Wrocław University of Technology

Jerzy Sas

Practical Advanced Computer Graphics - laboratory assignments

Advanced Computer Graphics

Wrocław 2010

Wrocław 2010

Reviewer:

Elżbieta Hudyma

Project Office

ul. M. Smoluchowskiego 25, room 407
50-372 Wrocław, Poland
Phone: +48 71 320 43 77
Email: studia@pwr.wroc.pl
Website: www.studia.pwr.wroc.pl

Table of contents:

Preface

Computer graphic nowadays is one of the most important elements of a software engineer's education. Computer graphics (CG for short) is being effectively applied in virtually all domains of engineering, science and in everyday life. Visualization of CAD projects in mechanical engineering, architectural design, lighting design and analysis, presentation of molecules in chemistry, 3D visualization in medical imaging, desktop publishing, advertising, computer games and entertainment are just a few examples of CG applications. The extending market of CG applications is a driving force for research and development both in software and hardware that support CG. The most important subarea of CG is related to its application to the rendering of 3D scenes, where the aim is to create realistic looking images of the virtual world.

Pure CG is a domain in computer science dealing with creation of images by computers, where the input data represent the contents of the image to be created but itself are not an image. Therefore, the pure CG deals with the process of image synthesis from data not being the image. CG is not the only domain in computer science that deals with images. *Image processing* is the domain dealing with processes, where the input image is transformed into another one, essentially consisting of the same contents. The aim of image processing is to improve the input image, to rearrange its layout (e.g. by geometrical transformations) or to extract some features of the image, necessary for further processing. *Image analysis* and *computer vision* are other domains that deal with images represented in computers. Here, the aim is to extract structural information about the objects depicted on the image (e.g. to localize human faces in the image, to identify classes of things visible on the image and to find their relative positions etc.). In this hand-book we will deal only with pure CG focused on image synthesis.

The aim of the course in "Advanced Computer Graphics" is to provide the knowledge and to develop practical skills necessary to build advanced 3D CG applications. This hand-book presents the series of laboratory practical exercises that lead a student through the topics related to essential CG programming, from very basic assignments that explain fundamentals of 2D graphics to advanced photorealistic rendering implementation with backward ray tracing technique and sophisticated procedural texturing. No initial knowledge in CG is required, but students are expected to have some experience in Java and C++ programming in popular integrated development environments like MSVC or Eclipse for Java.

Java programming language is recommended as the implementation platform for most of the simple assignments described here. This is because (contrary to other languages like C or C++) Java provides support for graphics programming in the language standard, including basic raster and vector graphics operations, advanced apparatus for easy implementation of advanced graphical user interface for interactive applications and convenient components for modeling and rendering of 3D scenes. Unfortunately, Java does not provide convenient interface to the popular 3D library, OpenGL. Although Java native interface to OpenGL is available, it is not as easy to use as Java3D. Also Java efficiency may be an issue as far as intensive computations are executed in the process of image rendering. Therefore, in the case of advanced assignments related to OpenGL and ray tracing, using C++ seems to be more appropriate.

The following assignments are suggested for implementation:

1. Procedural creation of raster graphics patterns
2. Application of vector and raster drawing services in on-screen graphics
3. Simple animation of vector images
4. Building simple graphical user interface with Swing - simple interactive line drawing program
5. Homogenous transformations in 2D
6. Bilinear interpolation and Gouraud shading of 2D triangles
7. Simple rendering with Phong lighting model
8. Basic elements of boundary-represented geometry rendering
9. 3D modeling of surface patches and solids
10. Software implementation of visible surfaces determination with Z-buffer
11. 3D rendering with OpenGL
12. Simple ray tracing program

Each laboratory assignment described in the hand-book is preceded by a short theoretical introduction. In the assignments explaining basic Java CG programming, the complete working code examples are presented and precisely commented, line by line. The students are encouraged to use these examples as a baseline for their own extensions necessary to satisfy the assignment specification. Then detailed assignment requirement specification is presented. The proposed assignments differ in difficulty level and labor intensity. The approximate estimation of the labor intensity necessary to completely understand related topics, implement a program and (where required) conduct tests and experiments varies from 2-3 hours for initial assignments to 15-20 work hours in case of ray tracing implementation. Assuming that a student is able to spend 3-4 hours per week, simple assignments (1-6) should be implemented in a one week cycle. Each of assignments 7-10 should be implemented in two-weeks. The reasonable time span for the remaining assignments seems to be 3 weeks. Not all assignments need to be implemented by students. The teacher should select appropriate assignments for implementation, depending on the student's actual level of expertise. However it is strongly recommended to preserve the order of assignments.

Assignment 1
Procedural creation of raster graphics patterns

Aim

The aim of this assignment is to learn how to use software components than encapsulate data structures representing raster images and typical raster image operations (loading/storing from/to a graphic file, initializing an empty image, accessing individual pixels, querying the resolution of the image) and how to create arbitrary patterns on pixel-by-pixel basis.

Theoretical fundamentals

Image representation in the computer memory

An image is typically a rectangle on the 2D plane. Each point in the image area is described by its visual properties that determine human's impressions when looking at the fragment of the image. We typically distinguish *color* and *monochromatic* images. The point on the surface of a color image is specified by the set of attributes that represent the spectral distribution of light energy emitted from the image fragment in the observer direction. This distribution can be represented in various coordinates system. In computer graphics the most common method of color representation is the representation in RGB color space. The spectrum of wavelengths is modeled by just three numbers representing the cumulative light energy in sub-bands corresponding to the three basic colors: red (R), green (G) and blue (B). *RGB* color space belongs to the family of physical color models because it relatively strictly corresponds to the physical nature of the light and its perception in the human eye. Tristimulus color vision theory by Thomas Young (1801) later extended by Herman von Helmholtz (1850) assumes that there are three types of receptors in the human eye which sensitivity maxima corresponding to *RGB* colors. Therefore, each visible color can be represented by the combinations of just three basic colors. In a image processing, RGB color space is not as convenient as in pure CG, therefore other color models are used that more closely correspond to the way, humans describe their visual impressions.

Here we will use *RGB* color space where visual attributes constitute a three element vector *(R,G,B),* where each component represents the corresponding basic color intensity. The range of element values of RGB vector are determined so as to assure a sufficient accuracy in the digital representation of continuous physical value and to make possible a efficient storage of RGB vectors. Typically, $R,G,B \in \{0,...255\}$ and therefore each color component can be stored in the single byte. Experiments proved that limiting color intensity level count to 256 makes possible to preserve the impression of color continuity when perceived by humans. This is because the human eye is not able to differentiate color combinations which component intensities differences are below certain threshold.

In case of monochromatic images, the uniform color is defined for the whole image. Each image point is characterized by just single attribute determining the point *brightness* (sometimes also called *lightness*) . The actual color of the image fragment can be obtained by modulating the uniform color with the point brightness. For the sake of the efficient storage and the easy access, also this attribute is typically represented by numbers form the

set $\{0,...255\}$. The particular case of the monochromatic image is a *grayscale* image. The common color is in this case neutral white represented by $R = G = B = 255$. In result, each image point is characterized by *RGB* vector where all components are equal, what corresponds to a level of gray.

The real world image (as e.g. the image obtained by an analog photo camera) is the rectangle defined in the continuous R^2 space. In the case of an image representation in digital computers, the continuous space needs to be discretized. Discretization consists in dividing the image rectangle into regular grid of picture elements called *pixels* (this acronym comes from **Pi**cture **El**ements). The pixels are arranged into the rectangular array consisting of specified number of columns and rows. The position of the pixel is specified by the pair of indexes:

$$(i, j) : i \in \{0,..., y_{res} - 1\},\ j \in \{0,..., x_{res} - 1\}$$

where:

i - row index,
j - column index,
x_{res} - image horizontal resolution (the number of columns in the pixel array)
y_{res} - image vertical resolution (the number of rows in the pixel array).

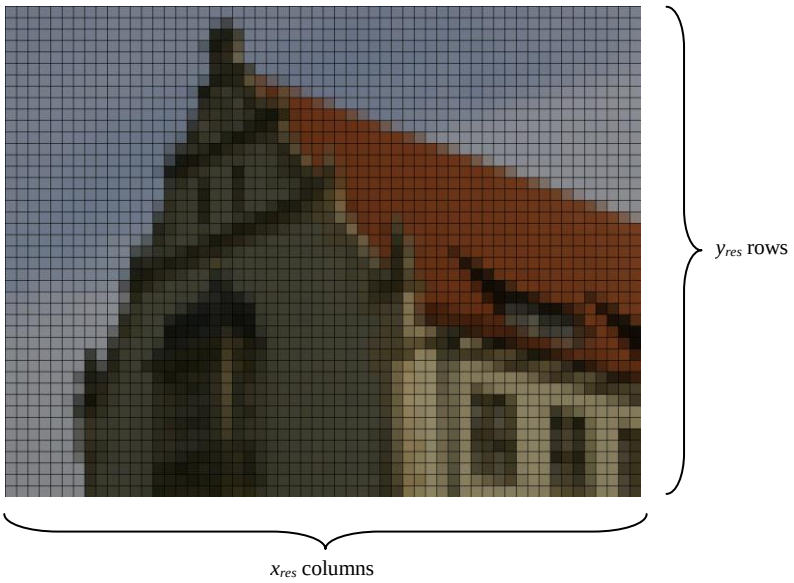


Fig. 1.1. Example of a raster image with visible pixel raster

The image represented by the array of pixels will be called *raster image*. If the raster image is created by setting attributes of each pixel individually (e.g. as the result of calculations performed in a pixel-by-pixel manner) then the procedure of image creation will be called *raster graphics* procedure. The raster image content consists of explicit specification of all image pixels attributes. The pixel array can be however filled by applying a sequence of simple drawing operations like e.g.:

– drawing a line segment,
– drawing a rectangle,

– drawing a circle or an arch,
– filling a circle or a rectangle with the specified color.

In this case, the image contents are defined by just specifying either relatively short sequence of drawing operations or by specifying the geometrical primitives like line segments or 2D figures being the result of drawing operations. The result can be the image as presented on Fig. 2.

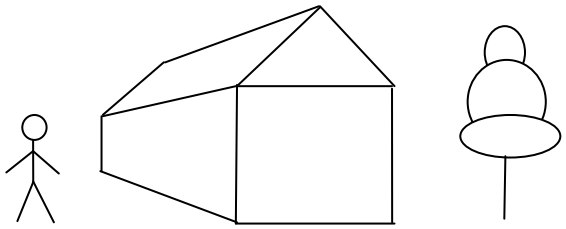


Fig. 1.2. Example of a vector image

The image contents can be specified by:
– specifying line segment end coordinates for 17 elements (68 numbers),
– specifying centers and axes lengths of 4 elements (16 numbers).

