Chapter 5

Input Devices and Interactive Techniques

Overview

This chapter will provide descriptions of several different types of input devices and how to control them, i.e. how to handle their input.

5.1 Input devices

Objectives

Introduce the basic input devices

- Physical Devices
- Logical Devices
- Input Modes

Event-driven input

Introduce double buffering for smooth animations

Programming event input with GLUT

5.1 Input devices

Project Sketchpad

Ivan Sutherland (MIT 1953) established the basic interactive paradigm that characterizes interactive computer graphics:

- User sees an *object* on the display
- User points to (*picks*) the object with an input device (light pen, mouse, trackball)
- Object changes (moves, rotates, morphs)
- Repeat

5.1 Input devices

Graphical Input

Devices can be described either by

- Physical properties
  - Mouse
  - Keyboard
  - Trackball
- Logical Properties
  - What is returned to program via API
    - **A position**
    - **An object identifier**

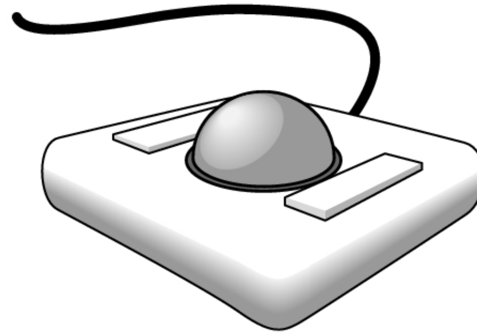
Modes

- How and when input is obtained
  - Request or event

5.1 Input devices

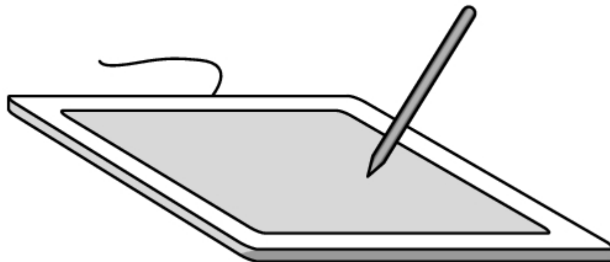
Physical Devices

mouse

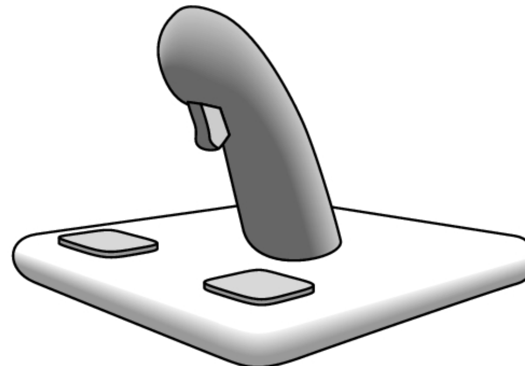


trackball

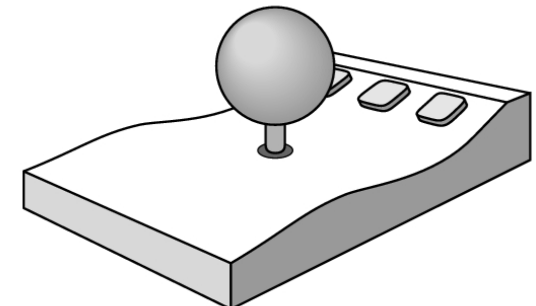
touch screen



data tablet



joy stick



space ball

5.2 Handling input

Incremental (Relative) Devices

Devices such as the data tablet return a position directly to the operating system

Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system

- Must integrate these inputs to obtain an absolute position
  - Rotation of cylinders in mouse
  - Roll of trackball
  - Difficult to obtain absolute position
  - Can get variable sensitivity

5.2 Handling input

Logical Devices

Consider the C and C++ code

- C++: `cin >> x;`
- C: `scanf ("%d", &x);`

What is the input device?

- Can't tell from the code
- Could be keyboard, file, output from another program

The code provides *logical input*

- A number (an `int`) is returned to the program regardless of the physical device

5.2 Handling input

Graphical Logical Devices

Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits

Two older APIs (GKS, PHIGS) defined six types of logical input

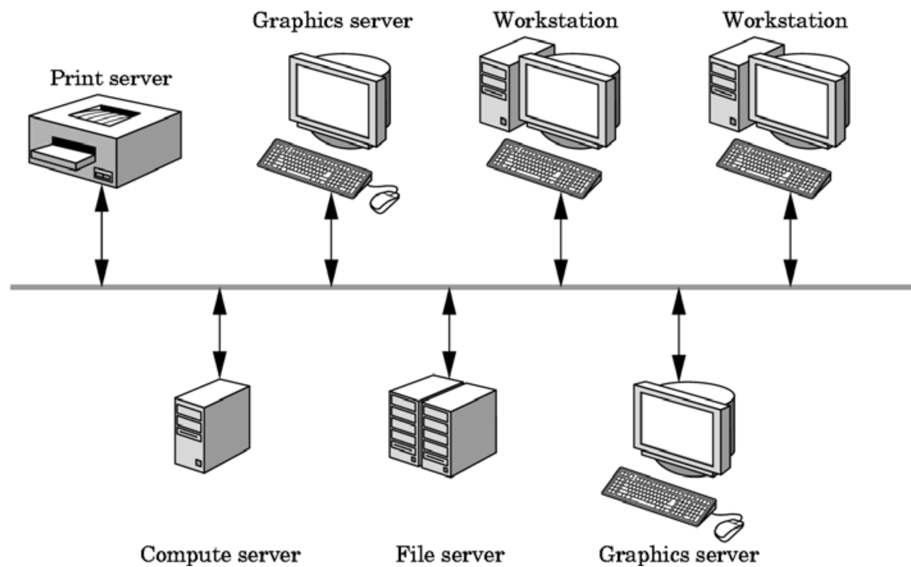
- **Locator**: return a position
- **Pick**: return ID of an object
- **Keyboard**: return strings of characters
- **Stroke**: return array of positions
- **Valuator**: return floating point number
- **Choice**: return one of n items

5.2 Handling input

The X Window System introduced a client-server model for a network of workstations

Client: OpenGL program

Graphics Server: bitmap display with a pointing device and a keyboard



5.2 Handling input

Input Modes

Input devices contain a *trigger* which can be used to send a signal to the operating system

- Button on mouse
- Pressing or releasing a key

When triggered, input devices return information (their *measure*) to the system

- Mouse returns position information
- Keyboard returns ASCII code

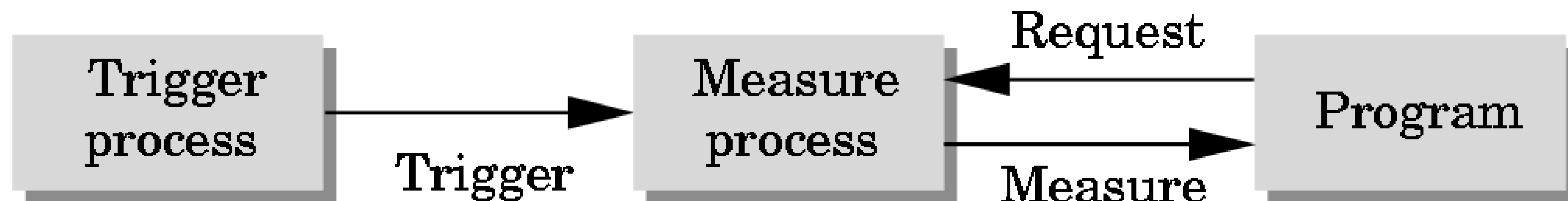
5.2 Handling input

Request Mode

Input provided to program only when user triggers the device

Typical of keyboard input

- Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed

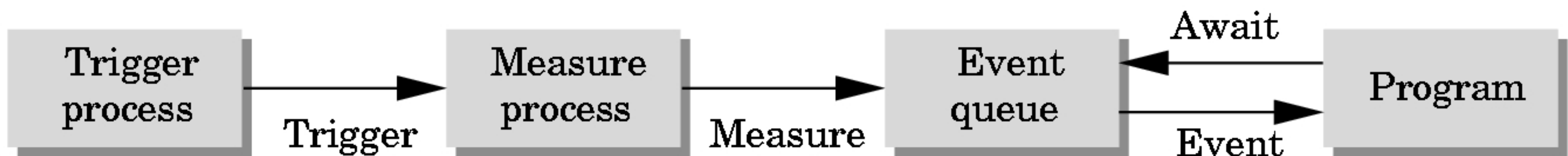


5.2 Handling input

Event Mode

Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user

Each trigger generates an *event* whose measure is put in an *event queue* which can be examined by the user program



5.2 Handling input

Event Types

Window: resize, expose, iconify

Mouse: click one or more buttons

Motion: move mouse

Keyboard: press or release a key

Idle: nonevent

Define what should be done if no other event is in queue

5.2 Handling input

Callbacks

Programming interface for event-driven input

Define a *callback function* for each type of event the graphics system recognizes

This user-supplied function is executed when the event occurs

GLUT example: `glutMouseFunc(mymouse)`

5.2 Handling input

GLUT callbacks

GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)

```
glutDisplayFunc
glutMouseFunc
glutReshapeFunc
glutKeyboardFunc
glutIdleFunc
glutMotionFunc, glutPassiveMotionFunc
```

5.2 Handling input

GLUT Event Loop

Recall that the last line in **`main.c`** for a program using GLUT must be

```
glutMainLoop();
```

which puts the program in an infinite event loop

In each pass through the event loop, GLUT

- looks at the events in the queue
- for each event in the queue, GLUT executes the appropriate callback function if one is defined
- if no callback is defined for the event, the event is ignored

5.2 Handling input

The display callback

The display callback is executed whenever GLUT determines that the window should be refreshed, for example

- When the window is first opened
- When the window is reshaped
- When a window is exposed
- When the user program decides it wants to change the display

In `main.c`

- **`glutDisplayFunc(mydisplay)`** identifies the function to be executed
- Every GLUT program must have a display callback

5.2 Handling input

Posting redisplays

Many events may invoke the display callback function

Can lead to multiple executions of the display callback on a single pass through the event loop

We can avoid this problem by instead using

```
glutPostRedisplay();
```

which sets a flag.

GLUT checks to see if the flag is set at the end of the event loop

If set then the display callback function is executed

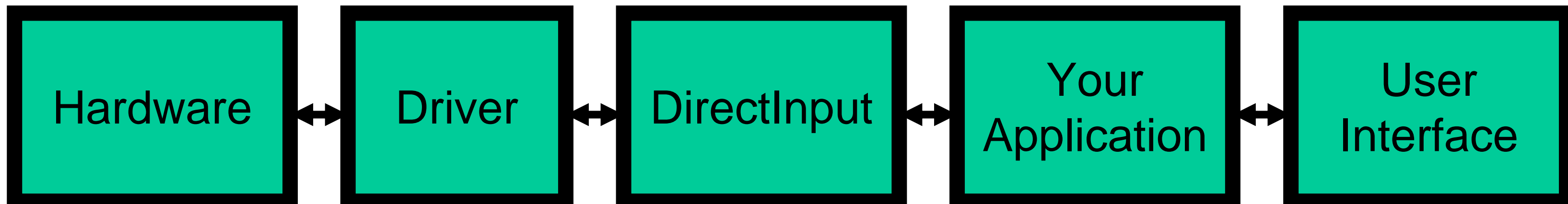
5.2 Handling input

Specialized hardware often requires the use of a vendor specific API. When using gaming devices for input, however, most operating systems provide a standard API than can be used. In Windows, for example, DirectInput allows us to directly support any type of gaming device. Linux as well provides support for a multitude of game devices directly without the need for specific drivers.

5.2 Handling input

What is DirectInput?

- A part of the DirectX library
- An application programming interface (API) to handle input devices



Note: DirectInput communicates with Windows driver directly and does not rely on Windows message

5.2 Handling input

What Can DirectInput Do?

