Lab „Platforms for Embedded Systems“
Chapter 06: OpenGL

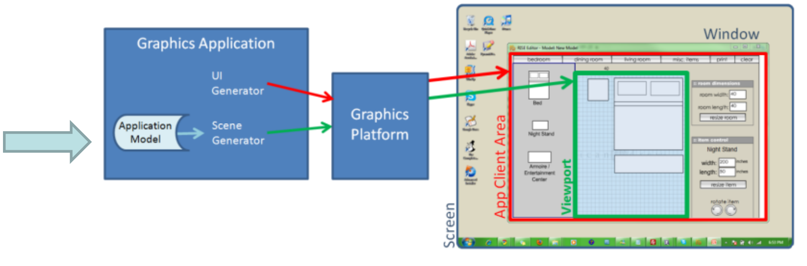
Prof. Dr. Elmar Cochlovius

06: Graphics (2) - OGL

- **Goals of this Chapter:**
  - Graphics Platforms
  - Introduction to OpenGL
  - Simple 2D Shapes
  - Transformations and 3D
  - Animations using SwapBuffer()

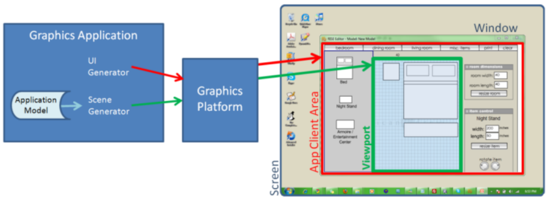
- **Overview**
  - Graphics Platforms
  - Introduction to OpenGL
  - Simple 2D-Shape
  - Transformations and 3D

- **Graphics Platforms**
  - **Graphics Applications,** which are focussed to only render *Pixels*, are not very common (mainly some simple draw demos). Instead:
  - Rather, graphics applications include an **application model** (AM), which is used to manage all data, which is displayed on the screen.
  - The AM is manipulated e.g. by the user interacting with the application (e.g. scene of a 3D game).



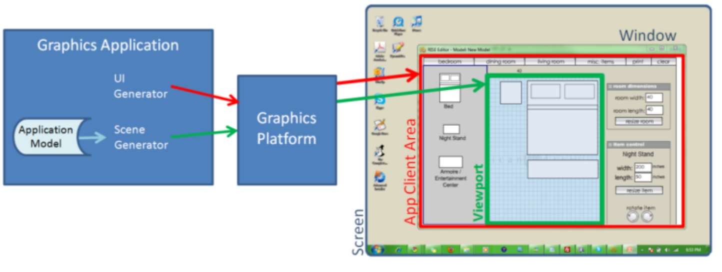
Graphics Platforms

- **Graphics platforms** are located between the AM and the display
- They collaborate closely with the **Window Manager** (WM), since the application does not have access to the complete screen.
- The WM controls, which part of the screen real estate is available to the application („client area“)
- The WM is responsible for the „eye candy“, e.g. title bar, resize handles, scrollers
- Only the client area on the screen can be accessed by the application.



Graphics Platforms

- The application model AM accesses the client area:
  - To render the GUI to receive user input / user actions
    - It is generated by the UI generator
  - To render a graphical representation of the AM to the viewport:
    - This is called a scene (2D and/or 3D)
    - It is generated by the scene generator



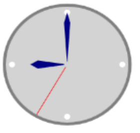
Graphics Platforms

- Early Graphics Platorms:
  - Apple QuickDraw (1984)
  - Microsoft Graphics Display interface GDI (1990)
  - Java.awt.Graphics2D
- These platforms are based on:
  - Geometric primitives („shape“) with their graphical attributes („context“), usually mode-based (modal) rather than list-oriented
  - Integer coordinates, which are directly mapped to the pixels on the screen
  - Rendering commands are not stored, i.e. **„Immediate Mode**“
  - No support for hierarchical shapes (composition)
  - No support for geometrical transformations
- In short:
  These early graphics platforms can be viewed as low-level assembler to access the display hardware