Even complex images can be specified in this way by using the amount of data incomparably smaller than in the case of raster graphics images. The image which contents are defined by specifying the attributes of simple graphical primitives will be called *vector image*. The method of a vector image creation consisting in applying a sequence of simple drawing operations will be called *vector graphics*. Despite its simplicity, vector images are quite useful in representing maps, blueprints, CAD drawings. It is difficult to visualize details on vector images, so they are in principle not appropriate in a realistic visualization, where raster images are recommended. However, it is possible to obtain near-photorealistic vector images by covering the photo image with the set of appropriately subdivided 2D polygons and by applying appropriate shading to them. Impressive results can be found at http://digital-artist-toolbox.com.

It should be pointed out that most of CG display devices follow raster graphics paradigm, i.e. display surfaces are covered by the array of pixels that can be controlled independently (e.g. LCD and CRT monitors, DLP projectors, most printers). Vector graphic formats are used as the method of image storage that is more compact as in the case of raster images. Drawing a vector image on the raster device consists in applying drawing operations defined by the vector image contents.

Predefined API components for raster graphics processing and synthesis

Most development environments provide software components that support both raster and vector graphics. In this exercise we will learn how to create raster images in pixel-by-pixel manner that provides the most flexible way of a raster image creation. The flexibility consists in the ability to compute and set each pixel color independently. In this way, any

imaginable pattern can be created and stored in the raster image. Typically, in order to create an image the programmer needs the following basic image utilities:

- loading an image from the graphic file,
- querying loaded image resolution,
- initializing an empty image,
- getting/setting individual pixel attributes,
- setting attributes of all pixels in the selected region of the image (typically a rectangle),
- storing the image in a graphic file.

Utilizing ready to use software component that encapsulates the data structure of the raster array and basic operations on the image simplifies the implementation of GC applications by isolating a programmer from details of pixel attributes efficient storage and from details of graphic file formats. In Java environment the raster image can be represented by objects from `BufferedImage` class. The class is defined in `java.awt.image` package. It provides all operations that are necessary in order to create the image programmatically on pixel-by-pixel basis. Below, the most useful methods of `BufferedImage` class are summarized. The complete specification can be found in Java documentation.

`public BufferedImage( int width, int height, int imageType )`
This constructor creates a buffered image of the resolution determined by `width` and `height` parameters. Keep in mind that in `BufferedImage` objects the row index increases downwards, i.e. the topmost row has the index 0 while the bottom row is indexed by `(height-1)`. The `imageType` parameter determines what attributes describe a pixel in the image and how pixel attributes are stored in the memory. The value should be set to one of constants defined in this class. It makes possible to define grayscale of color image. It also determines how many bits will be used to represent each attribute value. We recommend to use the constant `TYPE_INT_RGB`. The images created with this constant have 8-bit components of R,G,B in the 32-bit sequence corresponding to the integer number. Blue (B) component occupies 8 least significant bits, green (G) is placed on bits 8-15 and the red component (R) occupies bits 16-23. The most significant eight bits are not used.

`public int getHeight()`
`public int getWidth()`
These utilities return the resolution of the image.

`public int getRGB(int x, int y)`
The utility provides RGB values of the pixel at the raster array location determined by `x,y` parameters. The color value is returned as an integer value packed as described in the constructor specification. The packed attributes can be decomposed into individual R,G,B values by logical masking and bit shifting, using the code as follows:

```
int           R, G, B;
BufferedImage image;
int           packed;
// . . .
packed = image.getRGB( x, y );
R = (packed & 0x00FF0000) >> 16;
G = (packed & 0x0000FF00) >> 8;
B = (packed & 0x000000FF);
```

public void setRGB(int x, int y, int rgb)
This utility assigns RGB attributes to the pixel at the position specified by `x,y` parameters. The pixel attributes must be packed into 32-bit integer value. The attribute assembling can be achieved using bit shifting and logical OR operation by means of the code as follows:

```
int           R, G, B;
BufferedImage image
int           packed;
// . . .
packed = image.getRGB( x, y );
packed = (R & 0x000000FF) << 16 | (G & 0x000000FF) << 8 |
         (B & 0x000000FF);
image.setRGB( x, y, packed );
```

The color integer representation can be also created with the class `Color`. One of constructors of this class creates a color object from individual *R,G,B* values. Next, it can be converted to the integer representation using `getRGB()` method. Use the following code to follow this way of conversion:

```
Color c;
c = new Color( R, G, B );
image.setRGB( x, y, c.getRGB() );
```

Similarly, RGB components can be extracted from the packed color representation with `Color` class as follows:

```
Color c;
c = new Color( packed );
R = c.getRed();
G = c.getGreen();
B = c.getBlue();
```

The color packing/unpacking with `Color` class can be however slower that explicit assembling/disassembling with logical masking and bit shift operations.

In order to load or store the `BufferedImage` object, the appropriate static method from `ImageIO` class should be used. It can load/store an image in one of the supported graphic file formats:

public static boolean write(RenderedImage img,
String formatName,
File output)
This method of `ImageIO` class writes the image `img` using the file format specified by the `formatName` parameter to the file represented by the `output` parameter. The format name is a string that identifies requested graphic file format. It is practically equivalent to used typical extensions of graphic files in Windows environment ("bmp", "jpg", "png", "gif" etc.).

public static BufferedImage read(File input)

This method of `ImageIO` can be used to read contents of a graphic file into the `BufferedImage` object. The object is created by this method and the reference to it is returned as the method value.

`read()` and `write()` methods throw `IOException`, so they must be called in the context of `try/catch` statement.

`public static String[] getReaderFormatNames()`
`public static String[] getWriterFormatNames()`
These utilities can be used to query informal names of formats that can be used in `read()` and `write()` methods. The informal names are practically equivalent to typical graphic file extensions used in Windows environment.

Creating procedural patterns in raster images

Any image can be created in pixel-by-pixel manner. It means that the program visits each pixel of the raster image (typically in two nested loops, the external loop iterates over rows while the internal one iterates over columns in the current row) and calculates the color of each pixel individually. Therefore the general framework of a program should be arranged as follows:

```
initialize the empty image by specifying its resolution and pixel
    representation;
for each row i
   for each column j in i-th row
      calculate the color of the pixel (i,j);
      put calculated color to the raster array of the image;
save the complete raster image in a graphic file;
```

The only fragment of this pseudocode that is specific to the pattern being created is the step related to calculation of the pixel color. As an example, let us consider the method that renders the rectangular image containing the set of concentric black and white rings. At each pixel, the decision needs to be made what color (black or white) should be set at the position *(i,j)*. Let x_{res}, y_{res} denote variables determining the image resolution. The image center coordinates are $x_c = x_{res}/2$, $y_c = y_{res}/2$. The single ring width is w pixels. The decision on color selection is based on the Euclidean distance of the pixel *(i,j)* to the pattern center.

$$d = \sqrt{(i - y_c)^2 + (j - x_c)^2}\ .$$

The rings can be indexed so as the most inner ring has the index 0. The index of the ring covering the pixel *(i,j)* is

$$r = (\text{int})d\ /(\text{int})w\ ,$$

where / is the integer division operator. If the ring index is even then the pixel color is black, otherwise it is white. The complete code that implements this method is presented in Listing 1. The code is explained in details in included comments.

```
1:   /*
2:    * Computer graphics courses at Wroclaw University of Technology
3:    * (C) Wroclaw University of Technology, 2010
4:    *
```

```
 5:    * Description:
 6:    *   This demo shows basic raster operations on raster image
 7:    *   represented by BufferedImage object. Image is created
 8:    *   on pixel-by-pixel basis and then stored in a file.
 9:    *
10:    */
11:
12:   import java.io.*;
13:   import java.awt.image.*;
14:   import javax.imageio.*;
15:
16:   public class Demo0
17:   {
18:      public static void main(String[] args)
19:      {
20:          System.out.println("Ring pattern synthesis");
21:
22:          BufferedImage  image;
23:
24:          // Image resolution
25:          int x_res, y_res;
26:
27:          // Ring center coordinates
28:          int  x_c, y_c;
29:
30:          // Predefined black and white RGB representations
31:          // packed as integers
32:          int  black, white;
33:
34:          // Loop variables - indices of the current row and column
35:          int  i, j;
36:
37:          // Fixed ring width
38:          final int w = 10;
39:
40:          // Get required image resolution from command line arguments
41:          x_res = Integer.parseInt( args[0].trim() );
42:          y_res = Integer.parseInt( args[1].trim() );
43:
44:          // Initialize an empty image, use pixel format
45:          // with RGB packed in the integer data type
46:          image = new BufferedImage( x_res, y_res,
47:                                     BufferedImage.TYPE_INT_RGB);
48:
49:          // Create packed RGB representation of black and white colors
50:          black = int2RGB( 0, 0, 0 );
51:          white = int2RGB( 255, 255, 255 );
52:
53:          // Find coordinates of the image center
54:          x_c = x_res / 2;
55:          y_c = y_res / 2;
56:
57:          // Process the image, pixel by pixel
58:          for ( i = 0; i < y_res; i++)
59:             for ( j = 0; j < x_res; j++)
60:             {
61:                double d;
62:                int    r;
63:
64:                 // Calculate distance to the image center
65:                d = Math.sqrt( (i-y_c)*(i-y_c) + (j-x_c)*(j-x_c) );
66:
67:                // Find the ring index
68:                r = (int)d / w;
```

```
69:
70:               // Make decision on the pixel color
71:               // based on the ring index
72:               if ( r % 2 == 0)
73:                  // Even ring - set black color
74:                  image.setRGB( j, i, black );
75:               else
76:                  // Odd ring - set white color
77:                  image.setRGB( j, i, white );
78:            }
79:
80:            // Save the created image in a graphics file
81:            try
82:            {
83:               ImageIO.write( image, "bmp", new File( args[2]) );
84:               System.out.println( "Ring image created successfully");
85:            }
86:            catch (IOException e)
87:            {
88:               System.out.println( "The image cannot be stored" );
89:            }
90:      }
91:
92:      // This method assembles RGB color intensities into single
93:      // packed integer. Arguments must be in <0..255> range
94:      static int int2RGB( int red, int green, int blue)
95:      {
96:         // Make sure that color intensities are in 0..255 range
97:         red   = red   & 0x000000FF;
98:         green = green & 0x000000FF;
99:         blue  = blue  & 0x000000FF;
100:
101:        // Assemble packed RGB using bit shift operations
102:        return (red << 16) + (green << 8) + blue;
103:     }
104:  }
```

Listing 1. The example of procedural pattern creation

The image created by the exemplary program from Listing 1. is presented in Fig. 3.

