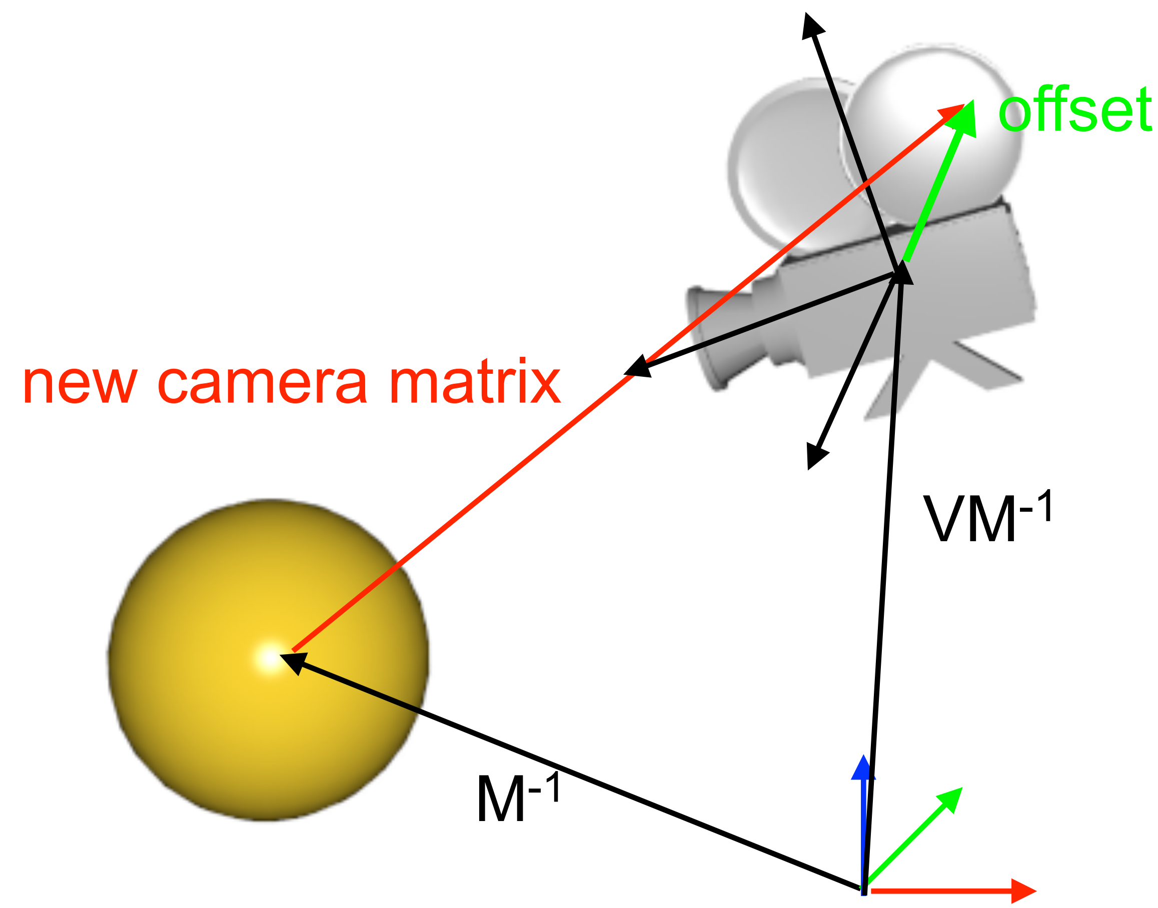


- Note, the view matrix VM - the return value from a lookAt function - represents the camera coordinate system.
- Representing this in world coordinates, it is the inverse view matrix.

Camera Motion

Camera is attached to the 3D model





In your main_navigation.cpp

```
// This code attaches the virtual camera just behind the object.
// Read the location from the object on the ground
object_transformation = sphere_result->getModelMatrix();

// Add the camera and a camera delta
camera_matrix = camera_delta *  camera_transformation *
glm::inverse(object_transformation);
```

Note

Key 1: switches between interpolated locomotion and non-interpolated locomotion

Key 2: switches the view matrix

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