Graphics Platforms

- Problems with early graphics platforms (1/2):
  - No geometric scalability:
    - i.e. integer coordinates are directly mapped to pixel coordinates
    - Display resolution determines position and size of the objects
    - Solution: the AM requires a flexible, internal representation of objects and coordinate system → floats or fixed data types
  - Each display update requires an update of the AM:
    - Graphical operations on the objects reqire a complete(!) list of all(!) objects (and their attributes), called a **„display list“**
    - Performance Problem: Many operations are transient (e.g. pick-and-move): the AM should only be updated once when finished
    - But due to the immediate mode, the scene has to be updated for each step in-between.
    - Solution: an internal representation of the objects of a scene („**Display Model**“) is managed separately before rendering → **Retained mode**

- **Graphics Platforms**
  - Problems of early Graphics Platforms (2/2):
    - User-Interaction:
      - E.g. how to implement object picking using cursor / mouse: „Which objects belong to coordinate (x.y)?“
      - The application developer is responsible for the „**pick correlation**“, i.e. testing for point-in-bounding-box for all potential objects
      - In complex scenes, this becomes increasingly costly, since the complete object hierarchy has to be searched
        - Example:
          Clock -> Hand -> Triangle

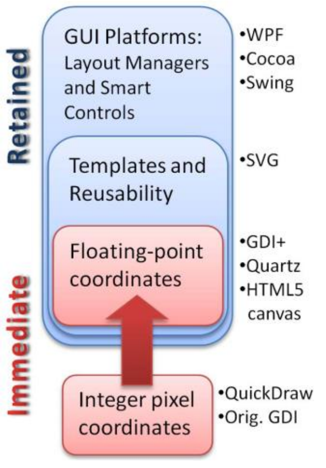


      - **Solution:** The retained mode allows the display model to implement the pick-correlation independently from the AM, since it manages an internal representation of the current scene.

06: Graphics (2) - OGL

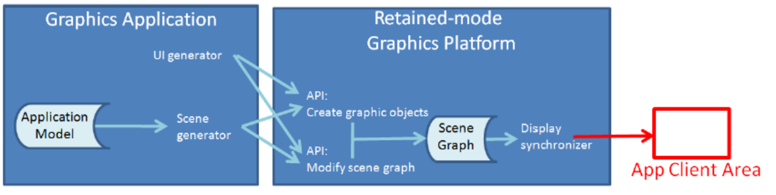
- **Graphics Platforms**
  - Current Graphics Platforms provide:
    - Coordinate system independent from physical devices (float)
      - Automatic transformation of AM coordinates into device coordinates
    - Specification of Object Hierarchies for Composition
      - Scenes are comprised as collection of hierarchic objects, based on (transformed) primitives
      - Positioning of the child objects in the coordinate system of the parent by scaling, rotation and translation
      - i.e. allows manipulating complex objects
    - Smart Objects („Widgets“, „Controls“, etc.)
      - Graphical objects with inherent behavior and reactions to user interactions
        - E.g. buttons with automatic highlighting on mouse-over events

- **Graphics Platforms**
  - Hierarchical Overviewt:



- **Graphics Platforms**
  - Immediate vs. Retained Mode (1/2):
    - Immediate Mode (OpenGL, DirectX)
      - The application model AM is responsible to keep geometric **AND** non-geometric information
      - The graphics platform does NOT manage the primitives, which comprise the current scene
      - **Rendering paradigm**: fire-and-forget

- **Graphics Platforms**
  - Immediate vs. Retained Mode (2/2):
    - Retained Mode (WPF, SVG)
      - The AM is managed in the application, the Display Model DM („scene graph“) is managed in the graphics platform
      - The DM contains all data, which define the geometry to be displayed
      - The DM is a subset of the AM and managed as a scene graph
      - Note: very „simple“ graphics apps do not need a separate AM

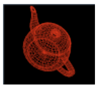


- **Overview**
  - Graphics Platforms
  - Introduction to OpenGL
  - Simple 2D-Shape
  - Transformations and 3D