Fig. 1.3. Ring pattern created by Demo0 program

Assignment scope

1. Modify Demo0 program so as to create the following patterns:

a. Fuzzy rings pattern - the pattern similar to this one created by Demo0 program where transitions between rings are fuzzy and in result, grayscale image is produced, where gray level *I* is a following function of the distance *d* to the image center:

$$I(d) = 128 * (\sin(\frac{\pi d}{w}) + 1)$$

b. Regular color grid. The pattern parameters: grid line width, distance between adjacent grid lines along x and y axes as well as grid and background colors should be passed as command line parameters. The example of the grid pattern is shown in Fig. 1.4.

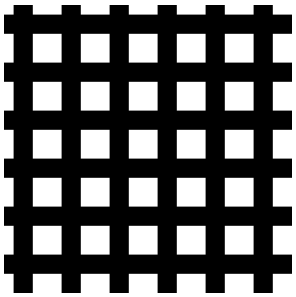


Fig. 1.4. The example of a grid pattern

c. Checkerboard pattern. Field colors and field square size should be passed as command line parameters.

2. Impose the regular pattern onto the loaded image. Load the specified image from a graphic file before the pattern creation loop starts. Then use concentric rings, grid and checkerboard patterns as a mask. In pixels filled with white color leave the original color of the loaded image. Set only the color of pixels that were filled with black color. Set the output image resolution equal to the resolution of the input image. The expected results are shown in Fig. 1.5.

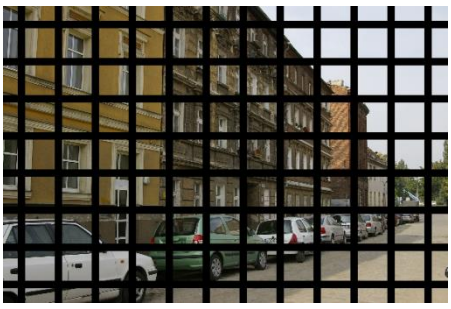


Fig. 1.5. The example of a grid pattern imposed on a read-in image

3. Create procedural patterns similar to these ones presented in Fig. 1.6

4. Load two images from graphic files. Use images of the same resolution. Then use rings, grid and checkerboard patterns as switching images, i.e. set the color of output image pixel by getting colors from corresponding positions in one of input images. Select the first or second input image as the source depending on the color of the switching procedural pattern. If the pixel color in the pattern is white then select the first image as the source, otherwise use the second input image.

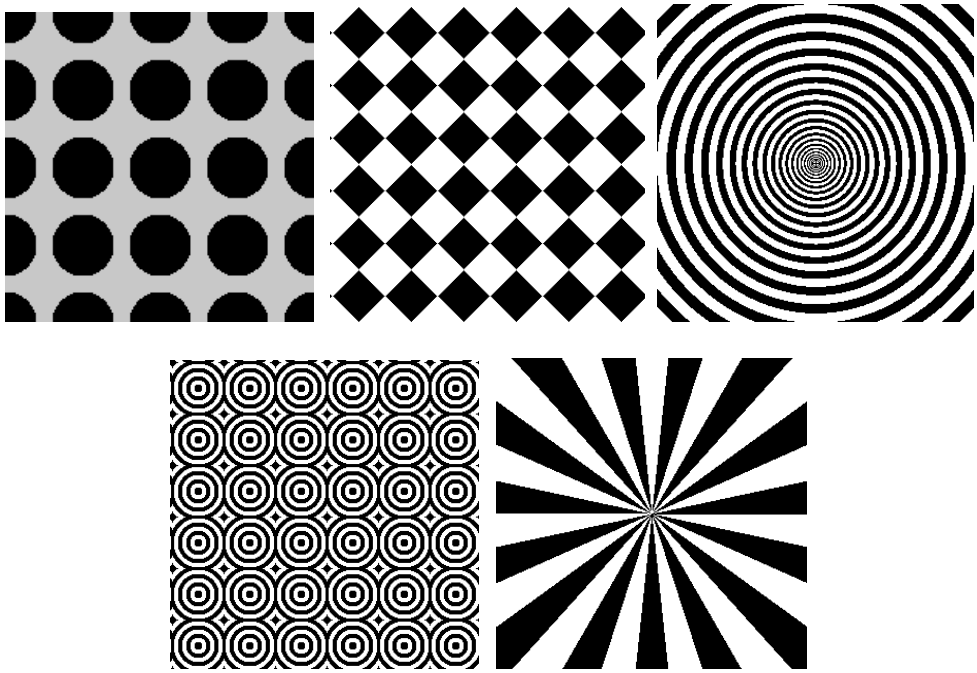


Fig. 1.6. Examples of other procedural patterns

Assignment 2

Application of vector and raster drawing services in on-screen graphics

Aim

The aim of this assignment is to learn how to build CG application that displays drawings and raster images on the surface of windows. The fundamental classes provided by Swing package will be explained. The geometric primitives available in Java2D will be used in vector image creation and display procedures.

Theoretical fundamentals

Graphics-related elements of the operating system and CG application architecture

Programming of modern applications in CG is strongly supported by software and hardware components provided by the *Graphical User Interface* (GUI) layer of the operating system (OS) and by development environments of programming languages. The set of components typically constitutes the layered architecture, partly included in GUI layer of OS and partly supported by runtime libraries linked to the CG application. The structure of typical CG-related components is presented in Fig. 2.1.

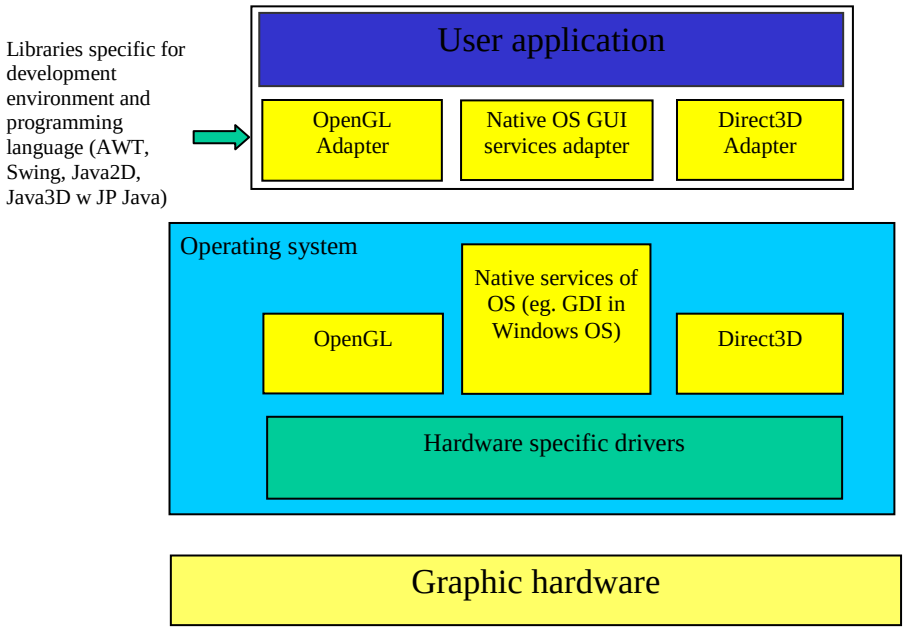


Fig. 2.1. The architecture of CG components

Most operations both in 3D and 2D graphics are supported by graphic hardware. For safety and stability reasons, the user application code is not allowed to make direct access to the installed hardware. The operations executed on the hardware level are called via the intermediate layers. Due to differences in communication protocols in the hardware interface, the control and data transfer operations related to the graphics board are implemented by manufacturer-specific drivers. Hardware drivers are specific software components installed as extensions (plug-ins) to the operating system. On the back-end side, the driver code controls the graphics board by sending commands and supervising data transfer on the system bus. On the front-end, it provides standardized (OS-specific) interface to higher layers of OS. The next, higher layer consists of software modules encapsulated in OS GUI. They may implement complex operations related to 3D scene management and display (as in the case of Windows Direct3D or OpenGL library) or provide low level utilities for screen display and for arrangement and management of interactive user interface elements. These libraries provide the interface to the hardware drivers. In case of many services provided at this level, the service implementation consists of just issuing the corresponding request to the hardware via the hardware driver. If the requested service is not provided by the installed hardware, the library provides its software implementations. In this way, the CG application programmer is provided with uniform set of CG services independent on the hardware configuration of the computer system.

The end user application code is linked with libraries corresponding to libraries available in OS GUI layer. The linked libraries may provide just "stub" subroutines of the library services. The role of linked stub is to call the OS service that implements the functionality to the called subroutine. In this way the CG programmer does not have to bother about the convention of OS service calls. The libraries on this level implement nontrivial operations in case where the elements of GUI are assumed to be OS independent. In such case the CG services need to be completely implemented inside the library code. We deal with such situation in Swing package that provides OS independent GUI style in Java programming environment. The CG application programmer typically uses high level utilities available in the applied programming environment.

Modes of graphics programming in the interactive environment

If the result of a CG program execution is to be displayed on the computer screen we will call such a program *interactive CG program*. Two modes of interactive CG can be distinguished:

- *Off-screen* – the image is created (drawn or processed) in the object (data structure), which typically represents the pixel array of a raster image. The image created in this mode is not directly linked with any GUI element and graphic operations executed on it have no immediate effect on the display screen.
- *On-screen* – the image is created directly on the screen, results of CG operations executed in the graphic object are immediately displayed on the screen. The image is represented by the data structure (object) registered by OS as the element of the window managed by the GUI module of OS. CG operations in this mode may be slower than in the case of off-screen mode, additionally some limitations may hold, usually related to abilities of installed hardware (e.g. limitations in the image resolution related to the amount of graphic board RAM or to the maximal buffer sizes supported by the graphic hardware).

API for CG programming provides the abstraction (metaphor) of things we use when drawing or painting in the real world. We need a surface on which we create the image (image canvas) and the specific drawing/painting tool like a brush, pen or pencil. The image canvas is typically framed. The drawing/painting is achieved by moving appropriately the drawing tool, e.g. along straight lines, curves of arches etc. In CG applications images are created in screen windows that are metaphor of frames of image canvas in the real world. The area of the window available for drawing is modeled by a panel, which corresponds to two-dimensional pixel array that can be used for raster or vector operations. The drawing tool is modeled by a *drawing context*. The drawing context is a data structure bound to a panel. It defines properties of the drawing tool. The results of drawing of painting operations depend on the style defined by the drawing context. The drawing context determines such properties like:

- pen color,
- pen thickness,
- line style (continuous, dotted, dashed),
- *drawing mode* – the method in which the pen color modifies the canvas pixel color at the point of drawing:
  - color replacement (final color is always the same as the pen color),
  - color blending (pen and background colors are mixed in proportions defined by the blending coefficients either defined as a common value for the whole image or determined by the *alpha* attribute of the raster image pixel),
  - **XOR** mode.