Control keyboard, mouse, joystick

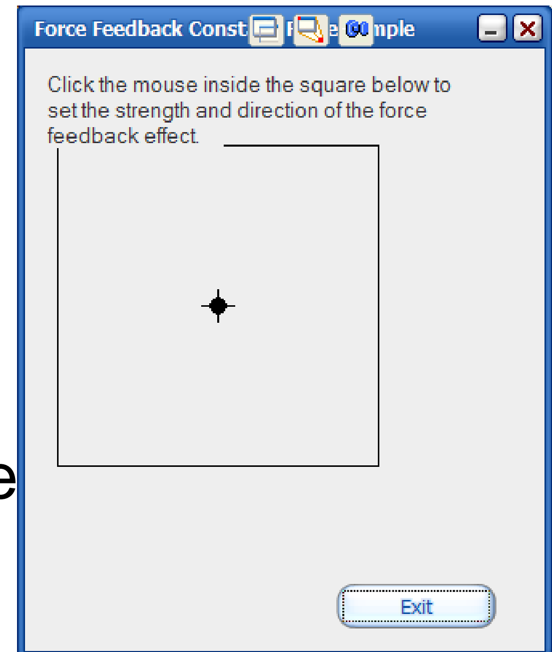
- Any device which is not keyboard nor mouse is a joystick

Handle force feedback

- Control the amount and direct of force generated

Action mapping

- Your application receive only game events (e.g. fire) rather than hardware events (e.g. mouse click)



5.2 Handling input

Key Terms in DirectInput

DirectInput object
- The root DirectInput interface

Device
- A keyboard, mouse, joystick

DirectInputDevice object
- Code representing a keyboard, mouse, joystick

Device object
- Code representing a key, button, trigger, and so on found on a DirectInput device object, also called device object instance

5.2 Handling input

Create the DirectInput Object

Use the DirectInput8Create() function

Example:

```
DirectInput8Create( GetModuleHandle(NULL),
DIRECTINPUT_VERSION, IID_IDirectInput8,
(VOID**)&g_pDI, NULL )
```

Highlight:

`DIRECTINPUT_VERSION` is the current DirectInput version

`IID_IDirectInput8` is the interface we wish to create, now it is a DirectInput 8 interface

`g_pDI` is type of `LPDIRECTINPUT8`, similar to `DIRECTINPUT8*`

5.2 Handling input

Enumerate Devices

Use the `EnumDevices()` function from the DirectInput8 interface

Example

```
g_pDI->EnumDevices(
    DI8DEVCLASS_GAMECTRL,
    DeviceCallback, NULL,
    DIEDFL_ATTACHEDONLY | DIEDFL_FORCEFEEDBACK )
```

Highlight:

- At first, we specify the device we wish to find, which could be: `DI8DEVCLASS_ALL`, `DI8DEVCLASS_GAMECTRL`, `DI8DEVCLASS_KEYBOARD`, `DI8DEVCLASS_POINTER`, etc.
- DeviceCallback is a user-defined callback function which will be called to create the device
- `DIEDFL_ATTACHEDONLY | DIEDFL_FORCEFEEDBACK` are two options which means we only find attached and force feedback device

5.2 Handling input

Create a `DirectInputDevice` Object for Each Device

In the callback function, we create the device using `CreateDevice()` function.

The callback function must be something like:

```
BOOL CALLBACK
DeviceCallback(LPCDIDEVICEINSTANCE pInst,
LPVOID pvRef);
```

Then we apply the create device function:

```
g_pDI->CreateDevice( pInst->guidInstance,
&pDevice, NULL );
```

where `pDevice` is the device pointer we want.

5.2 Handling input

Set Up the Device

- Get the device capabilities (optional)
- Enumerate the keys, buttons, and axes on the device (optional)
- Set the cooperative level (highly recommended)
- Set the data format (required)
- Set the device properties (you must at least set the buffer size if you intend to get buffered data)

5.2 Handling input

Set Up the Device - Device Capabilities

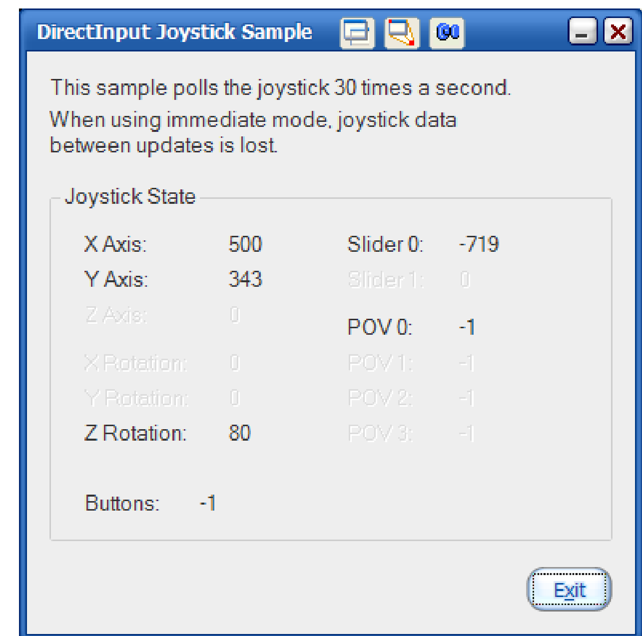
Use the `GetCapabilities()` method from the device pointer.

Example:

```
pDevice
->GetCapabilities(&DICaps)
```

where `DICaps` is a structure of type `DIDEVCAPS`.

Then, we can see the capabilities in the structure such as number of axes, buttons, POVs, etc.



5.2 Handling input

Set Up the Device – Device Object Enumeration

To find what button or axis available, use the `EnumObject` method of the device pointer.

We can enumerate different device objects such as DIDFT_AXIS, DIDFT_BUTTON, DIDFT_POV, etc.

5.2 Handling input

Set Up the Device – Cooperative Level

The cooperative level of a device determines how the input is shared with other applications. Two options are foreground and background which means that the application can get input only if in focus or not.

Exclusive / Non exclusive means that the application is the only one to process the input device or not. Use the `SetCooperativeLevel()` method of the device pointer to adjust the settings.

Example:

```
pDevice->SetCooperativeLevel(hwnd,
DISCL_NONEXCLUSIVE | DISCL_FOREGROUND)
```

5.2 Handling input

Set Up the Device – Data Format

You could use your own data format or predefined DirectInput data format. Use the `SetDataFormat()` method in the device pointer to adjust.

Example:

```
pDevice->SetDataFormat(c_dfDIJoystick)
```

Set Up the Device – Device Properties

Set up properties such as buffer size, auto center, dead zone, saturation, etc. using the `SetProperty()` method of the device pointer with different switch.

5.2 Handling input

Acquire the device

You need to acquire a device when you wish to get data from it. The device is automatically un-acquired when your application loses focus.

Hence, you need to acquire many times in an application. Use the `Acquire()` method of the device pointer to acquire the device. Once acquired, you cannot change the properties of the device pointer.

5.2 Handling input

Retrieve Data

Call the `Poll()` function of the device pointer to see if there are new data.

Use the `GetDeviceState()` method of the device pointer to get the current state and store it in a predefined data structure. If you set the device format as the default DirectX format for joy sticks as previously shown, this looks as follows:

```
DIJOYSTATE state;
GetDeviceState (sizeof (DIJOYSTATE),
                    &state);
```

5.2 Handling input

Retrieve Data

The structure `DIJOYSTATE` then reflects the current values of the joystick:

```
typedef struct DIJOYSTATE {
LONG lX;
LONG lY;
LONG lZ;
LONG lRx;
LONG lRy;
LONG lRz;
LONG rglSlider[2];
DWORD rgdwPOV[4];
BYTE rgbButtons[32]; } DIJOYSTATE, *LPDIJOYSTATE;
```

5.2 Handling input

Act on Data

Use your imagination

5.2 Handling input

Close DirectInput

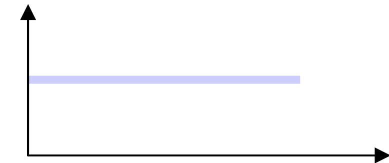
Un-acquire all devices by calling the `Unacquire ()` method of the device and the `Release()` method of all device pointers, as well as the DirectInput Object.

5.2 Handling input

Force Feedback Type

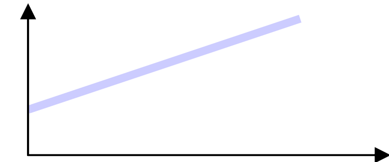
Constant force

Steady force in a single direction



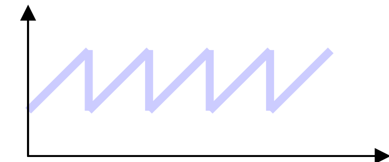
Ramp force

Force steadily increase or decrease magnitude



Periodic force

Repeating a defined wave pattern

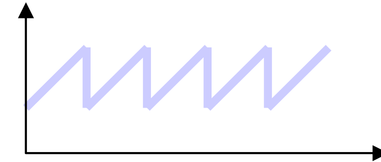


Condition force

Force react to the direction of movement, such as fiction and spring force

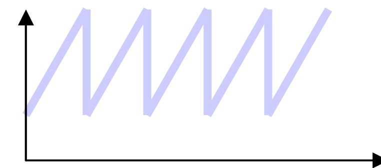
5.2 Handling input

Force Modifier



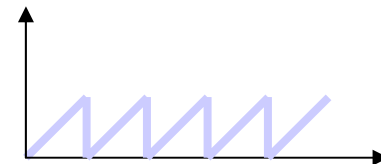
Gain

Magnify the force



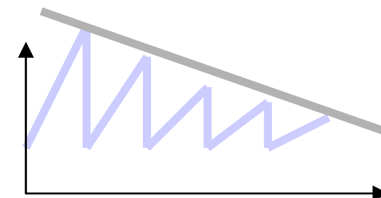
Offset

Shift the magnitude of the force towards +ve or –ve



Envelope

Shape the magnitude of the force



5.2 Handling input

Creating a Force Effect

Use the `CreateEffect()` method of the device pointer

You could choose to create `GUID_ConstantForce`, `GUID_RampForce`, `GUID_Square`, `GUID_Sine`, etc.

You have to specify the size, gain, direction, start delay, etc. in the `DIEFFECT` structure

Example

```
g_pDevice->CreateEffect(GUID_ConstantForce, &effStructure, &g_pEffect, NULL )
```

5.2 Handling input

To Play an Effect

- Call the `Download()` function of the effect pointer to download the effect to the device
- Call the `Start()` function of the effect pointer to play the effect
- Call the `Unload()` function of the effect pointer to unload the effect from the device

5.2 Handling input

Enumerate Effect

- A joystick may not support all kind of force feedback effects.
- You need to enumerate the supported effect before you can play it.
- It is similar to enumerate the supported device in the computer.

5.2 Handling input

Game devices in Linux

The Linux operating system provides control over games devices through a device file, e.g. `/dev/input/js0`.

Hence, by opening this device file we have all we need:

```
fd = open("/dev/input/js0", O_RDONLY));
fcntl(fd, F_SETFL, O_NONBLOCK);
```

5.2 Handling input

Game devices in Linux

To get information about the game device, we can use the system call `ioctl`:

```
unsigned char axes;
unsigned char buttons;
ioctl(fd, JSIOCGAXES, &axes);
ioctl(fd, JSIOCGBUTTONS, &buttons);
```

This gives us the number of axes and buttons available on the game device.

5.2 Handling input

Game devices in Linux

Whenever a status of the game device changes, we can retrieve information about this change in an event:

```
struct js_event {
 __u32 time; /* event timestamp in
                milliseconds */
 __s15 value;/* value */
 __u8 type;  /* event type */
 __u8 number;    /* axis/button number */
};
```

5.2 Handling input

Game devices in Linux

Similar to windows, you need to poll the game device using the `read` system call:

```
struct js_event e;
read (fd, &e, sizeof(struct js_event));
```

If the poll was successful read will return the size of the data structure `js_event`, i.e. when the poll resulted in a joystick event.

5.2 Handling input

Game devices in Linux

Linux also supports force feedback on several game devices. As usual in Linux, this again is achieved by using a device file, e.g. `/dev/input/event0`.