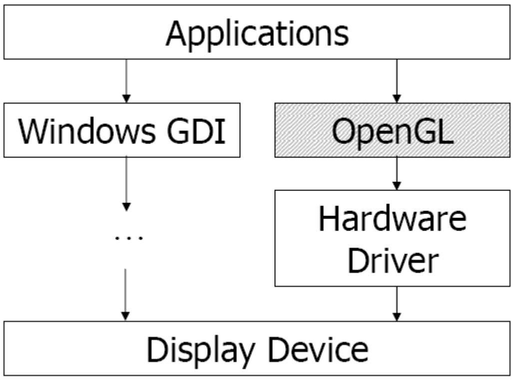
- **OpenGL: Open Graphics Language**
  - Developed by Silicon Graphics starting in 1992
  - Today managed by the **Khronos Group** non-profit consortium
  - Idea: OpenGL programs should run on different (graphics) hardware and still generate „similar“ display output
  - 1992: Graphics hardware with **„fixed-function**“ implementations
    - OpenGL function calls configure and activate the appropriate hardware pieces
  - Today: OpenGL supports fully programmable graphics HW (**GPU**)
    - GPUs are „stand-alone“ many-core parallel computers including onboard / onchip RAM
    - GPUs execute „simple“ programs („shaders“) to render a scene
    - GPUs run in parallel to the CPU
    - Developers can access and program the shader units, and do not need to wait until „next-year‘s graphics card“ with HW support
  - Demo: Nvidia showcasing „serial CPU vs. parallel GPU“

- **OpenGL**
  - **Fixed-Function API**
    - Easier to set up for rapid prototyping
    - Linear algebra etc. already implemented
    - GL utility library („GLU“) provides additional high-level utilities
  - **Programmable API**
    - Also provides the fixed function API (for backward compatibility)
    - But uses shaders in the background for implementation
    - **Note**: starting with OpenGL ES2.0+, the fixed function API is discontinued

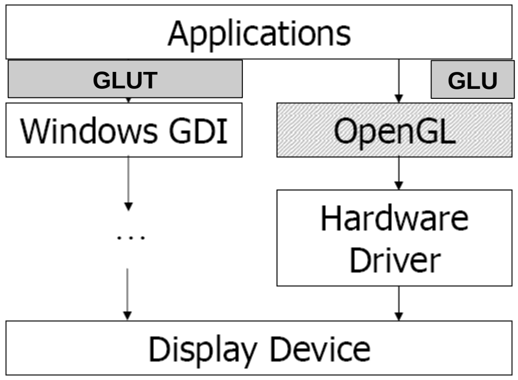
- **OpenGL**
  - **Main Functionalities of OpenGL:**
    - Rendering of points, lines and polygons
    - Matrix transformations
    - Z-Buffer and Hidden-Surface-Removal
    - Phong-Lighting
    - Gouraud-Shading
    - Texture Mapping
    - Operationen on pixels
  - **Additional Libraries:**
    - E.g. OpenGL Utility Toolkit (**GLUT**)
      - Abstraction layer for various Windows-APIs (Windows, X11, etc.)
      - High level functions for window management
      - Window- and user events
      - Some predefined 3D-Objects
      - Main-Loop for executing OpenGL programs




- **OpenGL**
  - **Hierarchy of APIs**



- **OpenGL**
  - **Hierarchy of APIs**



- **Overview**
  - Graphics Platforms
  - Introduction to OpenGL
  - Simple 2D-Shape
  - Transformations and 3D
  - Animations

OpenGL

- General structure of simple Open-GL programs:

```
#include "glut.h"   // GLUT-Lib nutzen

// Definition der Callbacks
void display();
void reshape(GLsizei, GLsizei);

// Initialisierung, Registrierung der Callbacks
// und Aufruf der Main-Loop
void main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("sample");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
}
```


OpenGL

- GLUT Functions:

```
void glutInit(int *argcp, char **argv);
```

- Initialisation of the GLUT library
- Mainly called by other GLUT functions
- http://www.opengl.org/resources/libraries/glut/spec3/node10.html

```
void glutInitDisplayMode(unsigned int mode);
```