Drawing in XOR mode consists in applying exclusive OR (XOR) logical operation to corresponding bits representing the background color and the color of the drawing tool. Consider the situation where the drawing tool color is white i.e. *R=G=B=255*. The packed pen color is represented by the sequence of 24 bits, each of them is equal to *1*. Let b_i and p_i denote the bits on the *i*-th position of the background and pen color representation at the point of drawing. The results of XOR ($\otimes$) operation applied to the *i*-th bit is shown in the following table:

b_i	p_i	$c_i = p_i \otimes b_i$
0	1	1
1	1	0

XOR operation applied with white pen color inverts each bit of the background color. In the result, the color of the background at the point of drawing in XOR mode always changes. It prevents the situation of drawing with the certain pen color in the area where the background color is equal to the pen color and the result of drawing is not visible. Moreover, if the drawing operation will be repeated again with the same pen color, the original background color will be restored and the result of the first draw operation will be canceled. Drawing in XOR mode is often applied in situations where the skeleton of the shape (e.g. the clipping rectangle, ruler etc) is to be moved interactively over the background image after each movement of the pointing device. Without XOR drawing, the whole background image would have to be repainted what would lead to unacceptably slow operation.

Preparing the window for graphic operations with Swing components

Java programming environment provides a wide set of component supporting interactive CG programming. Vector and raster operations can be implemented by the set of classes

known as Java2D. GUI elements related to on-screen window management and to building convenient interactive UI are provided in Swing package. Swing package is platform-independent, i.e. the look of the GUI of an application does not depend on the underlying OS GUI. In this exercise we will focus on basic operations related to vector drawing on the surface of screen windows.

Before any CG operation is executed, the interactive program has to create the window in which results of drawing operations will be visible. Screen windows are represented by objects from `JFrame` class. The window area contains its standard elements as window frame, window bar, control icons and menu bar (if created). The remaining area of the window available for drawing is called *content pane*. The content pane can contain other elements which in turn can contain its inner components. It constitutes a *containment hierarchy* which always starts with a top level container. Top level container can be an object of one of `JFrame`, `JDialog` or `JApplet` classes. The exemplary containment hierarchy is presented in Fig. 2.2.

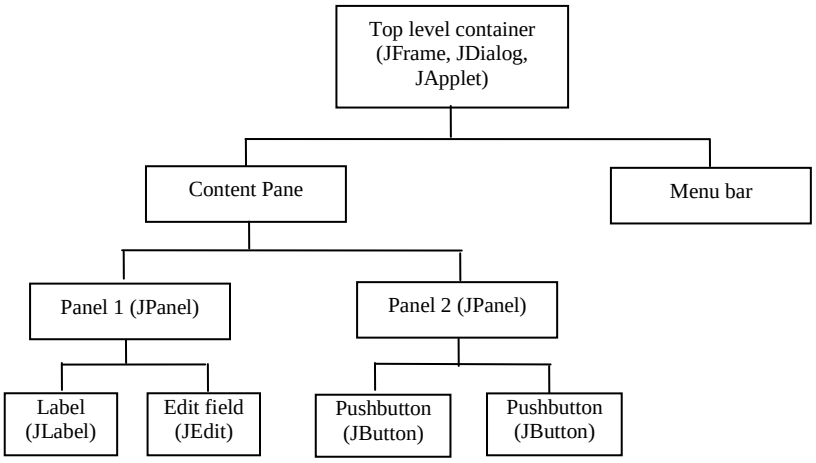


Fig. 2. 2. Exemplary containment hierarchy

Although it is possible to draw over the whole area of the window, it is recommended not to use the fragments of the window covered by standard elements. Drawing operations should be rather executed within the area of the content pane.

The reference to the content pane object can be queried with `getContentPane()` method of `JFrame`. It returns the reference to the `Container` class that is the superclass of all classes implementing various containers of UI elements. UI elements (or to be more precise - objects that model UI elements) can be added directly to the content pane. In the case of image drawing, the more common practice is however to replace the default content pane object with an own object entirely responsible for complete window image drawing. The class that can be used as canvas of images and as a container for other UI elements is `JPanel` class. The new content pane can be set using the `setContentPane()` method of `JFrame`.

In order to prepare the environment for image drawing the following operations should be executed:

```
create the window object from JFrame;
replace its contents pane with the object form own class
   derived from JPanel;
display the window;
```

Alternatively, the own window class can be derived from `JFrame` and the content pane replacement can be done within its constructor code.

Drawing operations are executed using a drawing context bound to the canvas on which we want to draw. The drawing context is modeled by the class `Graphics2d`. Before a drawing operation is executed the appropriate drawing context must be acquired from the object that models the drawing canvas.

The image drawn or painted on the canvas is volatile, i.e. drawn once it remains in the window only as long as it is the foreground window. If the window fragment is shifted out of physical screen or covered by any other window, the drawn image may be damaged. This is because the window manager does not store the contents of the window drawing panel. It is the CG application responsibility to take care of the window image refresh operation. `JFrame` and `JPanel` classes provide the mechanism that can be used to correctly refresh the window content when necessary. If the window manager detects the situation in which the window content can be damaged, it calls the specific method `paintComponent()` of `JPanel` object embodying the content pane of the window. If the whole image drawing procedure is implemented inside the `paintComponent()` method, then it will be executed automatically whenever the window image needs to be refreshed. It is automatically called also in the case of changing of the window size or aspect ratio, so the image can be redrawn taking into account the actual display panel size.

The following methods of the `JFrame` class can be useful when creating and displaying the window for image drawing operations.

`JFrame(String title)`
The constructor initializing a JFrame object. The window is not displayed until the display method is explicitly called. title parameter defines the window name that is displayed on its top bar.

`public void setDefaultCloseOperation(int operation)`
The method defines the action that will be executed when the window close icon (X) is clicked. The operation parameter can be one of DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE or EXIT_ON_CLOSE. The usage of the last constant causes the whole application is terminated on "close" operation.

`public void show()`
`public void hide()`
`public void setVisible(boolean visibility_status)`
These methods display or hide the window.

`public void setLocation( int x, int y )`

The method sets the position of the upper left vertex of the window to (x,y) in screen coordinates system. The same method is available in other classed inheriting from Component class. The (x,y) coordinates are in general related to the parent coordinates space.

`public void setBounds( int x, int y,`
`int width, int height)`
The method sets position and size of the window in the parent (screen) coordinate space.

`public Container getContentPane()`
The method returns the reference to the current content pane object. Formally, the reference to the `Container` class is returned which is a superclass of all classes implementing UI element containers. In most cases, typecast to the `JComponent` type is required.

`public void setContentPane(Container contentPane)`
The method sets the new content pane object for a `JFrame` window. The reference to `JPanel` object is passed as a `contentPane` parameter value. The `paintComponent()` method of the passed `JPanel` object is supposed to implement the complete image drawing operation of the window.

Drawing vector primitives with Java2D

It is recommended to place all window image drawing operations within `paintComponent()` method of the `JPanel` object that replaces default content pane of the top level window. All drawing operations are executed via the graphic context object of the image panel. If `JPanel` is the model of the image canvas then the drawing context reference can be acquired with `getGraphics()` method. Formall, this method returns the reference to the `Graphics` class, but in the case of Swing and Java2D, the actual reference is to `Graphics2D` object. In order to make use of most of Java2D advantages, the typecast should be applied to convert the reference to `Graphics2d`.

The simples method of vector drawing consists in using simple drawing operations which draw simple shapes like lines, polygons, ovals. The geometric parameters of shapes are explicitly specified as drawing method parameters. All drawing operations are implemented as `Graphics2D` methods:

`public void drawLine(int x1, int y1, int x2, int y2)`
The method draws the line segment which end point coordinates are defined by `(x1, y1)` and `(x2, y2)`
.
`public void drawRect(int x, int y, int width, int height)`
This method draws the rectangle which upper left vertex is located at the point `(x,y)` and the size is determined by `width` and `height` parameters.

`public void drawOval(int x, int y, int width, int height)`
This method draws the ellipse inscribed into the rectangle which upper left vertex is located at the point `(x,y)` and the size id determined by `width` and `height` parameters.

`public void drawArc(`

```
    int x, int y,
    int width, int height,
    int startAngle, int arcAngle)
```

The method draws the fragment of the circle or ellipse inscribed into the rectangle as in the case of `drawOval` method. `startAngle` and `arcAngle` define the fragment of the ellipse to be drawn. A Star angle is measured as the positive angle in Euclidean 2D coordinate system.

If the image drawing procedure is based on the usage of the above methods then the image contents and structure are entirely embodied by the drawing code. Sometimes it is more convenient to create the explicit data structure that models the contents of the vector image. Then, to draw any image the code can be used that does not depend on the particular image. The code interprets the data structure and executes appropriate drawing operations according the image contents represented in the data. Java2D provides the set of classes that are models of basic 2D shapes: line segments, rectangles, polylines and ovals. All classes representing 2D shapes are derived from the common superclass `Shape`. The objects can be used to permanently represent the contents of the image. 2D objects can be passed to drawing context `draw()` method that draws them on the image canvas.

2D shape classes are defined in a bit unusual way. Two variants of each shape are available. One of them accepts double parameters, the other accepts float parameters. The variants are defined as inner public classes of the outer class defined for a 2D shape which is in fact an abstract class. For example, if `X2D` is the identifier of the shape class then in order to define the shape variable x that will used double type parameters the following declaration should be used:

```
X2D.Double  x;
```

Java2D provides the wide class of 2D shape objects including: `Arc2D`, `CubicCurve2D`, `Ellipse2D`, `Line2D`, `Path2D`, `Polygon`, `QuadCurve2D`, , `Rectangle2D`, `RoundRectangle2D`. Refer to Java2D documentation for application details.

All drawing operations apply the pen style defined in used drawing context. Most commonly used pen attributes are: pen color, pen line width, pen line style and drawing style. `Graphics2D` class provides utilities that can be used to set appropriate pen attributes.

`public abstract void` **`setColor`**`(Color c)`
The method sets the pen color for all subsequent drawing operations.