Using `ioctl`, we can check what kind of force feedback is available:

```
unsigned long features[size];
ioctl(fd, EVIOCGBIT(EV_FF, size),
      features);
```

To determine the size of the list of features use:

```
ioctl(fd, EVIOCGEFFECTS, &size);
```

5.2 Handling input

Game devices in Linux

The list of features is a bitfield and can contain one or more of the following:

- `FF_X` has an X axis (usually joysticks)
- `FF_Y` has an Y axis (usually joysticks)
- `FF_WHEEL` has a wheel (usually steering wheels)
- `FF_CONSTANT` can render constant force effects
- `FF_PERIODIC` can render periodic effects (sine, triangle, square...)
- `FF_RAMP` can render ramp effects
- `FF_SPRING` can simulate the presence of a spring
- `FF_FRICTION` can simulate friction
- `FF_DAMPER` can simulate damper effects
- `FF_RUMBLE` rumble effects (normally the only effect supported by rumble pads)
- `FF_INERTIA` can simulate inertia

5.2 Handling input

Game devices in Linux

The include file `linux/input.h` contains the structure `effect`. This data structure is used to define the force feedback effect, i.e. if it resembles a constant force, a ramp force, a periodic force, or a rumble effect. If provided by the device, a direction for the force feedback can be specified (up, down, left, or right). By combining force feedback effects for x- and y-axis individually, an arbitrary force direction can be achieved.

5.2 Handling input

Game devices in Linux

To upload a force feedback effect into the device's memory:

```
struct ff_effect effect;
int fd;
ioctl(fd, EVIOCSFF, &effect);
```

The structure `effect` contains an entry `id` This field has to be set to `-1` and will be adjusted by the driver once the effect is uploaded.

5.2 Handling input

Game devices in Linux

To remove a force feedback effect from the device's memory:

```
struct ff_effect effect;
int fd;
ioctl(fd, EVIOCRMFF, effect.id);
```

5.2 Handling input

Game devices in Linux

To play a force feedback effect something like this could be used:

```
struct input_event play;
struct ff_effect effect;
int fd;
fd = open("/dev/input/event0", O_RDWR);
/* Play three times */
play.type = EV_FF;
play.code = effect.id;
play.value = 3;
write(fd, (const void*) &play, sizeof(play));
```

5.2 Handling input

Game devices in Linux

The previous sequence of commands plays the force feedback effect three times. To stop it before it finishes, use:

```
struct input_event stop;
struct ff_effect effect;
stop.type = EV_FF;
stop.code = effect.id;
stop.value = 0;
write(fd, (const void*) &play, sizeof(stop));
```

5.2 Handling input

3-D input devices

Phantom Omni (Sensable)

5.2 Handling input

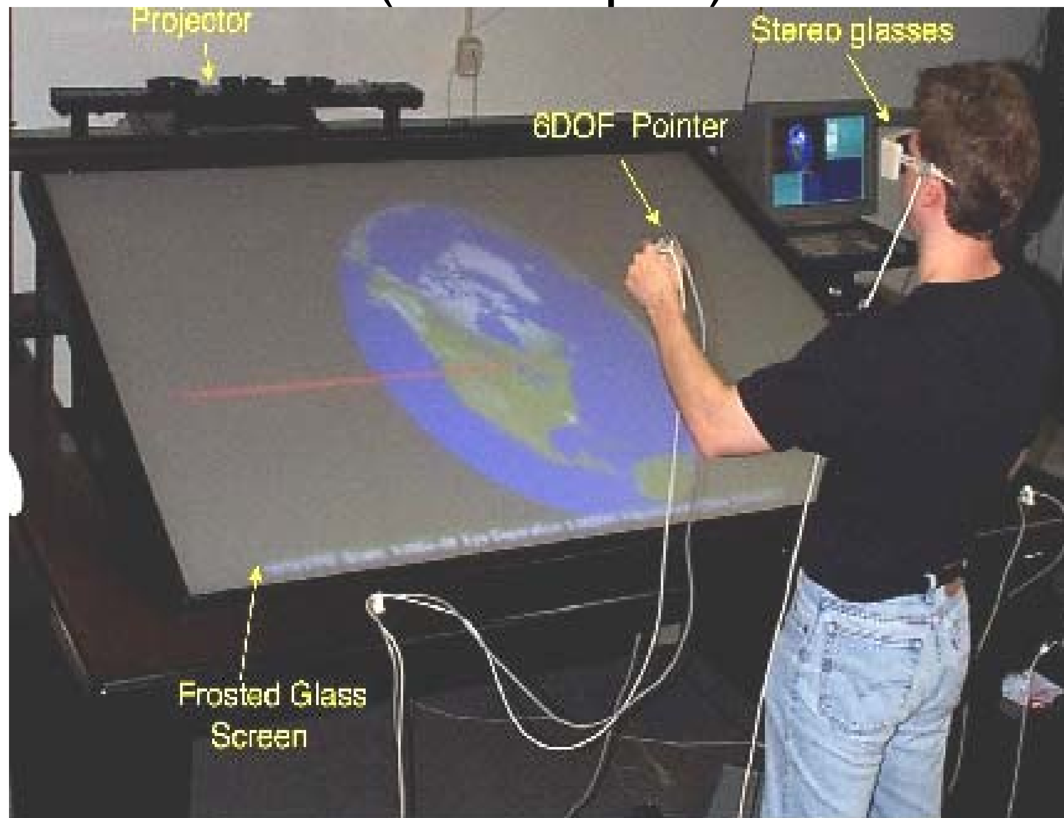
3-D input devices

CyberGrasp / CyberForce

5.2 Handling input

3-D input devices

Immersive Workbench (Fakesapce)



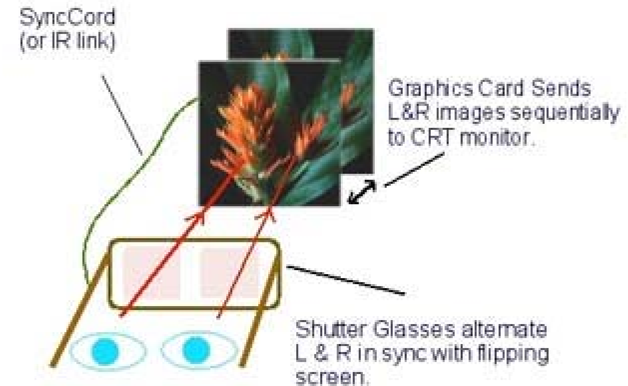
5.2 Handling input

3-D input devices

Most of these specialized 3-D input devices come with their own drivers and API. There is no common interface. Often times, only a limited choice of operating systems is supported (mostly Windows).

5.3 3-D Rendering in OpenGL

To achieve a 3-D viewing effect, shutter-glasses can be used to show different images to the left and right eye. The shutter-glasses use filters that can block or let pass incoming light. This property can be changed electrically. By showing alternating images for the left and right eye and letting the light pass only for the corresponding eye, 3-dimensional viewing is possible.



5.3 3-D Rendering in OpenGL

How can we achieve, that OpenGL shows different images for the left and right eye?

We saw earlier how to enable double buffering using the GLUT library:

```
glutInitDisplayMode(GLUT_DOUBLE |
   GLUT_RGB | GLUT_DEPTH);
```

Similarly, we can enable the stereo mode, i.e. show different images for the left and right eye:

```
glutInitDisplayMode(GLUT_DOUBLE |
   GLUT_RGB | GLUT_DEPTH | GLUT_STEREO);
```

5.3 3-D Rendering in OpenGL

Now we need to draw separate images for the left and right eye. OpenGL therefore provides two sets of buffers (similar to double buffering), one for the left and one for the right eye. Hence, we just need to enable one of the sets of buffers and draw the scene, switch to the other set and draw again:

```
glDrawBuffer(GL_BACK_LEFT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// draw the scene
glDrawBuffer(GL_BACK_RIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// draw the same scene again
```

5.3 3-D Rendering in OpenGL

Note, that some graphic cards are optimized to clear both buffers (left and right) at the same time. Thus, it might be significantly faster to clear them at the same time:

```
glDrawBuffer(GL_BACK);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glDrawBuffer(GL_BACK_LEFT);
// draw the scene
glDrawBuffer(GL_BACK_RIGHT);
// draw the same scene again
```

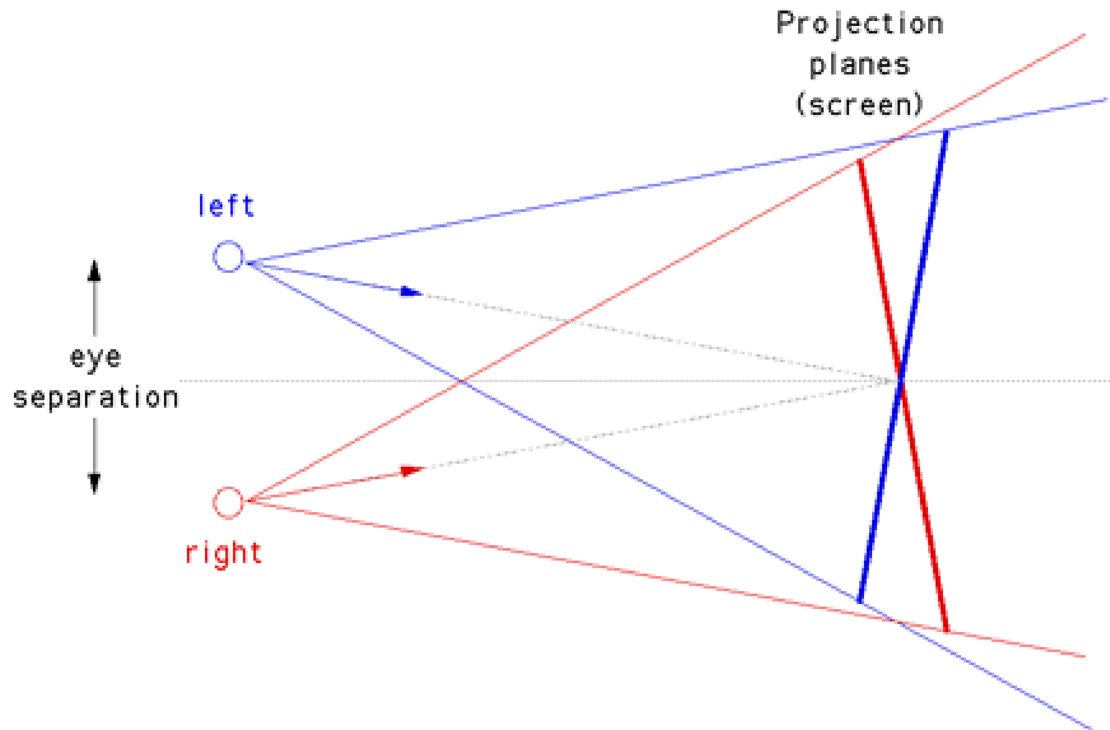
5.3 3-D Rendering in OpenGL

Now that we know how to generate two different views onto our scene in an alternating fashion: how do we achieve 3-D effect?

Mimic the way we would see the scene with our own eyes while standing in front of it. This means that we need to place two cameras pointing towards the scene. The distance between these cameras should be in accordance to the human eye distance.

5.3 3-D Rendering in OpenGL

The common approach is the so-called **toe-in** where the camera for the left and right eye is pointed towards a single focal point and `gluPerspective()` is used.