- Specifies the display mode of upcoming windows
  - GLUT_RGB / GLUT_RGBA / GLUT_INDEX
  - GLUT_SINGLE / GLUT_DOUBLE
  - GLUT_DEPTH / GLUT_STENCIL / GLUT_ACCUM
- http://www.opengl.org/resources/libraries/glut/spec3/node12.html

- **OpenGL**
  - GLUT Functions:

```
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
```

- Initializes the window position and size
- http://www.opengl.org/resources/libraries/glut/spec3/node11.html

```
int glutCreateWindow(char *name);
```

- Creates a window according to the settings before
- http://www.opengl.org/resources/libraries/glut/spec3/node16.html#SECTION00051000000000000000

- **OpenGL**
  - GLUT Functions:

```
void glutDisplayFunc(void (*func)(void));
```

- Registers the **display Callback function** `(*func)`
- Which gets calles, whenever the content of the window has to be refreshed
- `(*func)` contains the complete functionality to render our scene graph
- Using `glutPostRedisplay(),` we can manually ask for a window refresh (= i.e. invocation of `func()`)
- http://www.opengl.org/resources/libraries/glut/spec3/node46.html

```
void glutMainLoop(void);
```

- Enters the main processing loop of GLUT (never returns)
- http://www.opengl.org/resources/libraries/glut/spec3/node14.html#376

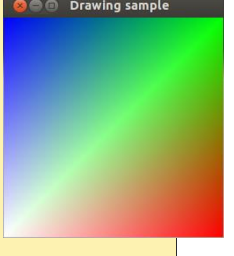
- **OpenGL**
  - GLUT Functions:

```
void glutReshapeFunc(void (*func)(int width,
int height));
```

- Registers the **reshape callback function** `(*func)`
- Which is called, whenever the window gets moved or scaled
- Inside `(func())` we should call `glViewport()`, so we can adjust the viewport if required.
- http://www.opengl.org/resources/libraries/glut/spec3/node48.html

- **OpenGL**
  - Our first OpenGL program: Rectangle with Color gradient:

```
// in the display routing,
// we finally do the rendering
void display(){
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glColor3d(1.0f, 1.0f, 1.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glColor3d(1.0f, 0.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
        glColor3d(0.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 1.0f, 0.0f);
        glColor3d(0.0f, 0.0f, 1.0f);
        glVertex3f(-1.0f, 1.0f, 0.0f);
    glEnd();
    glFlush();
}
```



- **OpenGL**
  - Now we are ready for:
    - **Exercise 15 –**
      Hello-World with OpenGL

- **Overview**
  - Graphics Platforms
  - Introduction to OpenGL
  - Simple 2D-Shape
  - Transformations and 3D
  - Animations

- **OpenGL: Next step**
  - Now we want to work on **user interaction** and simple **3-D models**
  - For this, we need some add. callback functions:

```
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
```

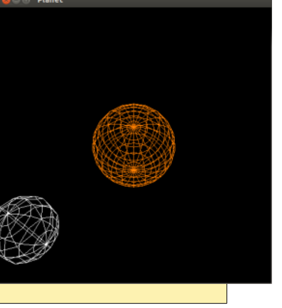
- Registers the **keyboard callback function** `(*func)`
- `*func()` will be called as a keyboard callback of the current window
- http://www.opengl.org/resources/libraries/glut/spec3/node49.html

```
void glutIdleFunc(void (*func)(void));
```

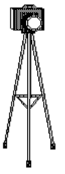
- This is the globaler **idle callback**
- http://www.opengl.org/resources/libraries/glut/spec3/node63.html

- **OpenGL**
  - Now our 2nd OpenGL program – **Rotating Planets:**

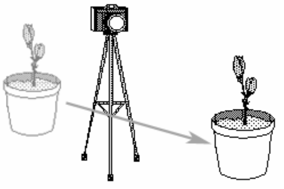
```
// GLUT library
#include "glut.h"

// Globals
static GLfloat year=0.0f, day=0.0f;

// Callbacks
void display();
void reshape(GLsizei , GLsizei );
void idle();
void keyboard(unsigned char ,
              int, int);
```