`public abstract void` **`setStroke`**`(Stroke s)`
The method sets other pen properties that are defined by attributes of the passed stroke object. Usually the object form the class `BasicStroke` is passed as the parameter. The application example is presented in the listing of Demo1 program.

```
1: import java.awt.BasicStroke;
2: import java.awt.Color;
3: import java.awt.Container;
4: import java.awt.Dimension;
5: import java.awt.Graphics;
```

```
 6: import java.awt.Graphics2D;
 7: import java.awt.Toolkit;
 8: import java.awt.geom.Ellipse2D;
 9: import java.awt.geom.Line2D;
10: import java.awt.geom.Rectangle2D;
11: import java.util.Scanner;
12:
13: import javax.swing.JFrame;
14: import javax.swing.JPanel;
15:
16:
17: public class Demo1a
18: {
19:    private static Scanner in;
20:
21:     public static void main(String[] args)
22:     {
23:        // Component for reading simple data from the console
24:       in = new Scanner(System.in);
25:
26:       // ================================================================
27:       // Actions below show how to create a window and how to manipulate
28:        // its position on the screen and dimensions. The example utilizes
29:        // components defined in Swing package. One can also perform
30:        // analogous operations using older and smaller package AWT.
31:       // ================================================================
32:
33:        // Create object of type derived from JFrame window for
34:       // our graphical operation.
35:       // ================================================================
36:       Demo1Window wnd = new Demo1Window();
37:
38:       // Set the window property causing tha window close operation
39:       // also terminates the program
40:       wnd.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
41:
42:       // Display the window on the screen
43:       wnd.setVisible(true);
44:
45:       // Positioning and resizing the window
46:       // ================================================================
47:       writeln( "Press ENTER to set window position" );
48:       readln();
49:        // Set window position only
50:        wnd.setLocation( 50, 50);
51:
52:       writeln( "Press ENTER to set window position and size" );
53:        readln();
54:        // Set both position and size
55:       wnd.setBounds( 70, 70, 300, 300);
56:
57:       writeln( "Press ENTER to set the window size " +
58:                "to the half of the screen area" );
59:        readln();
60:
61:       // Acquire screen size data
62:       Toolkit scrinfo = Toolkit.getDefaultToolkit();
63:       Dimension dim = scrinfo.getScreenSize();
64:       // Set the window size to cover half of the screen
65:       wnd.setBounds( 50, 50, dim.width / 2, dim.height / 2);
66:
67:       // ================================================================
68:       // The following operations have volatile effects because drawn
69:       // elements are not refreshed automatically.
```

```
70:       // ============================================================
71:
72:       // Get drawing context of the displayed window
73:       // ============================================================
74:       Graphics g = wnd.getGraphics();
75:       Graphics2D  g2d = (Graphics2D)g;
76:
77:       // Example of usage of XOR mode for displaying a shape and
78:       // to undo the drawing operation results
79:       // ============================================================
80:       writeln( "Press a key to display a rectangle" );
81:       readln();
82:
83:       // Set XOR drawing mode, use white pen
84:       g2d.setXORMode( new Color( 255, 255, 255 ));
85:       Rectangle2D.Double rectangle;
86:       rectangle = new Rectangle2D.Double(40, 40, 60, 90);
87:       g2d.draw( rectangle );
88:
89:       // Now draw the same rectangle again - observe that it disappears
90:       writeln( "Press a key to undo the drawing operation" );
91:       readln();
92:       g2d.draw( rectangle );
93:
94:       writeln( "Press a key to close a terminate the program" );
95:       readln();
96:
97:       System.exit( 0 );
98:    }
99:
100:   //=====================================================================
101:   // Console methods
102:   //=====================================================================
103:   static void writeln( String stg )
104:   {
105:      System.out.println( stg );
106:   }
107:
108:   static void readln()
109:   {
110:      try
111:      {
112:         while( System.in.read() != '\n' );
113:      }
114:      catch( Throwable e )
115:       {
116:       }
117:   }
118: }
119:
120: // ====================================================================
121:
122: class Demo1Panel  extends JPanel
123: {
124:    Demo1Panel()
125:    {
126:       super();
127:
128:       // Set the panel background color to gray
129:       setBackground( new Color( 200, 200, 200) );
130:    }
131:
132:    //=====================================================================
133:    // paintComponent method called automatically to refresh
```

```
134:    // the window contents
135:    //=====================================================================
136:    public void paintComponent( Graphics g)
137:    {
138:       super.paintComponent(g);
139:
140:       Graphics2D  g2d = (Graphics2D)g;
141:
142:       // ============================================================
143:       // Implement the whole window drawing operations here, inside
144:       // PaintComponent method. It will be automatically called
145:       // whenever the window image id destroyed. In result the window
146:       // contents will be automatically refreshed.
147:       // ============================================================
148:
149:       // Write a string in the window
150:       g2d.drawString( "Image is automatically refreshed", 300, 100);
151:
152:       // Draw a diagonal of the content pane, use Java2D object
153:       // representing a line segment
154:       Line2D.Double diagonal;
155:       Dimension size = getSize();
156:       diagonal = new Line2D.Double( 0, size.height , size.width, 0 );
157:       g2d.draw( diagonal );
158:
159:       // Set the color for subsequent drawing operations
160:       g2d.setColor( new Color( 255, 0, 0 ) );
161:
162:       // Set stroke width to 10 pixels
163:       g2d.setStroke( new BasicStroke( 10 ) );
164:
165:       // Now draw a line using defined pen attributes
166:       Line2D.Double  line;
167:       line = new Line2D.Double( 0, 0, size.width, size.height );
168:       g2d.draw( line );
169:
170:       // Restore default attributes
171:       g2d.setColor( new Color( 0, 0, 0 ) );
172:       g2d.setStroke( new BasicStroke( 1 ) );
173:
174:       // Create a rectangle object and draw it
175:       Rectangle2D.Double  rectangle;
176:       rectangle = new Rectangle2D.Double(1, 1, 60, 90);
177:       g2d.draw( rectangle );
178:
179:       // Create the ellipse object and draw it in red color
180:       Ellipse2D.Double  ellipse;
181:       g2d.setPaint( new Color( 255, 0, 0) );
182:       ellipse = new Ellipse2D.Double(100, 100, 80, 80);
183:       g2d.draw( ellipse );
184:
185:       // Now set the fill color to blue and fill the interior area
186:       // of the ellipse with this color
187:       g2d.setPaint( new Color( 0, 0, 255) );
188:       g2d.fill( ellipse);
189:    }
190: }
191:
192: // ========================================================================
193:
194: class Demo1Window extends JFrame
195: {
196:    public static final int WIDTH  = 800;
197:    public static final int HEIGHT = 500;
```

```
198:
199:    public Demo1Window()
200:    {
201:       // Call the superclass constructor that sets the window title
202:       super ("Demo1 - how to use vector drawing utilities" );
203:
204:       // The default size of the window can be set here
205:       setSize( WIDTH, HEIGHT );
206:
207:       // Create own panel and use it as the replacement
208:       // of the default window pane
209:       setContentPane( new Demo1Panel() );
210:    }
211: }
```

Two drawing styles are presented in Demo1 program. The first style consists in executing all window drawing operations within `paintComponent` method of the `JPanel`-derived class that replaces the default component implementing the window content pane. If this drawing style is applied then the window contents are always refreshed correctly. The drawing operation examples are contained in lines 138-188. The alternative style consists in acquiring a window drawing context (line 74) and executing drawing operations form the code outside the content pane class (lines 84-92).

It is also possible to display the image represented by `BufferedImage` object on the window panel. It can be done with `drawImage()` method of Graphics2D drawing context. Many variants of overloaded `drawImage()` functions are available. The following variants seem to be most useful (see Java documentation for remaining variants):

```
public boolean drawImage(
    Image img,
    int x, int y,
    ImageObserver observer)
```

The image passed as `img` reference parameter is displayed on the pane so that the upper left vertex is located a the position (`x,y`). `observer` reference can be set to null. The image is displayed in its original size. If there is not enough room on the pane to display the whole image, it is clipped, i.e. only the upper left fragment of the image is displayed.

```
public boolean drawImage(
    Image img,
    int x, int y,
    int width, int height;
    ImageObserver observer)
```

The image passed as `img` reference parameter is displayed on the pane so that the upper left vertex is located a the position (`x,y`) and the image is scaled appropriately so as to fit in the rectangle of the size defined by width and height parameters. The display area aspect ratio does not have to correspond to the image aspect ratio. In case of differences the image will be distorted. The scaling procedure may not be precise and some artifacts can be introduced. Fig. 2.3. shows the result of displaying the image with scaling.

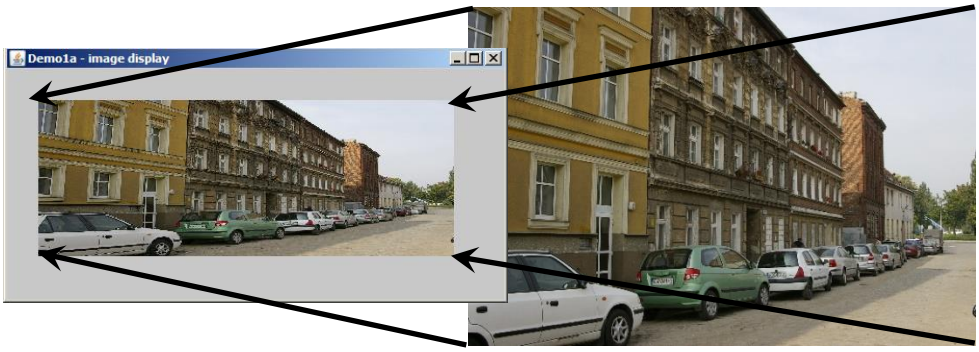


Fig. 2.3. Image distortions being a result of not equal aspect ratios of the image and the display area

```
public boolean drawImage(
    Image img,
    int dx1, int dy1, int dx2, int dy2,
    int sx1, int sy1, int sx2, int sy2,
    ImageObserver observer)
```

This variant displays the image rectangular fragment specified by `sx1, sy1, sx2, sy2` parameters in the display rectangle on the window pane specified by `dx1, dy1, dx2, dy2`. `(sx1, sy1)` and `(sx2, sy2)` are upper left and lower right vertices in the image. `(dx1, dy1)` and `(sd2, sd2)` are corresponding upper left and lower right vertices in the window pane.