5.3 3-D Rendering in OpenGL

In OpenGL this might look like this:

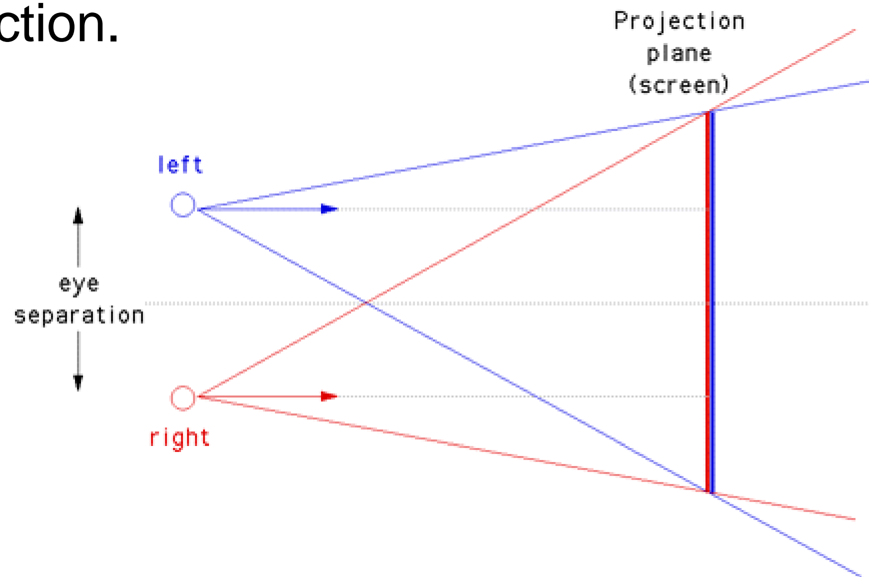
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(camera.aperture,
               screenwidth/(double)screenheight,
               0.1,10000.0);
CROSSPROD(camera.vd,camera.vu,right);
Normalise(&right);
right.x *= camera.eyesep / 2.0;
right.y *= camera.eyesep / 2.0;
right.z *= camera.eyesep / 2.0;
glMatrixMode(GL_MODELVIEW);
glDrawBuffer(GL_BACK);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

5.3 3-D Rendering in OpenGL

```
glDrawBuffer(GL_BACK_RIGHT);
glLoadIdentity();
gluLookAt(camera.vp.x + right.x, camera.vp.y +
          right.y, camera.vp.z + right.z,
          focus.x,focus.y,focus.z,
          camera.vu.x,camera.vu.y,camera.vu.z);
// draw the scene
glDrawBuffer(GL_BACK_LEFT);
glLoadIdentity();
gluLookAt(camera.vp.x - right.x, camera.vp.y -
          right.y, camera.vp.z - right.z,
          focus.x,focus.y,focus.z,
          camera.vu.x,camera.vu.y,camera.vu.z);
// draw the scene
```

5.3 3-D Rendering in OpenGL

The toe-in method while giving workable stereo pairs is not correct, it also introduces vertical parallax which is most noticeable for objects in the outer field of view. The correct method is to use what is sometimes known as the **parallel axis asymmetric frustum perspective projection**. In this case the view vectors for each camera remain parallel and a `glFrustum()` is used to describe the perspective projection.



5.3 3-D Rendering in OpenGL

In OpenGL this can be done using something like this:

```
ratio = camera.screenwidth /
   (double)camera.screenheight;
/* convert degree to radians */
radians = DTOR * camera.aperture / 2;
wd2 = near * tan(radians);
ndfl = near / camera.focallength;
/* Derive the two eye positions */
CROSSPROD(camera.vd,camera.vu,r);
Normalise(&r);
r.x *= camera.eyesep / 2.0;
r.y *= camera.eyesep / 2.0;
r.z *= camera.eyesep / 2.0;
```

5.3 3-D Rendering in OpenGL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
left = - ratio * wd2 - 0.5 * camera.eyesep * ndfl;
right = ratio * wd2 - 0.5 * camera.eyesep * ndfl;
top = wd2;
bottom = - wd2;
glFrustum(left,right,bottom,top,near,far);
glMatrixMode(GL_MODELVIEW);
glDrawBuffer(GL_BACK_RIGHT)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(camera.vp.x + r.x,camera.vp.y + r.y,camera.vp.z +
r.z,            camera.vp.x + r.x + camera.vd.x, camera.vp.y + r.y
+               camera.vd.y, camera.vp.z + r.z + camera.vd.z,
                    camera.vu.x,camera.vu.y,camera.vu.z);
// draw the scene
```

5.3 3-D Rendering in OpenGL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
left = - ratio * wd2 + 0.5 * camera.eyesep * ndfl;
right = ratio * wd2 + 0.5 * camera.eyesep * ndfl;
top = wd2; bottom = - wd2;
glFrustum(left,right,bottom,top,near,far);
glMatrixMode(GL_MODELVIEW);
glDrawBuffer(GL_BACK_LEFT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(camera.vp.x - r.x,camera.vp.y - r.y,camera.vp.z
          - r.z, camera.vp.x - r.x + camera.vd.x,
          camera.vp.y - r.y + camera.vd.y, camera.vp.z -
          r.z + camera.vd.z,
          camera.vu.x,camera.vu.y,camera.vu.z);
// draw the scene
```

5.3 3-D Rendering in OpenGL

Obviously, this method is more complicated to implement but yields better (more accurate) results since it better resembles reality.

More information and downloadable source code can be found on Paul Bourke’s web page:

http://local.wasp.uwa.edu.au/~pbourke/stereographics/stereogl/

5.4 Interactive techniques

Now that we know how to get data from a series of input devices, we can use it to interact with our OpenGL software. For example, we could use the mouse input to change the view onto the objects within our scene. Typical operations are zooming, panning, and rotating.

5.4 Interactive techniques

Zooming

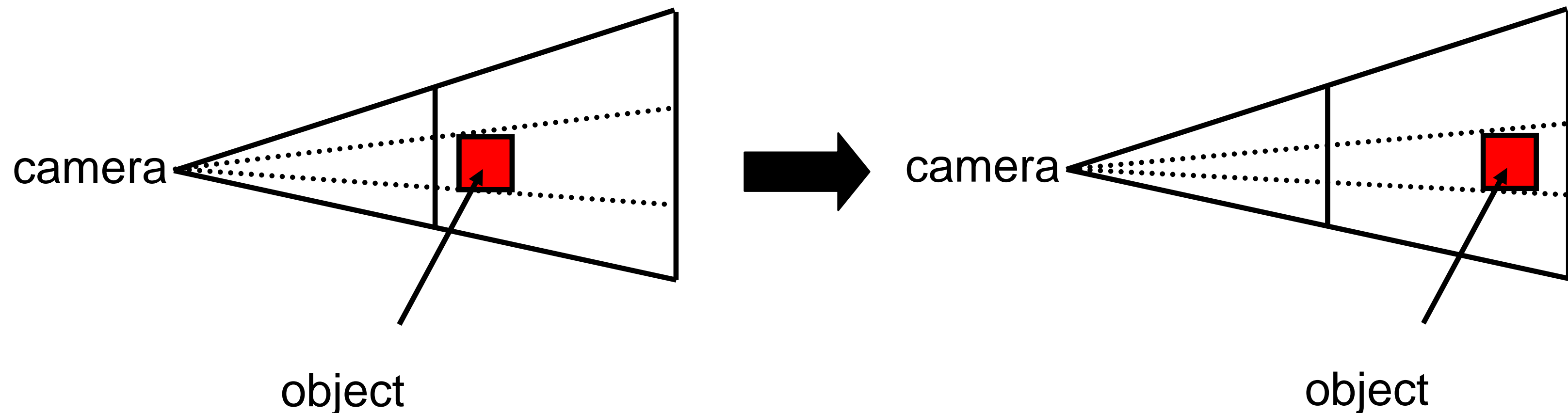
Zooming allows us to enlarge the objects our make them appear smaller on the display. This effect could be achieved in several ways.

We need to, however, differentiate between parallel and perspective projection.

5.4 Interactive techniques

Zooming for perspective projections

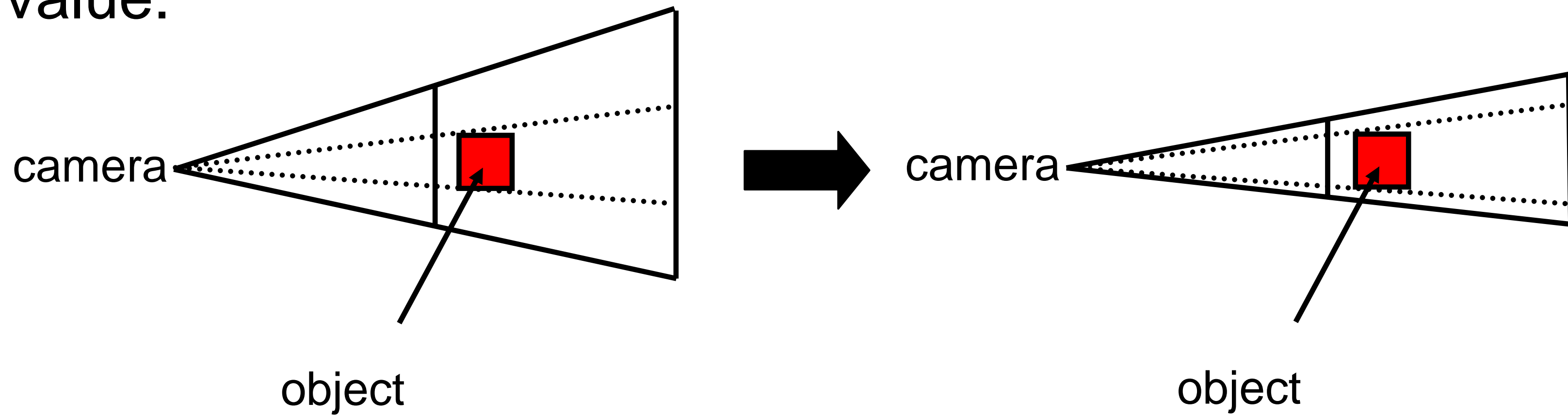
Following the camera analogy, a zoom effect can be achieved for a perspective projection by moving the camera closer to the objects or placing it farther away from them.



5.4 Interactive techniques

Zooming for perspective projections

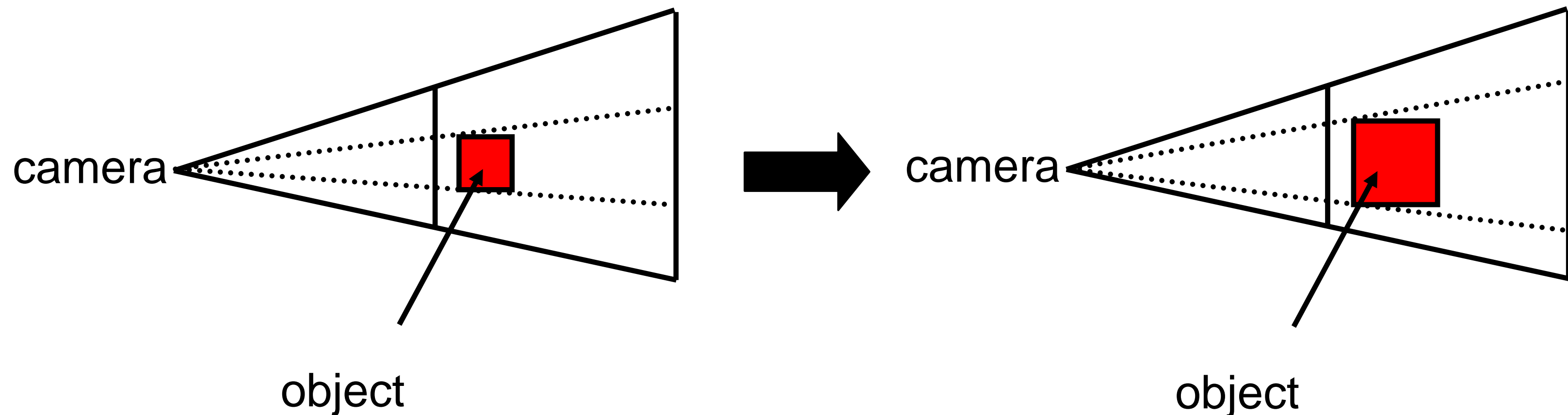
Similarly, changing the field of view can, for example, make the view volume smaller so that the objects take up more room within the view volume and hence appear larger. A zoom out effect, i.e. making the objects appear smaller can be achieved by using a larger field of view value.