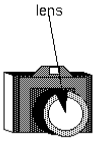
- **OpenGL: 3D Transformations**
  - To better understand the **3D functionality** of OpenGL, we use the well-known **„analogy of a camera“:**
  - **Step 1: Viewing Transformation:**
    - Positioning of the camera



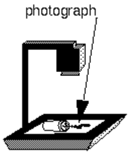
  - **Step 2: Modeling Transformation:**
    - Adjusting the scene, which we will take a picture of



- **OpenGL: 3D Transformations**
  - „Analogy of a camera“ (2/2)
  - **Step 3: Projektion Transformation:**
    - Selection and Adjusting the appropriate lens (wide angle, standard or tele lens)



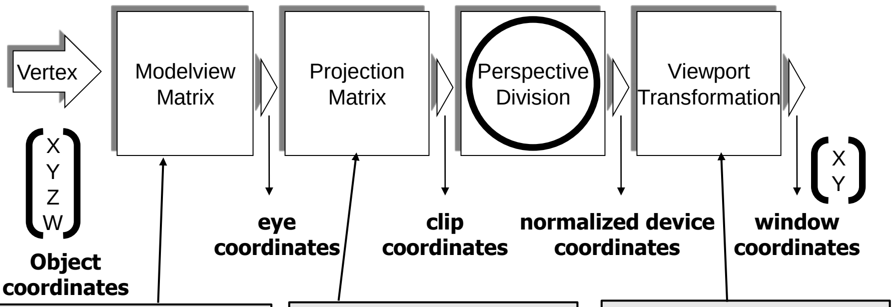
  - **Step 4: Viewport Transformation:**
    - Define size and position of the final image



16

06: Graphics (2) - OGL

- **OpenGL**
  - And now the same using OpenGL:



Step 1 und 2:
1: set Matrix-Mode to `GL_MODELVIEW` and call `glLoadIdentity()`
2: Call `gluLookAt()`
3: add. model transforms, if required

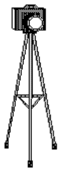
Step 3:
1: Set Matrix-Mode to `GL_PROJECTION` and call `glLoadIdentity()`
2: Depending on projection, call `gluPerspective()` or `gluOrtho2D()`

Step 4:
1: call `glViewport()`

06: Graphics (2) - OGL

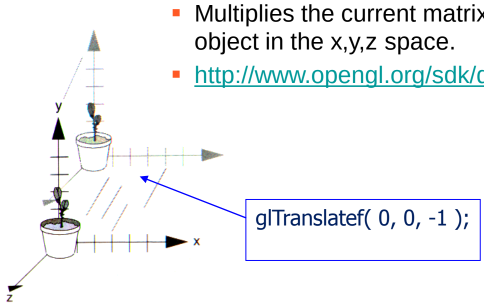
- **OpenGL**
  - And now step-by-step with add. parameters:
  - Step 1 - Viewing Transformation:

```
gluLookAt( GLdouble eyex, GLdouble eyey,
GLdouble eyez,    GLdouble centerx, GLdouble
centery, GLdouble centerz, GLdouble upx,
GLdouble upy, GLdoubpe upz );
```

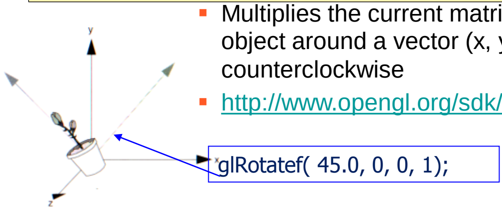


- **`eyex,y,z:`** position of the camera (default: 0,0,0)
- **`centerx,y,z:`** orientation of the camera reference point (default: negative Z axis)
- **`upx,y,z:`** the up-vector of the camera (default: positive Y axis)
- http://www.opengl.org/sdk/docs/man2/xhtml/gluLookAt.xml