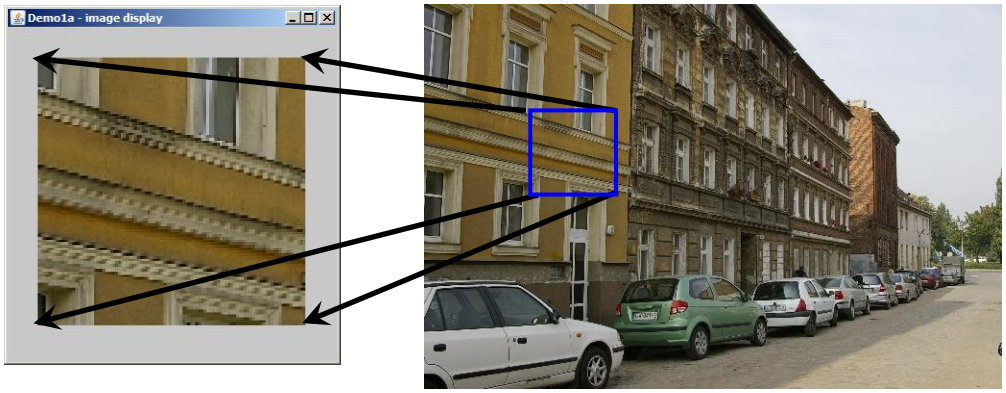


Fig. 2.4. Displaying the selected fragment on the image in a rectangular area of the window

Assignment scope

1. Write the program that plots sine and cosine functions in the interval <0,4π>. Plot the sine curve with the red line and the cosine curve with the blue line. Scale the plot so as the whole window pane is filled with the image. Add axes and grid lines drawn with doted lines. Implement plots of other functions defined by the teacher.

2. Draw the star as the multi-vertex polygon. The number of star vertices should be defined by a command line parameter.

3. Draw the color flag of your country. Preserve the correct aspect ratio of the flag and assure that the flag image is extended so as to completely occupy at least one of the window pane sizes.

4. Write a `drawTree` method that draws a shape of tree at the specified position. Pass the position of the figure and the drawing context as parameters of the method. Than draw a set of trees (forest) on the window panel by calling the `drawTree` method with various parameters.

5. Write the program that displays the image stored in the graphic file. The path and name of the graphic file is specified as the command line parameter. Initially set the window size so as to cover 80% of available screen resolution along horizontal and vertical axis. Assure that the available area of the window content pane is utilized in the maximal degree but preserve the image aspect ratio. This requirement should be satisfied also after the window is resized by dragging of its edges or vertices with the mouse.

Assignment 3
Simple animation of vector images

Aim

The aim of this assignment is to learn how to implement a simple animation of 2D vector image. Useful Java components and their methods that can be used in the animation are presented. In particular, methods of the current date and time access are presented.

Theoretical fundamentals

The animation in CG consists in rendering of the series of images which, when displayed in appropriate rate, give the effect of smooth changes in the displayed scene or image. The animation is mainly used in 3D computer graphics, but the same concepts can be easily explained in the domain of 2D graphics. The animation is achieved by changing some scene description elements in time and by rendering static images corresponding to subsequent time moments. The single static image being an element of animated sequence will be called a *frame*. The following elements of 2D images are most often animated:

- position and size of the elements,
- visual attributes of image elements (colors, transparency),
- patterns used to fill areas of 2D shapes.

In the case of 3D graphics the following elements can be animated:

- scene geometry,
- light intensity, position and directional properties,
- observer parameters (in walk-through animations),
- appearance of volumetric effects (e.g. animation of smog, animation of flames etc.).

In order to define the animation sequence, appropriate functions defining the dependency of the image contents description elements on time must be defined. For each frame in the animation sequence its time stamp is known. The parameters are computed for the current time and the image is displayed or stored. The animation can be implemented as an interactive program where the animated image in displayed on the screen in real time. It can be also implemented as a batch process, where the sequence of created animation frames is stored in a graphic file (or as a series of single image files). Here we will consider the basic implementation of the interactive animation. The interactive animation can be implemented using the following general code pattern:

```
create the window for animation display;
repeat
   get current time t;
   compute the current image description parameters as functions
      of the current time t;
   display the image using computed image content description parameters;
   suspend the program for the time interval Δt;
until animation is broken;
```

In the simplest animation the main animation loop as well as the initialization of display window can be implemented in the `main()` method of the program. The procedure that acquires current time and complete drawing activities is executed within the

`paintComponent()` method of the window content pane. The following utilities can be useful when implementing the simple animation.

`public static void` **`sleep`**`(long sleep_time)`
This static method of `Thread` class suspends program execution for the time interval of the duration specified by `sleep_time` parameter. The suspension time is defined in milliseconds. In the case of its application in animation, the suspension time determines the image refresh rate. It should be selected reasonably. A High refresh rate results in smooth transitions in the animated image, but if the display procedure is complicated the animation activates can load the processor heavily. Selection of a too low refresh rate leads to discontinuities in the transitions in the animated image.

`public void` **`repaint`**`()`
This method defined in `Component` class forces the visual component contents to be redrawn. If the visual component is an object belonging to JFrame derived classes, the result of `repaint()` call is that the `paint()` method of the object is called. It in turn causes that the `paintComponent()` of the content pane object is called and finally the window contents is redrawn. It is recommended to use `repaint()` rather that to call `paintComponent()` directly. This is because direct calls of `paintComponent()` may be in conflict with system calls of this method. Additionally, calling `paintComponent()` requires the drawing context to be specified explicitly. If `paintComponent()` is called indirectly (as the consequence of `repaint()` call) then the drawing context of the pane is provided automatically.

Acquiring date and time

The current date and time can be queried with `Date` and `GeorgianCalendar` classes. The `Date` object, if created with the default constructor, contains the time stamp at the moment of creation. The time stamp is the number of milliseconds since 1st of January 1970. To get the current time in milliseconds create the date object and then query the current time immediately by calling `getTime()` method of the `Date` object. The method returns the current time as the long integer number being the object creation time stamp. In some applications such form of time representation may be sufficient to animate the image. For example if we want to display a rotating object then the only parameter that changes in time is the rotation angle. The rotation angle can be simply based on the elapsing time. Let *T* denote the period of rotation in seconds, i.e. the objects rotates by 360° each *T* seconds. Then the rotation angle at the current time can be determined with the following code.

```
Date    time;
double  angle;
Double  T;
long    msec;

// Set the rotation period to 5 seconds
T = 5.0;

// Acquire the time stamp in miliseconds
time = new Date();
msec = time.getTime();

// Set the angle in radians
angle = 2.0 * Math.PI * (0.001 * msec) / T;
```

Sometimes however the time expressed by hours, minutes and seconds since midnight is required (e.g. as in the example of an animated clock presented in the further part of this chapter). The time stamp acquired with the `Date` object can be converted to the complete date and time record (year, month, day of year, hour minute second) using `GregorianCalendar` class. The class implements `get()` method that returns the requested element of the date record, depending on the constant passed as the method parameter. The necessary constants are defined inside the abstract `Calendar` class which is a superclass for `GregorianCalendar`. See Java documentation for complete list of constants. The example below presents how to get current hour, minute and second.

```
Date               time;
GregorianCalendar  calendar;
int                hour, minute, second;

// Acquire the time stamp in milliseconds
time = new Date();

// Create the calendar and set it to current time
calendar = new GregorianCalendar();
calendar.setTime( time );

// Extract hour, minute and second from the calendar object
minute = calendar.get( Calendar.MINUTE );
hour   = calendar.get( Calendar.HOUR );
second = calendar.get( Calendar.SECOND );
```

The described animation techniques are used in the exemplary animated clock program presented in Listing 3.1.

```
 1:   import java.util.Calendar;
 2:   import java.util.Date;
 3:   import java.util.GregorianCalendar;
 4:   import java.awt.*;
 5:   import java.lang.Thread;
 6:   import java.lang.InterruptedException;
 7:   import javax.swing.*;
 8:
 9:   public class Clock
10:   {
11:      public static void main(String[] args)
12:      {
13:         // Create the window of the clock
14:         ClockWindow wnd = new ClockWindow();
15:
16:         // Closing window terminates the program
17:         wnd.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
18:
19:         // set the initial position of the window on the screen
20:         // and make the window visible
21:         wnd.setBounds( 70, 70, 300, 300);
22:         wnd.setVisible(true);
23:
24:         // Start the infinite loop of animation.
25:         // The program will run until the clock window is closed
26:         while ( true )
27:         {
28:            try
29:            {
30:               // Wait a second before the clock is redisplayed
31:               Thread.sleep( 1000 );
```

```
32:            }
33:            catch ( InterruptedException e )
34:            {
35:               System.out.println("Program interrupted");
36:            }
37:            // Redraw the clock according to current time
38:            wnd.repaint();
39:         }
40:      }
41:   }
42:
43:   // ================================================================
44:   // ClockPane class implements the content pane of the window
45:   // in which the clock is displayed
46:   // ================================================================
47:
48:   class ClockPane extends JPanel
49:   {
50:      // The length of the tick mark
51:      final int TICK_LEN = 10;
52:
53:      // Coordinates of the clock dial center
54:      int  center_x, center_y;
55:
56:      // Radiuses of the inner and outer circle enclosing the dial
57:      int  r_outer, r_inner;
58:
59:      // The calendar object - will be used to acquire the current
60:      // hour. minute and second
61:      GregorianCalendar calendar;
62:
63:
64:      ClockPane()
65:      {
66:         super();
67:         setBackground( new Color( 200, 200, 255) );
68:         calendar = new GregorianCalendar();
69:      }
70:
71:      // This method draws the single tick mark on the clock dial
72:      public void DrawTickMark( double angle, Graphics g )
73:      {
74:         int xw, yw, xz, yz;
75:
76:         angle = 3.1415 * angle / 180.0;
77:
78:         // The tick is drawn as a line segment
79:         xw = (int)(center_x + r_inner * Math.sin( angle ));
80:         yw = (int)(center_y - r_inner * Math.cos( angle ));
81:         xz = (int)(center_x + r_outer * Math.sin( angle ));
82:         yz = (int)(center_y - r_outer * Math.cos( angle ));
83:
84:         g.drawLine( xw, yw, xz, yz );
85:      }
86:
87:      // The method draws the clock hand. The hand angle and length
88:      // are specified by the method arguments
89:      public void DrawHand( double angle, int length, Graphics g )
90:      {
91:         int   xw, yw, xz, yz;
92:
93:         // Convert the angle from degrees to radians
94:         angle = 3.1415 * angle / 180.0;
95:
```

```
 96:         // Use this angle to find the hand outer end
 97:         xw = (int)(center_x + length * Math.sin( angle ));
 98:         yw = (int)(center_y - length * Math.cos( angle ));
 99:
100:         // Complement the angle and find the hand inner end
101:         angle += 3.1415;
102:         xz = (int)(center_x + TICK_LEN * Math.sin( angle ));
103:         yz = (int)(center_y - TICK_LEN * Math.cos( angle ));
104:
105:         g.drawLine( xw, yw, xz, yz );
106:      }
107:
108:      // This method draws the circular dial of the clock
109:      public void DrawDial( Graphics g )
110:      {
111:         g.drawOval( center_x - r_outer,
112:                     center_y - r_outer,
113:                      2*r_outer, 2*r_outer );
114:
115:         // Draw tick mark at location corresponding to
116:         // hours 1 .. 12
117:         for ( int i = 0; i <= 11; i++ )
118:            DrawTickMark( i * 30.0, g );
119:      }
120:
121:      // The complete drawing procedure in implemented in
122:      // paintComponent method, so it will be refreshed
123:      // automatically when necessary and also the repaint
124:      // of the clock can be forced by calling repaint()
125:      // method of the window.
126:      public void paintComponent( Graphics g )
127:      {
128:         int  minute, second, hour;
129:
130:         super.paintComponent(g);
131:         Graphics2D  g2d = (Graphics2D)g;
132:
133:         // Get actual window size in order to compute
134:         // basic clock dimensions
135:         Dimension size = getSize();
136:
137:         // Calculate the position of the dial center
138:         center_x = size.width/2;
139:         center_y = size.height/2;
140:
141:         // Find out the radiuses of inner and outer dial rings
142:         r_outer = Math.min( size.width, size.height)/2;
143:         r_inner = r_outer - TICK_LEN;
144:
145:         // Acquire the current time
146:         Date time = new Date();
147:
148:         // Convert it to hours/minutes/seconds
149:         calendar.setTime( time );
150:         minute = calendar.get( Calendar.MINUTE );
151:         hour   = calendar.get( Calendar.HOUR );
152:         if ( hour > 11 )
153:            hour = hour - 12;
154:         second = calendar.get( Calendar.SECOND );
155:
156:         // Draw the dial with tick marks
157:         DrawDial( g );
158:
159:         // Set the color and the line style how hour hand
```

```
160:         g2d.setColor( new Color( 255, 0, 0 ) );
161:         g2d.setStroke( new BasicStroke( 5 ) );
162:         // Draw the hour hand - the current hour hand angle
163:         // is passed as the method argument. The second argument
164:         // is the distance of the outer hand end from the dial
165:         // center
166:         DrawHand( 360.0 * (hour * 60 + minute) / ( 60.0 * 12 ) ,
167:                   (int)(0.75 * r_inner), g);
168:
169:         // Set the color and the line style of the minute hand
170:         // and display it
171:         g2d.setColor( new Color( 255, 0, 0 ) );
172:         g2d.setStroke( new BasicStroke( 3 ) );
173:         DrawHand( 360.0 * (minute * 60 + second )/( 3600.0),
174:                   (int)(0.97 * r_outer), g);
175:
176:         // Finally draw the second hand
177:         g2d.setColor( new Color( 0, 0, 0 ) );
178:         g2d.setStroke( new BasicStroke( 1 ) );
179:         DrawHand( second * 6.0, (int)(0.97 * r_inner), g);
180:      }
181:   }
182:
183:   // ================================================================
184:   // ClockWindow class implements the window containing the clock
185:   // ================================================================
186:
187:   class ClockWindow extends JFrame
188:   {
189:      public ClockWindow()
190:      {
191:         setContentPane( new ClockPane() );
192:         setTitle( "Clock");
193:      }
194:   }
```