5.4 Interactive techniques

Zooming for perspective projections

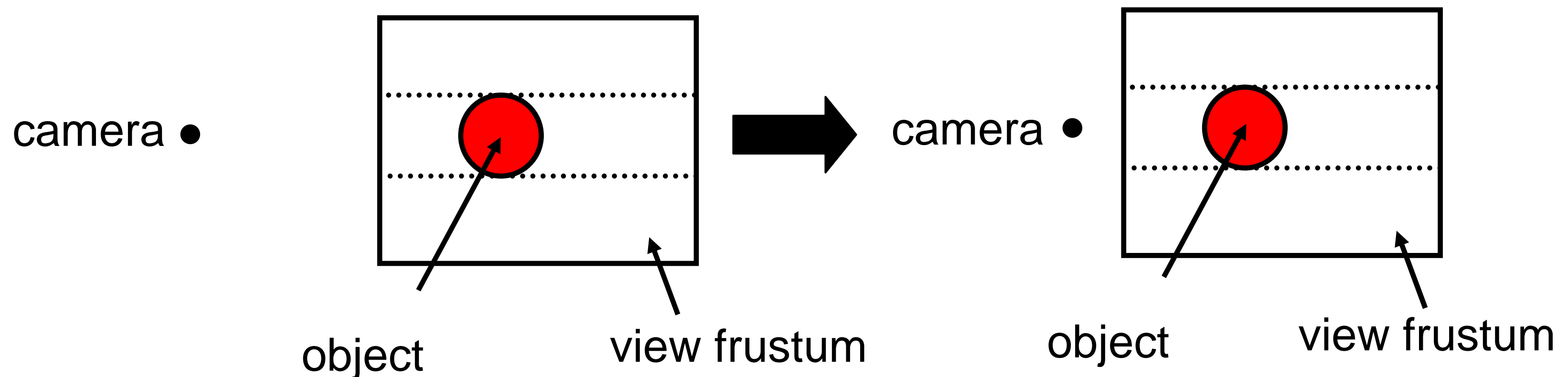
The third option for zooming, i. e. making the objects appear at a different size, is by scaling the entire scene. Using, for example, `glScale` to enlarge the entire scene results in a zoom-in effect. Scaling down all the objects appears like a zoom-out effect.



5.4 Interactive techniques

Zooming for parallel projections

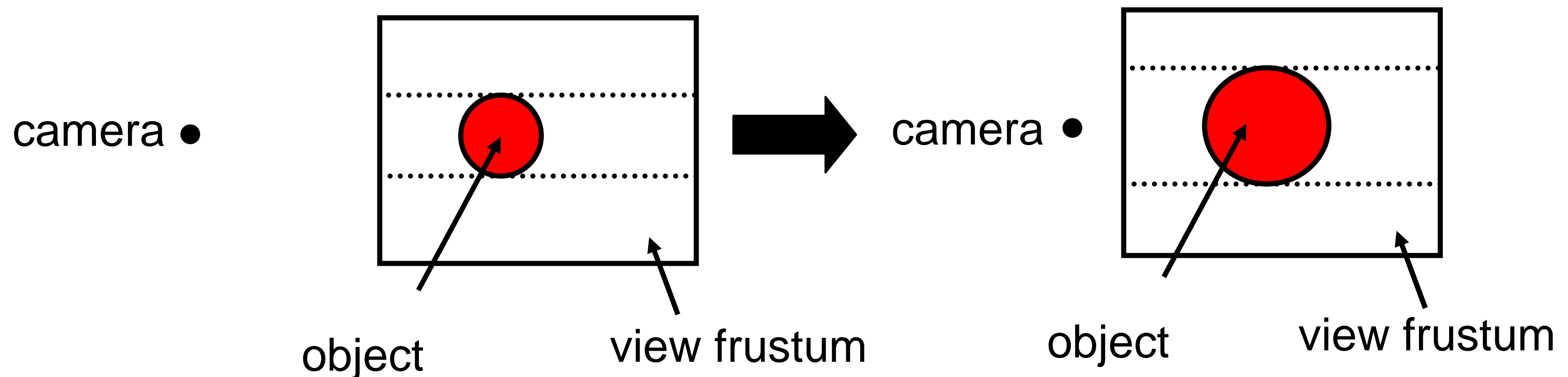
When using parallel perspective, the options for achieving a zoom effect are limited. For example, positioning the camera closer to the objects does not change the size with which they appear on the display at all.



5.4 Interactive techniques

Zooming for parallel projections

Since there is no function available that allows us to specify a field of view value (it would not make sense for a parallel projection anyway), the only option for achieving zoom is to scale the objects.



5.4 Interactive techniques

Zooming using mouse input

We could now use mouse input for zooming. One option would be to implement a slider to determine the zoom factor. Another way would be to allow the user to click in the display window and – based on the distance between the picked location and the current position of the mouse cursor – a zoom factor is determined. For example, moving the cursor up would zoom in, while moving the mouse cursor down results in a zoom-out effect.

5.4 Interactive techniques

Zooming in OpenGL

We then can modify the OpenGL source code in such a way, that a `glScale` function call is added after setting up the view matrix (for example by using `gluLookAt`). As parameters we could use a global variable for all three entries to achieve an isotropic zooming, i.e. the same change in size for all three coordinates.

This parameter is initially set to 1.0, and then potentially changed according to the mouse input.

5.4 Interactive techniques

Rubber band zoom

As another option for zoom-in, a rubber-band zoom could be used. The user selects a rectangular area of interest and the software zooms in in such a way that only this selected area is visible. To select the desired area the user can click onto the display to mark the first corner of the rectangular area. By keeping the mouse button pressed and dragging the mouse cursor along, the current location of the mouse cursor can be used as the second corner of the rectangle. Usually, a rectangle is drawn on top of the scene and constantly updated to visualize the selection.

5.4 Interactive techniques

Rubber band zoom (continued)

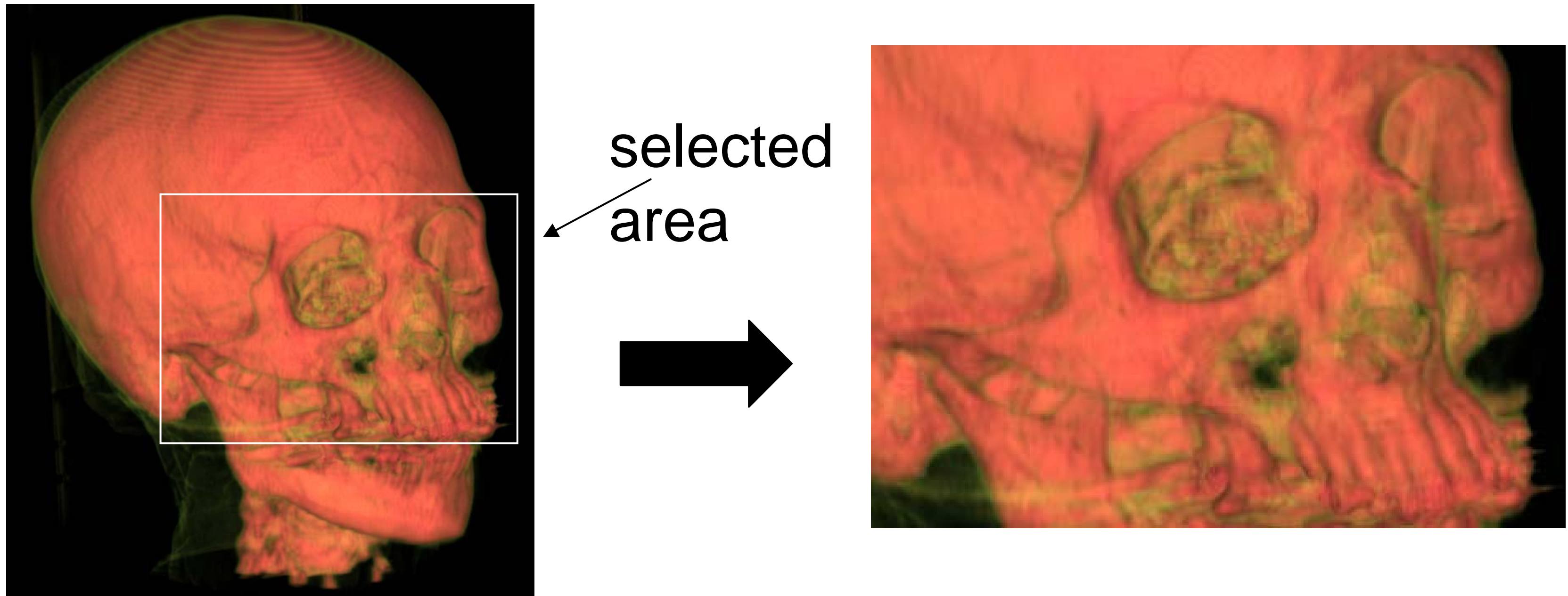
On release of the mouse button, the application zooms in based on the current rectangular area. To preserve the aspect ratio, the rectangle may be adapted accordingly.

The zoom factor can be computed as the ratio between the height of the display windows over the height of the rectangular area (in display coordinates). Alternatively, we could use the width values of the display window and the rectangle (we should use either one only to preserve the aspect ratio; the decision could be based on which value is larger: the width or height of the rectangle).

5.4 Interactive techniques

Rubber band zoom (continued)

Example.



5.4 Interactive techniques

Panning

To move the objects parallel to the viewing plane, i.e. up, down, left, or right, a panning feature can be implemented. Similar to zooming, the click/drag technique can be applied. The user clicks on the display and keeps the mouse button pressed. By moving the mouse cursor around, the objects are translated accordingly, using the vector from the original click position to the current cursor location as the displacement vector. The translation should be scaled appropriately, so that, for example, moving the mouse cursor by ten pixels results in a displacement of the objects by ten pixels in the display coordinate system.

5.4 Interactive techniques

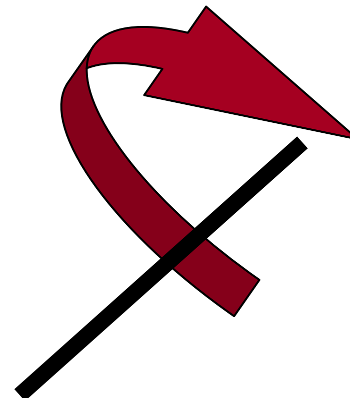
Rotating

Achieving a good technique for rotating our scene is a little more tricky. First, we need to look at the theory a little.

How many degrees of freedom are there for 3-D orientations?

3 degrees of freedom:

- direction of rotation and angle
- or 3 *Euler Angles*



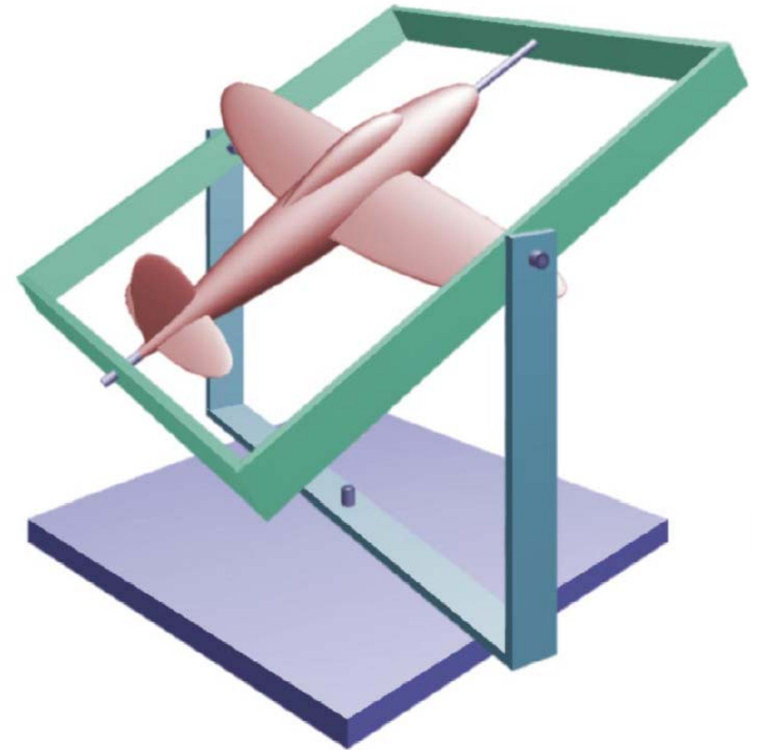
5.4 Interactive techniques

Euler Angles

An Euler angle is a rotation about a single axis.

Any orientation can be described by composing three rotations, one around each coordinate axis.