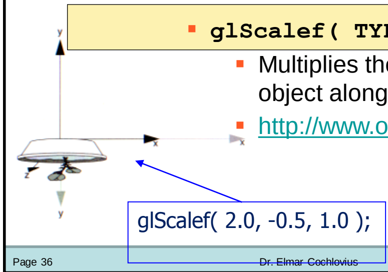
- **OpenGL**
  - Step 2 - Modeling Transformation (1/2):
    - Executes scaling, translation and rotation or a combination thereof
    - Note: In OpenGL we can combine the View and the Model transforms into a single matrix („**Modelview Matrix**")
    - **`glTranslatef( TYPE x,TYPE y,TYPE z );`**
      - Multiplies the current matrix with a matrix, which moves teh object in the x,y,z space.
      - http://www.opengl.org/sdk/docs/man2/xhtml/glTranslate.xml



- **OpenGL**
  - Step 2 - Modeling Transformation (2/2):
    - **`glRotatef( TYPE angle, TYPE x,TYPE y,TYPE z );`**
      - Multiplies the current matrix with a matrix, which rotates the object around a vector (x, y, z) <angle> degrees counterclockwise
      - http://www.opengl.org/sdk/docs/man2/xhtml/glRotate.xml

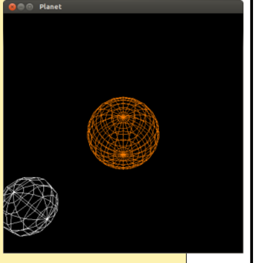


- **`glScalef( TYPE x,TYPE y,TYPE z );`**
  - Multiplies the current matrix with a matrix, which scales the object along the axes with the factors of x,y,z.
  - http://www.opengl.org/sdk/docs/man2/xhtml/glScale.xml



- **OpenGL**
  - How can we rotate the planets in the rotating planet program? (1/3)

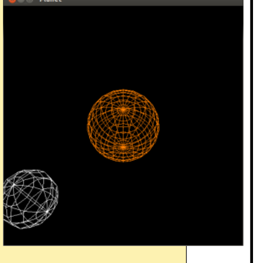
```
void display()
{   // clear the buffer
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glColor3f(1.0, 1.0, 1.0);
    glutWireSphere(1.0, 20, 16);//Sun
    glPushMatrix();
        glRotatef(year, 0.0, 1.0, 0.0);
        glTranslatef(3.0, 0.0, 0.0);
        glRotatef(day, 0.0, 1.0, 0.0);
        glutWireSphere(0.5, 10, 8);// the Planet
    glPopMatrix();
    // swap the front and back buffer
    glutSwapBuffers();
}
```



06: Graphics (2) - OGL

- **OpenGL**
  - How can we rotate the planets in the rotating planet program? (2/3)

```
// GLUT idle function
// here we modify the globals for
// the display() routine
void idle()
{
        day += 10.0;
        if(day > 360.0)
            day -= 360.0;
        year += 1.0;
        if(year > 360.0)
            year -= 360.0;
        // recall GL_display() function
        glutPostRedisplay();
}
```



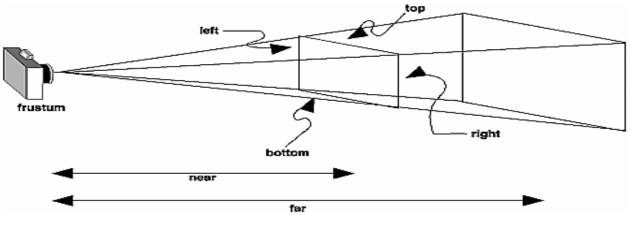
- **OpenGL**
  - How can we rotate the planets in the rotating planet program? (3/3)

```
// GLUT keyboard function
void keyboard(unsigned char key, int x, int y)
{   switch(key)
    {   case 'd': day += 10.0;
            if(day > 360.0)
                day-= 360.0;
            glutPostRedisplay();break;
        case 'y': year += 1.0;
            if(year > 360.0)
                year-= 360.0;
            glutPostRedisplay();break;
        case 'a': // assign idle function
            glutIdleFunc(idle);break;
        case 'A': glutIdleFunc(0); break;
        case 27:  exit(0);
    }
}
```


- **OpenGL**
  - Step 3 – Projection Transformation (1/3):

```
glFrustum( GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top, GLdouble near,
GLdouble far );
```