Listing 3.1. The example of a vector image animation – the working clock program

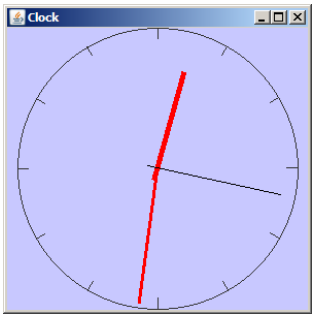


Fig 3.1. The working clock rendered with the program from Listing 3.1.

Assignment scope

1. Extend the clock from Demo3 program with the swinging pendulum. Set the pendulum length to be double of the dial diameter. Model the pendulum with the line segment ended with the round bob. Make possible to set the pendulum movement period and amplitude with the command line parameters.

2. Write the program that displays the animated solar system. Draw the planet's trajectories with circular lines and make them circulating around the Sun. Add the Moon circulating around the Earth.

3. Implement the animation of balls on the snooker table. Assume initial velocities and movement directions of balls. Detect collisions of balls with other balls and with the table edges. Change balls movement directions after collisions appropriately.

4. Implement animation of falling raindrops. Create raindrops at the upper edge of the image randomly. Then trace the created drops in their movement towards the bottom edge of the image. Model individual raindrops with small ellipses. Control the simulated rain intensity with the command line parameter.

Assignment 4
Building simple graphical user interface with Swing - simple interactive line drawing program

Aim

The aim of this assignment is to learn how to capture the mouse and keyboard events in a Java program and how to use then in arranging an interactive vector drawing program. The examples of Swing user interface controls applications in CG programs are also presented here.

Theoretical fundamentals

Capturing mouse and keyboard events

Many CG applications need an user interaction. In 2D graphics the “desktop metaphor” is commonly used. It means that the window looks like a desktop on which various elements are located. The user can change positions of elements, remove them or add a new ones using a mouse as a pointing, catching and dragging tool. In the case of 3D applications the user can interact with elements of 3D scene in a similar way.

In order to implement this style of interaction the software tools are necessary that make possible to pass the information about mouse and keyboard events to the program. In reaction to the even the program executes the appropriate operation that leads to the modification of the displayed image or more generally - to the modification of the image or scene contents.

The method of control device events capturing implemented in Java is based on the concept of a *listener*. The listener is the object that is notified by the external device control system (supported by OS) about occurring events the object is interested in. The listener class must implement specific methods that will be called by the system in response to specific events occurrences. It is achieved by implementing specific Java interfaces within the listener classes. The detailed attributes of the event (e.g. the cursor position in the window pane where the mouse event occurred, the identifier of the clicked mouse button, the symbol of the pressed keyboard key etc.) are provided in an object passed to the listening method. The following interfaces are defined for basic event listeners:

- `ActionListener` - captures events related to UI elements like push buttons, checkboxes, edit fields etc.,
- `MouseListener` - receives notifications of mouse events related to button press/release operations and to crossing the boundary of the sensitivity area (i.e. the area in which mouse events are reported) by the cursor,
- `MouseMotionListener` - tracks the mouse motion and makes possible to find out if a button is pressed while mouse is moved (we call it *dragging* operation),
- `KeyListener` - receives notifications related to keyboard like press/release operations and button clicks.

The class that is intended to be an even listener must implement the appropriate interface. The object from this class must then be registered as a listener of events occurring in the area of the graphical object on the screen. The registration informs the event management system which objects are to be notified about the particular events. The registration can be achieved by the call of a method specific to the type of events:

- **`public void addMouseListener(MouseListener listener)`** - for listening of mouse events not related to a mouse motion,
- **`public void addMouseMotionListener(MouseMotionListener listener)`** - for listening of mouse motion events,
- **`public void addKeyListener(KeyListener listener)`** - for listening of keyboard events

The registration methods are defined in the basic `Component` class, so they are available in UI elements derived (directly or indirectly) from it. It means that the related events can be captured when the cursor is in the area of each of UI elements derived from the `Component` class. Registering the listener `listener` for the UI object `uiobject` using the code as follows:

```
uiobject.addMouseMotionListener( listener );
```

causes that events occurring in the area of `uiobject` on the screen will re reported to the object `listener`.

Any class can be made a listener of mouse or keyboard events. In the typical framework of CG application it is convenient to implement a listener within the same class that implements the window pane, i.e. in a `JPanel` subclass. The following code shows the example of the implementation mouse events handlers implementation within a `JPanel` derived class. For the sake of this example, actions executed in response to mouse event notifications consist just in printing an appropriate message on the console.

```
 1:   import java.awt.event.ActionEvent;
 2:   import java.awt.event.ActionListener;
 3:   import java.awt.event.KeyEvent;
 4:   import java.awt.event.KeyListener;
 5:   import java.awt.event.MouseEvent;
 6:   import java.awt.event.MouseListener;
 7:   import java.awt.event.MouseMotionListener;
 8:   import java.awt.event.*;
 9:
10:   import javax.swing.JPanel;
11:
12:
13:   // This class implements mouse and keyboard listeners MouseListener
14:   // and MouseMotionListener interfaces need to be implemented in order
15:   // to receive notification on mouse events
16:   class DrawWndPane
17:      extends     JPanel
18:      implements  MouseListener, MouseMotionListener, KeyListener
19:   {
20:      DrawWndPane()
21:      {
22:          super();
23:
24:          // Make this object an event listener i.e. register "this"
25:          // object as a listener of mouse and keyboard events.
26:          // Only events occurring in the areas of the component for
```

```
27:          // which the add...Listener method was called will be
28:          // reported to event handlers.
29:          addMouseListener( this );
30:          addMouseMotionListener( this );
31:          addKeyListener( this );
32:       }
33:
34:       // Mouse event handlers:
35:       // ===================================================
36:
37:       // This handler is called when mouse button i clicked
38:       public void mouseClicked(MouseEvent arg0)
39:       {
40:          System.out.println( "mouseClicked at " +
41:                              arg0.getX() + "  " + arg0.getY() );
42:       }
43:
44:       // This handler is called when the mouse enters the window
45:       // pane area
46:       public void mouseEntered(MouseEvent arg0)
47:       {
48:          System.out.println( "mouseEntered at " +
49:                              arg0.getX() + "  " + arg0.getY() );
50:       }
51:
52:       // This handler is called when the mouse exits the window
53:       // pane area
54:       public void mouseExited(MouseEvent arg0)
55:       {
56:          System.out.println( "mouseExited at " +
57:                              arg0.getX() + "  " + arg0.getY() );
58:       }
59:
60:       // This handler is called when mouse button is pressed
61:       public void mousePressed(MouseEvent arg0)
62:       {
63:          String which;
64:
65:          // Get information which mouse button was clicked
66:          if ( arg0.getButton() == MouseEvent.BUTTON1)
67:             which = " Button 1";
68:          else
69:          if ( arg0.getButton() == MouseEvent.BUTTON2 )
70:             which = " Button 2";
71:          else
72:             which = " Button 3";
73:          System.out.println( "mousePressed at " +
74:                              arg0.getX() + "  " + arg0.getY() +
75:                              " " + which + " was pressed" );
76:       }
77:
78:       // This handler is called when the button is released
79:       public void mouseReleased(MouseEvent arg0)
80:       {
81:          System.out.println( "mouseReleased at " +
82:                              arg0.getX() + "  " + arg0.getY() );
83:       }
84:
85:       // This handler is called is the cursor is moved
86:       // in the pane area with a button pressed down
87:       public void mouseDragged(MouseEvent arg0)
88:       {
89:          System.out.println( "mouseDragged at " +
90:                              arg0.getX() + "  " + arg0.getY() );
```

```
 91:      }
 92:
 93:      // This handler is called is the cursor is moved
 94:      // in the pane area with no button pressed down
 95:      public void mouseMoved(MouseEvent arg0)
 96:      {
 97:         System.out.println( "mouseMoved to " +
 98:                             arg0.getX() + "  " + arg0.getY() );
 99:      }
100:
101:      // Keyboard event handlers:
102:      // ================================================
103:
104:      // This handler is called when the keyboard key is pressed
105:      public void keyPressed(KeyEvent e)
106:      {
107:         // Use getKeyCode to get the symbol of the pressed key.
108:         // It can be used to distinguish virtual keys.
109:         // getKeyChar() returns the character associated
110:         // with the key. Use it for "ordinary" character keys only.
111:         System.out.println( "keyPressed " +
112:                             "Key code: " + e.getKeyCode() +
113:                             "Char: "     + e.getKeyChar() );
114:      }
115:
116:      // This handler is called when a key is released
117:      public void keyReleased(KeyEvent e)
118:      {
119:         System.out.println( "keyReleased" );
120:      }
121:
122:      // This handler notifies about the event consisting in
123:      // typing a character. The modifier keys (Shift, Alt) are not
124:      // reported with this handler.
125:      public void keyTyped(KeyEvent e)
126:      {
127:         // Use getKeyChar() to get the character associated
128:         // with the key. Use it for "ordinary" character keys.
129:         System.out.println( "keyTyped " +
130:                             "Char: "   + e.getKeyChar() );
131:      }
132:   }
```