Roll, pitch, and yaw
(perfect for flight simulation)

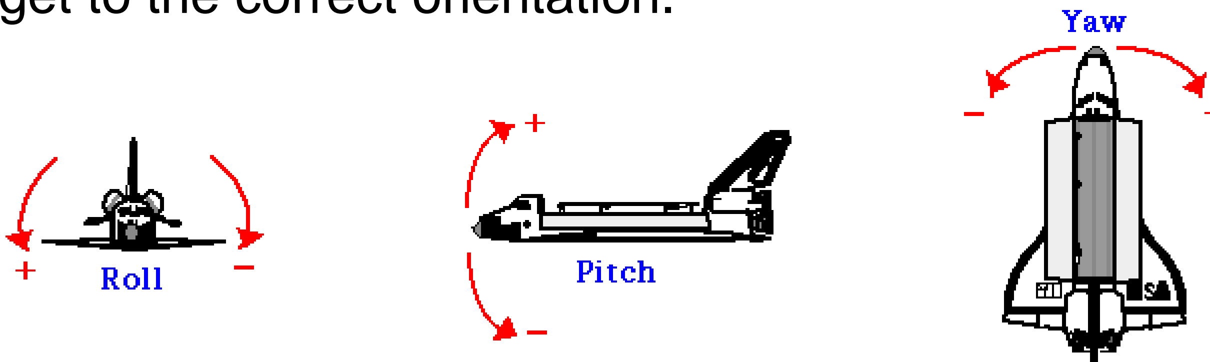


http://www.fho-emden.de/~hoffmann/gimbal09082002.pdf

5.4 Interactive techniques

Roll, pitch, and yaw

The orientation of, for example, a space shuttle in space is defined as its **attitude**. Orbiter attitudes are specified using values for **pitch, yaw, and roll**. These represent a relative rotation of the shuttle about the Y, Z, and X axes, respectively, to the desired orientation. However, the shuttle doesn't actually perform each rotation separately. It calculates one axis, called the eigen axis, to rotate about to get to the correct orientation.

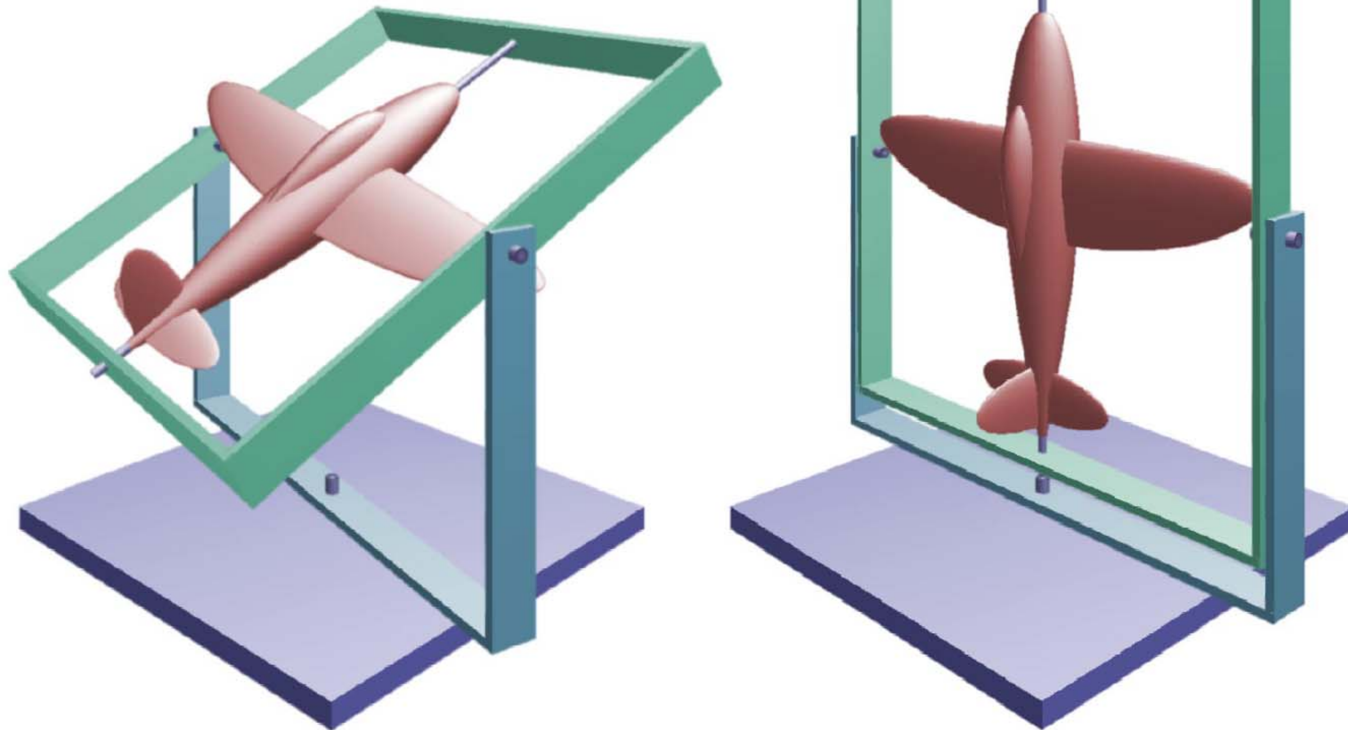


http://liftoff.msfc.nasa.gov/academy/rocket_sci/shuttle/attitude/pyr.html

5.4 Interactive techniques

Gimbal Lock

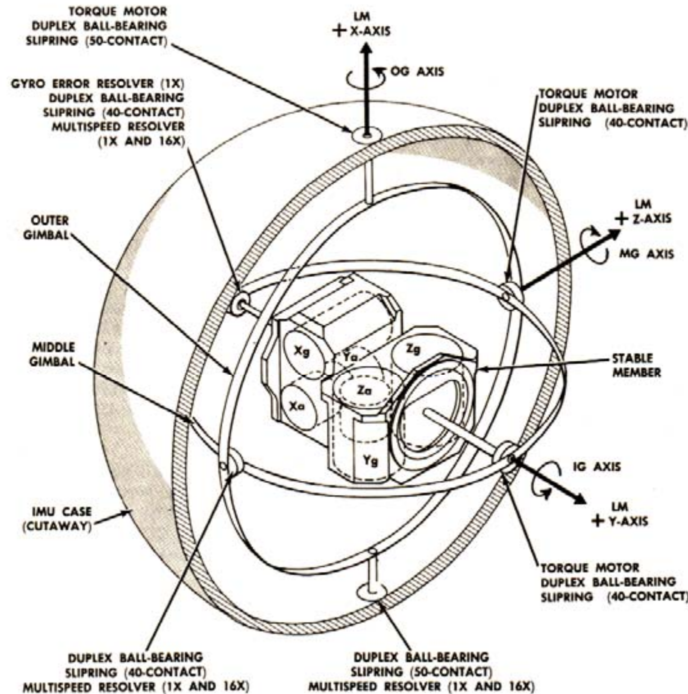
Two or more axis align resulting in a loss of rotation degrees of freedom.

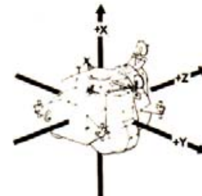


5.4 Interactive techniques

Euler Angles in the Real World

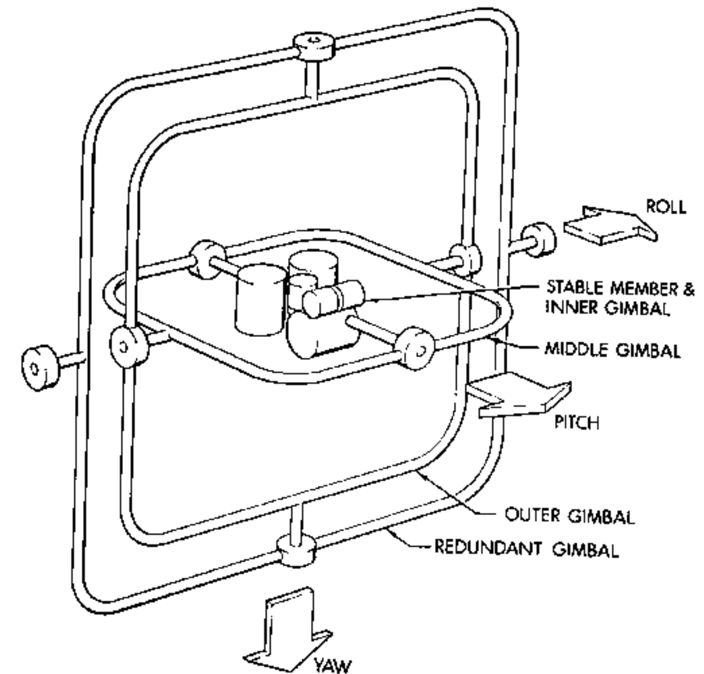
Apollo inertial measurement unit: to “prevent” lock, they added a fourth Gimbal!





Note:
Xg — X IRIG; Xa — X PIP
Yg — Y IRIG; Ya — Y PIP
Zg — Z IRIG; Za — Z PIP

300LM4-152



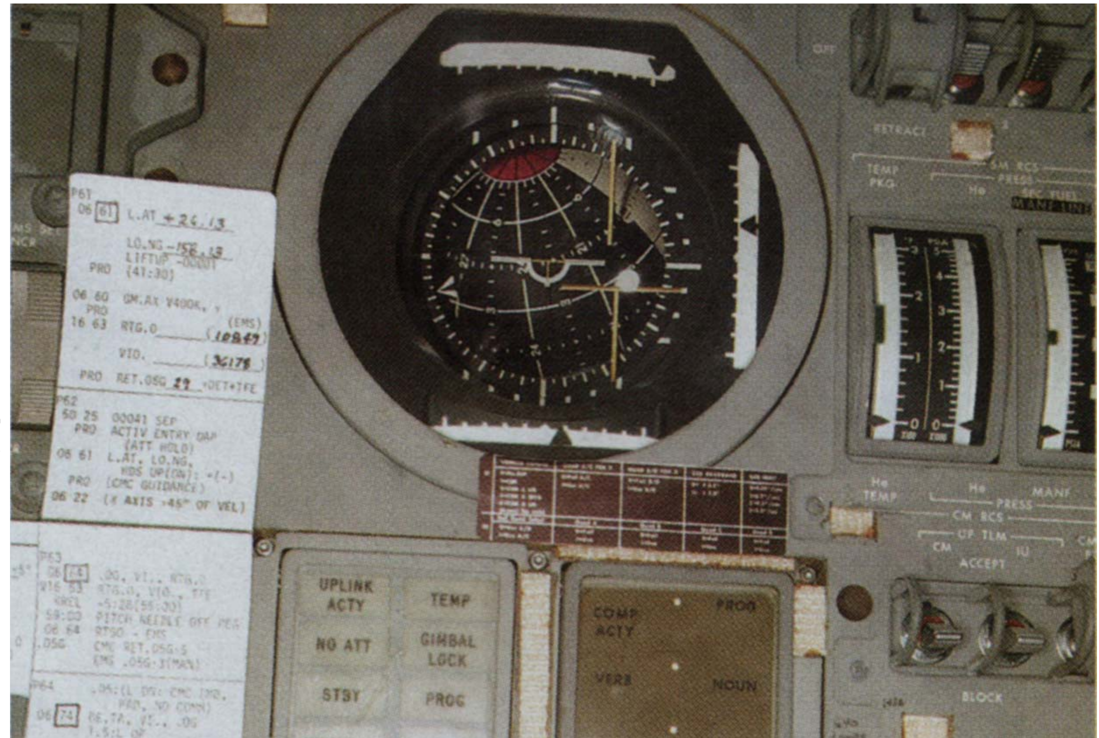
http://www.hq.nasa.gov/office/pao/History/alsj/gimbals.html

Figure 2.1-24. IMU Gimbal Assembly

5.4 Interactive techniques

Euler Angles in the Real World

Astronauts, like animators, try to avoid gimbal lock. Shown here is the Apollo 15 control panel with the “eight ball” indicating the gimbal-lock danger zone with red.