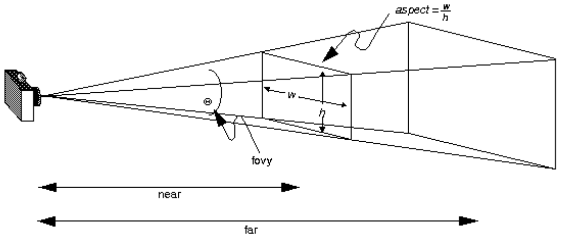
- Defines the „field of view“, i.e. the „volume of visibility“ („Frustum“)
- http://www.opengl.org/sdk/docs/man2/xhtml/glFrustum.xml

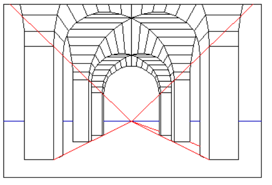


- **OpenGL**
  - Step 3 – Projection transformation (2/3):

```
gluPerspective( GLdouble fovy, GLdouble aspect,
GLdouble near, GLdouble far );
```

- Defines the projection using a central perspective based on focal length / aperture angle of the camera lens



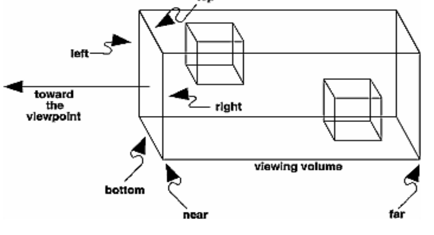


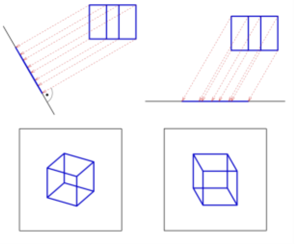
06: Graphics (2) - OGL

- **OpenGL**
  - Step 3 – Projektion transformation (3/3):

```
glOrtho( GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top, GLdouble near,
GLdouble far );
```

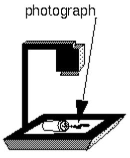
- Alternatively, defines a parallel perspective (used for „orthographic“ projections





- **OpenGL**
  - Viewport Transformation:

```
void glViewport( GLint x, GLint y, GLsizei
width, GLsizei height );
```



- Translates the finale image to a region inside the window
- **`x,y`**: left, bottom corner in pixels (default: 0,0)
- **`width, height`**: height, width of the viewport (default: dimensions of the complete window)

- **OpenGL**
  - Matrix Manipulations:

```
void glMatrixMode( GLenum mode );
```

- Sets one of three matrix modes:
  - GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE
- Each matrix mode has a separate stack of matrices

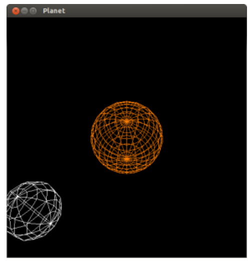
```
void glLoadIdentity();
```

- Current matrix gets initialized with the 4x4 identity matrix

```
void glPushMatrix(); void glPopMatrix();
```

- Stack operations used by hierarchical structures

- **OpenGL**
  - Now we can work on:
    - **Exercise 16 –**
      Simple 3D projections using OpenGL



06: Graphics (2) - OGL

- **OpenGL**
  - And now we turn to:
    - **Exercise 17 –**
      OpenGL Gears

- **Summary:**
  - Early and current graphics platforms: Immediate vs. Retained Mode
  - Short Introduction to OpenGL: Direct-Function API vs. Programmable API
  - Simple 2D shapes using glutInit, glutMainLoop and callbacks
  - Transformations and 3D: Camera analogy and matrix operations
  - Animations using SwapBuffer()

- **References and add. Information:**

  - Wikipedia: perspektive, orthogonal projection
  - Graphics platforms: Andries van Dam, Introduction to Computer Graphcis, http://cs.brown.edu/courses/csci1230/
  - 2D, 3D: Marco Schaerf, Computer-Graphics: https://sites.google.com/site/marcoschaerfcomputergraphics/course-notes
  - E. Angel: OpenGL – A Primer, 2nd ed., Pearson, Addison Wesley