5.4 Interactive techniques

Quaternions

Invented in 1843 by Hamilton as an extension to the complex numbers

Used by computer graphics since 1985

Quaternions:

- Provide an alternative method to specify rotation
- Can avoid the gimbal lock problem
- Allow unique, smooth and continuous rotation interpolations

5.4 Interactive techniques

Mathematical Background

A quaternion is a 4-tuple of real number, which can be seen as a vector and a scalar

$Q = [q_x, q_y, q_z, q_w] = \mathbf{q}_v + q_w$, where

q_w is the real part and

$\mathbf{q}_v = iq_x + jq_y + kq_z = (q_x, q_y, q_z)$ is the imaginary part

$i*i = j*j = k*k = -1$;

$j*k= -j*k= i$; $k*i=-i*k=j$; $i*j=-j*i= k$;

All the regular vector operations (dot product, cross product, scalar product, addition, etc) applied to the imagineary part $\mathbf{q}_v$

5.4 Interactive techniques

Basic Operations

Multiplication: $QR = (\mathbf{q}_v \times \mathbf{r}_v + r_w\mathbf{q}_v + q_w\mathbf{r}_v,$

$q_w r_w - \mathbf{q}_v . \mathbf{r}_v)$

Imaginary

real

Addition: $Q+R = (\mathbf{q}_v+\mathbf{r}_v, q_w+r_w)$

Conjugate: $Q^* = (-q_v, q_w)$

Norm (magnitude) = $QQ^* = Q^*Q = q_x*q_x+q_y*q_y+q_z*q_z+q_w*q_w$

Identity $i = (\mathbf{0},1)$

Inverse $Q^{-1} = (1/ \text{Norm}(Q)) Q^*$

5.4 Interactive techniques

Polar Representation

Remember a 2D unit complex number

$\cos\theta + i\sin\theta = e^{i\theta}$

A unit quaternion Q may be written as:

$Q = (\sin\phi\, \mathbf{u_q}, \cos\phi) = \cos\phi + \sin\phi\, \mathbf{u_q}$, where $\mathbf{u_q}$ is a *unit* 3-tuple vector

We can also write this unit quaternion as:

$Q = e^{\phi \mathbf{u_q}}$

5.4 Interactive techniques

Quaternion Rotation

A rotation can be represented by a unit quarternion Q = ($\sin\phi\,\mathbf{u_q}$, $\cos\phi$)

- Given a point p = (x,y,z) -> we first convert it to a quaternion p' = $ix+jy+kz+0 = (p_v, 0)$
- Then, $Qp'Q^{-1}$ is in fact a rotation of p around $\mathbf{u_q}$ by an angle 2θ !!

5.4 Interactive techniques

How can we achieve a rotation now?

For example, a joystick could be used to determine pitch and roll (like in a flight simulator, rolling a plane results in a sideways movement).

What about the mouse?

The mouse is a 2-D input device. Thus, we can control two different degrees of freedom directly, for example roll and pitch. Similar to panning, the relative movement in x- and y-direction is converted into a rotation using an appropriate scaling factor.

5.4 Interactive techniques

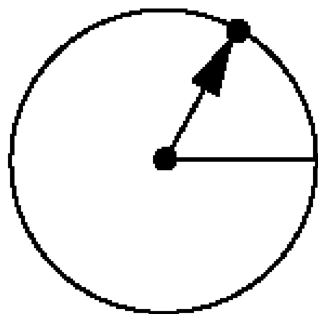
Rotational center

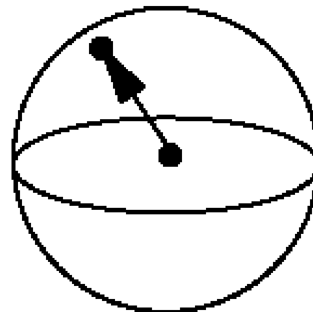
Using a fixed rotational center often results in undesirable rotations and confuses the user. Better results can be achieved by computing the rotational center as the center of the current view frustum or the center of the bounding box of all currently visible objects.

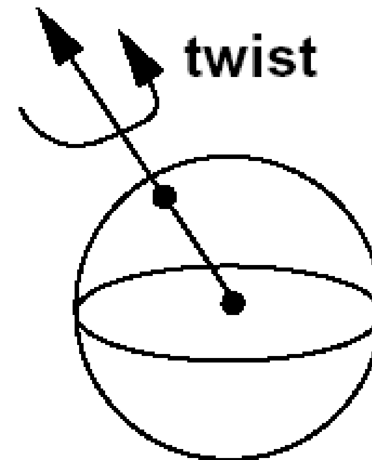
5.4 Interactive techniques

What are we missing?

Since we can only cover two degrees of freedom (DOF), we obviously lack something.



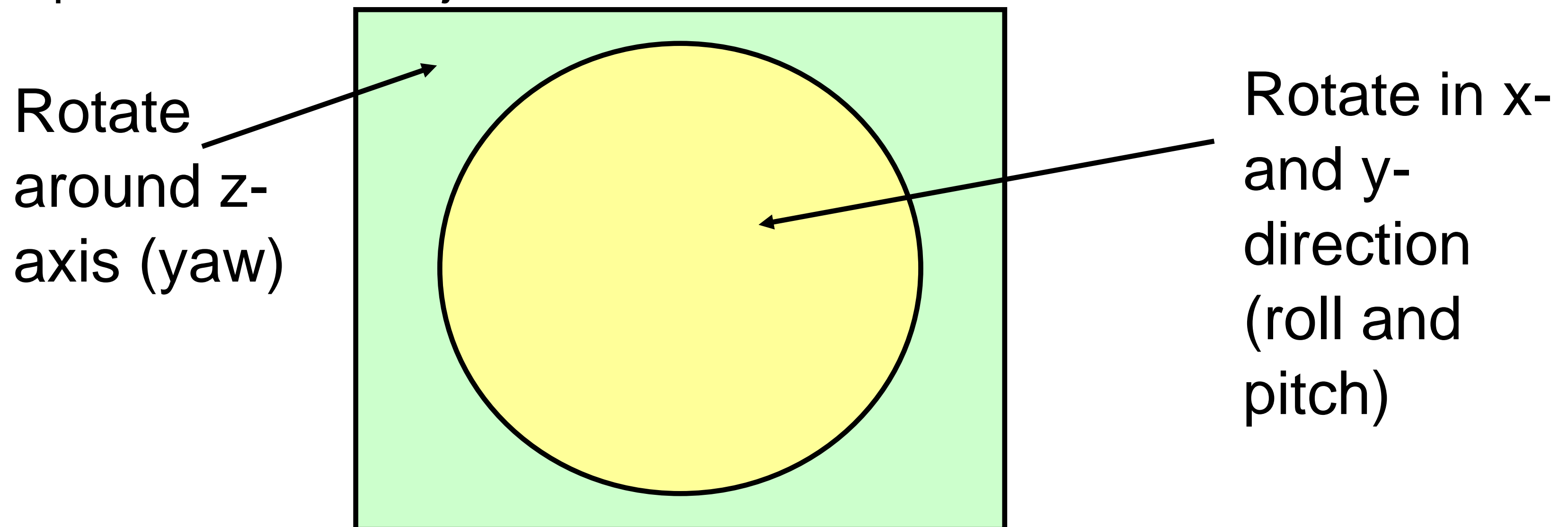




5.4 Interactive techniques

Virtual trackball

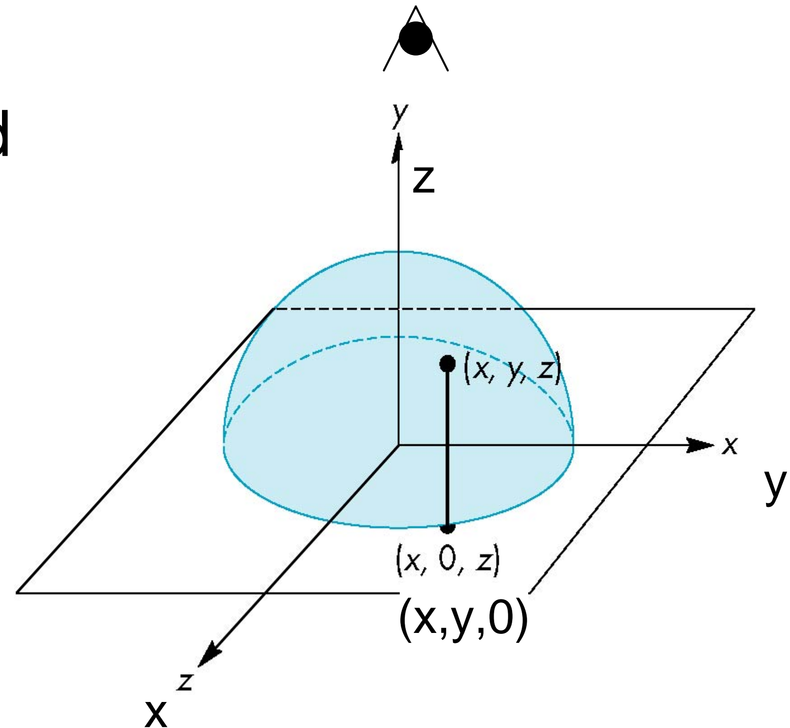
Better results for rotating the camera can be achieved by using a virtual trackball. The idea is to project mouse movement on an hypothetical sphere filling the 3D window and to apply the rotation resulting from the sphere being manipulated when the mouse button is pressed to the object.



5.4 Interactive techniques

Virtual Trackball (continued)

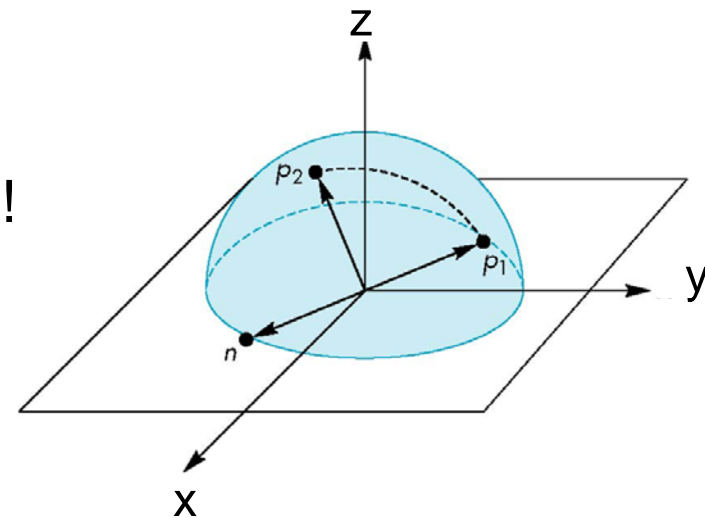
- Superimpose a hemi-sphere onto the viewport
- This hemi-sphere is projected to a circle inscribed to the viewport
- The mouse position is projected orthographically to this hemi-sphere



5.4 Interactive techniques

Virtual Trackball (continued)

- Keep track of the previous mouse position and the current position
- Calculate their projection positions p_1 and p_2 to the virtual hemi-sphere
- We then rotate the sphere from p_1 to p_2 by finding the proper rotation axis and angle
- This rotation (in world coordinate space! is then applied to the object (call the rotation before you define the camera with `gluLookAt()`)
- You should also remember to accumulate the current rotation to the previous modelview matrix



5.4 Interactive techniques

Virtual Trackball (continued)

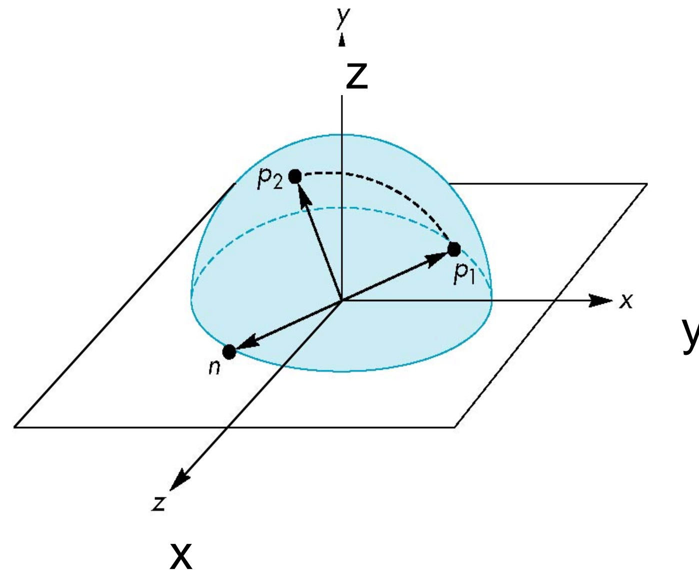
The axis of rotation is given by the normal to the plane determined by the origin, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$

$$\boldsymbol{n} = \boldsymbol{p}_1 \times \boldsymbol{p}_1$$

The angle between $\boldsymbol{p}_1$

and $\boldsymbol{p}_2$ is given by

$$|\sin\theta| = \frac{|\mathbf{n}|}{|\mathbf{p}_1||\mathbf{p}_2|}$$



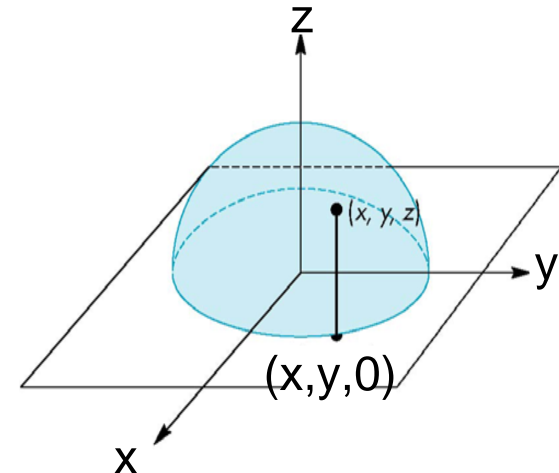
5.4 Interactive techniques

Virtual Trackball (continued)

- How to calculate p_1 and p_2?
- Assuming the mouse position is (x,y), then the sphere point P also has x and y coordinates equal to x and y
- Assume the radius of the hemi-sphere is 1. So the z coordinate of P is

$$\sqrt{1-x^2-y^2}$$

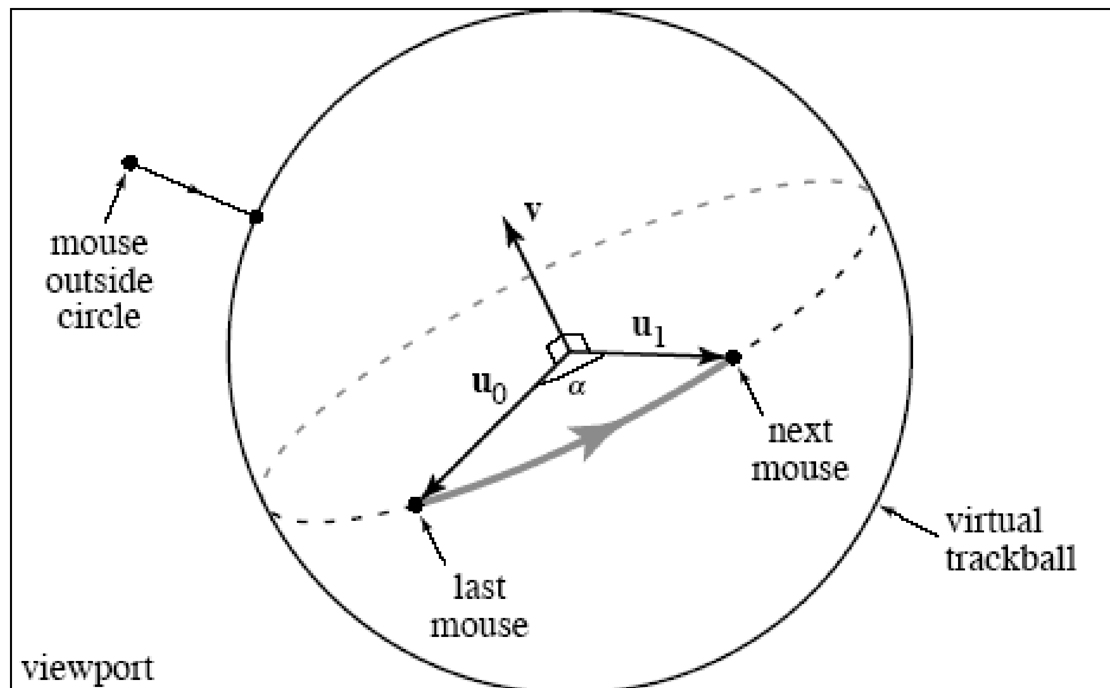
- Note: normalize viewport y extend to -1 to 1
- If a point is outside the circle, project it to the nearest point on the circle (set z to 0 and renormalize (x,y))



5.4 Interactive techniques

Virtual Trackball (continued)

Visualization of the algorithm



5.4 Interactive techniques

Example

Example from Ed Angel's OpenGL Primer

In this example, the virtual trackball is used to rotate a color cube

The code for the colorcube function is omitted

I will not cover the following code, but I am sure you will find it useful

http://www.cs.unm.edu/~angel/BOOK/PRIMER/SECOND_EDITION/PROGRAMS/

5.4 Interactive techniques

Initialization

```
#define bool int   /* if system does not support
                      bool type */
#define false 0
#define true 1
#define M_PI 3.14159 /* if not in math.h */

int    winWidth, winHeight;

float angle = 0.0, axis[3], trans[3];

bool   trackingMouse = false;
bool   redrawContinue = false;
bool    trackballMove = false;

float lastPos[3] = {0.0, 0.0, 0.0};
int curx, cury;
int startX, startY;
```

5.4 Interactive techniques

The Projection Step

```
void trackball_ptov(int x, int y, int width,
                    int height, float v[3])
{
    float d, a;
    /* project x,y onto a hemisphere centered
       within width, height , note z is up here*/
    v[0] = (2.0*x - width) / width;
    v[1] = (height - 2.0F*y) / height;
    d = sqrt(v[0]*v[0] + v[1]*v[1]);
    v[2] = cos((M_PI/2.0) * ((d < 1.0) ? d : 1.0));
    a = 1.0 / sqrt(v[0]*v[0] + v[1]*v[1] +
      v[2]*v[2]);
    v[0] *= a;    v[1] *= a;    v[2] *= a;
}
```

5.4 Interactive techniques

glutMotionFunc (1)

```
void mouseMotion(int x, int y)
{
     float curPos[3],
     dx, dy, dz;
     /* compute position on hemisphere */
     trackball_ptov(x, y, winWidth, winHeight,
                    curPos);
     if(trackingMouse)
    {
        /* compute the change in position
             on the hemisphere */
        dx = curPos[0] - lastPos[0];
        dy = curPos[1] - lastPos[1];
        dz = curPos[2] - lastPos[2];
```

5.4 Interactive techniques

glutMotionFunc (2)

```
if (dx || dy || dz)
 {
   /* compute theta and cross product */
     angle = 90.0 * sqrt(dx*dx + dy*dy + dz*dz);
     axis[0] = lastPos[1]*curPos[2] –
          lastPos[2]*curPos[1];
     axis[1] = lastPos[2]*curPos[0] –
          lastPos[0]*curPos[2];
     axis[2] = lastPos[0]*curPos[1] –
          lastPos[1]*curPos[0];
   /* update position */
     lastPos[0] = curPos[0];
     lastPos[1] = curPos[1];
     lastPos[2] = curPos[2];
   }
 }
 glutPostRedisplay();
}
```

5.4 Interactive techniques

Idle and Display Callbacks

```
void spinCube()
{
   if (redrawContinue) glutPostRedisplay();
}

void display()
{    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
 if (trackballMove)
 {
     glRotatef(angle, axis[0], axis[1], axis[2]);
 }
  colorcube();
 glutSwapBuffers();
}
```

5.4 Interactive techniques

Mouse Callback

```
void mouseButton(int button, int state, int x, int y)
{
 if(button==GLUT_RIGHT_BUTTON) exit(0);

/* holding down left button
      allows user to rotate cube */
 if(button==GLUT_LEFT_BUTTON) switch(state)
 {
    case GLUT_DOWN:
        y=winHeight-y;
        startMotion( x,y);
        break;
    case GLUT_UP:
        stopMotion( x,y);
        break;
  }
 }
```

5.4 Interactive techniques

Start Function

```
void startMotion(int x, int y)
{
  trackingMouse = true;
  redrawContinue = false;
  startX = x;
  startY = y;
  curx = x;
  cury = y;
  trackball_ptov(x, y, winWidth, winHeight,
                 lastPos);
  trackballMove=true;
}
```

5.4 Interactive techniques

Stop Function

```
void stopMotion(int x, int y)
{
    trackingMouse = false;
  /* check if position has changed */
    if (startX != x || startY != y)
      redrawContinue = true;
    else
    {
      angle = 0.0;
      redrawContinue = false;
      trackballMove = false;
    }
}
```

5.4 Interactive techniques

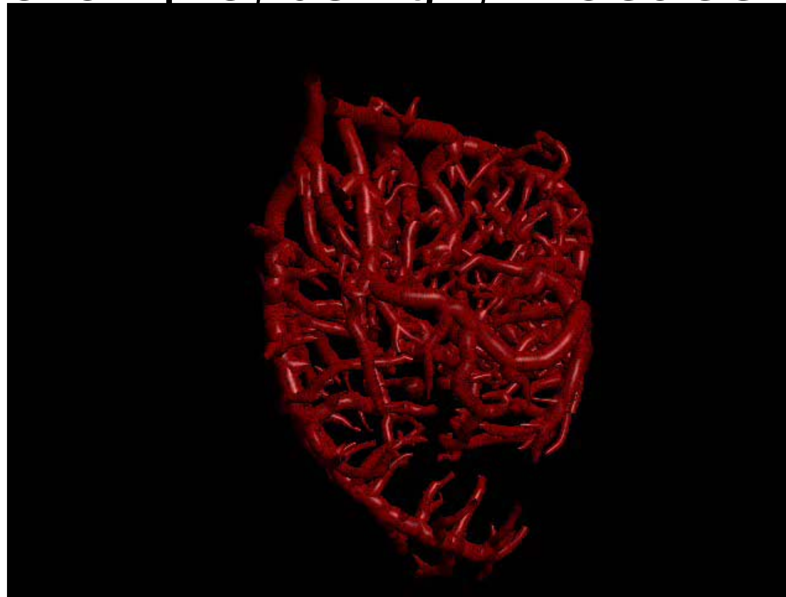
Animation

Animations are often useful showcasing your results. The human eye is more receptive for movement; hence, a moving scene is more likely to catch the eye. Also, a complex scene may be difficult to analyze in a 2-D projection. It may be easier to imagine the missing third dimension if the objects in the scene move, for example, around a center axis of the scene.

5.4 Interactive techniques

Animation

Simple animation of the scene can be achieved by moving the camera around. The simplest way would be to just rotate the camera around all our objects. We could do this by, for example, using `glRotate`.



5.4 Interactive techniques

Camera flight path

Another option is to define a flight path that the camera follows. This could be done, for example, by defining a parametric curve. The parameter can be advanced based on time so that the camera moves along the path defined by the curve. The curve should be at least continuous to avoid jumps in camera movement. Similarly, a curve can be defined for the look-at point. This gives us two curves $e(t)$ and $c(t)$ for the eye and center point. Using gluLookAt, we can let the camera move according to these curves:

```
gluLookAt (e(t), c(t), up);
```

5.4 Interactive techniques

Camera flight path (continued)

In order to get a more realistic setup and to allow zooming, a perspective projection should be used.

The parameter t can be advanced using the idle callback function using GLUT. In that function, we can check if time progressed by a specific amount. If that is the case, we can increment t by a pre-defined value.

To specify the up vector, we do not need to define a parametric curve. However, it is necessary to adapt it to the new eye and center values to avoid undesired rotation of the camera.

5.4 Interactive techniques

Camera flight path (continued)

After initializing the up-vector, we can update it after every camera movement using something like this:

```
Vector tmp, v = c(t) - e(t);
vectorProduct (tmp, v, up);
vectorProduct (up, tmp, v);
normalize (up);
```

This assumes that the change of the view direction is not too drastic. The up-vector is changed in such a way, that it is orthogonal to the view direction, i.e. the vector connecting the locations of eye and center as specified in `gluLookAt`.

5.4 Interactive techniques

Camera flight path (continued)

As an example, the camera could fly along a curve *e(t)* defined by a consecutive list of straight-line segments. The point the camera is pointing towards is chosen to be on the exact same curve, but at a slightly larger parameter value.

Then, we could set up our camera like this:

```
gluLookAt (e(t), e(t+0.05), up);
```

5.4 Interactive techniques

Camera flight path (continued)

Example