Listing 4.1. The example of mouse and keyboard capturing

Using push buttons and other UI controls

The swing package provides the set of classes that model the most commonly used UI controls like push buttons, edit fields, checkboxes, radio buttons, combo boxes and many others. Each UI control type is modeled by a specific Swing class. Table 4.1. presents class identifiers for popular UI controls. The principles of their usage are similar. Here we will explain basic concepts by using push buttons as examples.

UI control	Swing class
push button	`JButton`
edit field	`JTextField`
formatted text field (e.g. used for numbers, dates etc.)	`JFormattedTexField`
radio button	`JRadioButton`

check box	`JCheckBox`
fixed text area	`JLabel`
text field with list of contents (combo box)	`JComboBox`
slider (the control that can be used to select the value from a fixed interval by positioning the tick mark on the scale)	`JSlider`

Table 4.1. Classes defined in Swing for basic UI controls

In order to add the control to the content pane the following steps need to be executed

```
make a selected object the listener of the events related to used controls
   (e.g. push button clicks, moving focus out of controls) by implementing
   appropriate interfaces;
create the control object;
set its position and size with respect to the including contents pane;
add the UI component to the content pane using add() method;
register listeners of events related to UI elements.
```

As far as our "standard" window architecture is being used, the appropriate place to undertake these actions is the content pane object constructor. It should be noted that Java provides tools that set UI elements layout (position, size) automatically, according to the actual content pane area size and aspect ratio. Here we will rather use fixed layout of elements and therefore the automatic layout functionality is switched off.

The code example in Listing 4.2. explains how to create three push buttons in the content pane and how to capture events related to button clicks. If there are a few push buttons (as in this example) then button click events coming from various buttons have to be distinguished. Sometimes it can be achieved by registering individual listener object for each button. More common practice (also followed in this example) is however to use the common listener for all buttons and to find out which button was the actual source of the event by using the event attributes passed the even handler. The object from `ActionEvent` class is passed the event handler (`actionPerformed()` method of the `ActionListener` interface). It implements `getSource()` method that returns the reference to the object being the event source. Which button was the actual event source can be found out by comparing the result of `getSource()` with stored references to buttons. Pay attention to the solution of this problem in lines 82-93 of the example 4.2.

```
 1:   import javax.swing.*;
 2:   import java.awt.*;
 3:   import java.awt.event.*;
 4:
 5:   public class demo2
 6:   {
 7:       public static void main(String[] args)
 8:       {
 9:           SmpWindow wnd = new SmpWindow();
10:           wnd.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11:           wnd.setVisible(true);
12:           wnd.setBounds( 70, 70, 450, 300);
13:           wnd.setTitle( "Mouse event handling demo" );
14:       }
15:   }
16:
17:   // Implement event handling functions within the same class
18:   // which implements the window content pane
19:   class ButtonPane extends JPanel implements ActionListener
20:   {
21:      // Declare references to three buttons located on the pane
22:      JButton button1;
```

```
23:      JButton button2;
24:      JButton button3;
25:
26:      // The message string to be displayed on the pane
27:      // after each click of a button.
28:      String  message;
29:
30:      // Initialize UI elements and set up event listeners
31:      // within the constructor of the content pane object
32:      ButtonPane()
33:      {
34:          super();
35:
36:          // Switch off automatic components positioning
37:          setLayout( null );
38:
39:          // Create button objects. The button label
40:          // is specified at the constructor parameter
41:          button1 = new JButton( "1" );
42:          button2 = new JButton( "2" );
43:          button3 = new JButton( "3" );
44:
45:          // Set fixed position and size of each button
46:          button1.setBounds( 100, 100, 70, 30 );
47:          button2.setBounds( 190, 100, 70, 30 );
48:          button3.setBounds( 280, 100, 70, 30 );
49:
50:          // Add buttons to the content pane in order to assure
51:          // correct display
52:          add( button1 );
53:          add( button2 );
54:          add( button3 );
55:
56:          // Add listener to allow reacting to clicking -
57:          // The same listener is used by all buttons
58:          button1.addActionListener( this );
59:          button2.addActionListener( this );
60:          button3.addActionListener( this );
61:
62:          message = "No button has been pressed yet";
63:      }
64:
65:      public void paintComponent( Graphics g)
66:      {
67:          super.paintComponent(g);
68:          Graphics2D  g2d = (Graphics2D)g;
69:
70:          // Just draw the message string indicating
71:          // which button was just clicked
72:          g2d.drawString( message, 130, 150);
73:      }
74:
75:      // The button event handler. It implements the method of
76:      // ActionListener interface. It will be called each time one of
77:      // buttons is clicked
78:      // ================================================================
79:      public void actionPerformed( ActionEvent event)
80:      {
81:         // Acquire reference to the object being the event source
82:         Object source = event.getSource();
83:
84:         // Distinguish which button has been clicked
85:         if ( source == button1 )
86:            message = "Button 1 clicked";
```

```
 87:         else
 88:         if ( source == button2 )
 89:            message = "Button 2 clicked";
 90:         else
 91:            message = "Button 3 clicked";
 92:
 93:         // Force window redraw to see the result immediately
 94:         repaint();
 95:      }
 96:   }
 97:
 98:   class SmpWindow extends JFrame
 99:   {
100:       public SmpWindow()
101:       {
102:           // Acquire drawing surface and add own panel to it
103:           // and add the panel containing buttons
104:           Container contents = getContentPane();
105:           contents.add( new ButtonPane() );
106:       }
107:   }
```

Listing 4.2. The example of push button clicks handling

Assignment scope

Implement the simple vector graphics editor. The editor makes possible to create vector images consisting of line segments, rectangles and circles. The required functionality of the editor comprises:

- insertion of new elements to the image (lines, rectangles, circles),
- modification of position of the selected image element,
- deletion of an image element,
- color setting for the selected element,
- saving of the vector image in a text file,
- loading the previously stored vector image so that all vector elements previously saved can be manipulated,
- saving the image area as a raster image.

Apply the following scenarios to particular actions:

Adding of the new line segment:

- click the point that is out of sensitivity area of other already created elements with mouse left button - the new line start point is at the point of click,
- drag the mouse to the point, where the second line end should be positioned - the line connecting the click point with the current cursors position should be displayed instantly (use XOR line drawing mode to avoid the complete pane redisplay),
- release the button at the position of the second line end.

Adding of the circle:

- click at the position of the circle center with the left mouse button,
- drag the mouse to the position on the circle edge (in this way determine the circle radius) - the temporal circle should be redrawn after each mouse movement,
- release the mouse button at the required distance from the circle center.

Adding of a rectangle:

- click at the position of the leftmost rectangle vertex,

- drag to the position of the opposite rectangle vertex - the temporal rectangle should be redrawn after each mouse movement,
- release the mouse.

Which shape is to be added using one of operations described above depends on the shape type selection determined by three radio buttons located in bottom part the window (use individual button for each shape type).

Modification of the line segment end position:
- click with the left button near the end of the line segment - the close line segment end should be selected for modification and attracted to the cursor,
- drag the mouse to the new desired position of the line end - the line in its temporal position should be redrawn after each mouse movement,
- release the mouse button.

Translation of the whole shape:
- click near the center of the shape (line, rectangle, circle) with the left mouse button,
- drag the shape center to the new position - the shape in its temporal position should be redrawn after each mouse movement,
- release the button at the desired shape center position.

Deletion of the shape:
- click near to the shape center with the right mouse button

The vector image store and load operation as well as the operation of the raster image writing to the graphic file should be invoked by appropriate push buttons. Use three edit fields for RGB values of the current color. The new shape should be created in the current color set by RGB values in these fields.

Assignment 5
Homogenous transformations in 2D

Aim

The aim of this assignment is to get acquainted with the concept of homogenous coordinates, learn how to express various 2D transformations by matrices in homogenous coordinates, how to use it in practical applications and how to utilize Java2D API components that support affine transformations.

Theoretical fundamentals

Algebra of affine transforms in homogenous coordinates

In many applications, 2D images need to be transformed so as to fit to desired positions. A desktop publishing application can be an example, where many images are combined on a single sheet of paper in various positions, scales and orientations as shown in Fig 5.1.

Fig. 5.1. The example of image compositions by translating rotating and scaling

2D image transformation can be consider as a mapping T of infinite input R^2 image plane onto the infinite output R^2 image plane: $T : R^2 \rightarrow R^2$. The mapping is defined for the whole plane. In the case of raster images we are interested only in the rectangular subarea of the output image of the size defined by the output image resolution. In order to create the output raster image that is a transformation of the input one, for each pixel of the output image we need to find the corresponding position in the input image and compute the pixel attributes of this pixel appropriately, using the attributes of the corresponding pixel in the input image. In the case of a vector image, the geometric primitives constituting the input image are transformed using T transformation to the output image space.

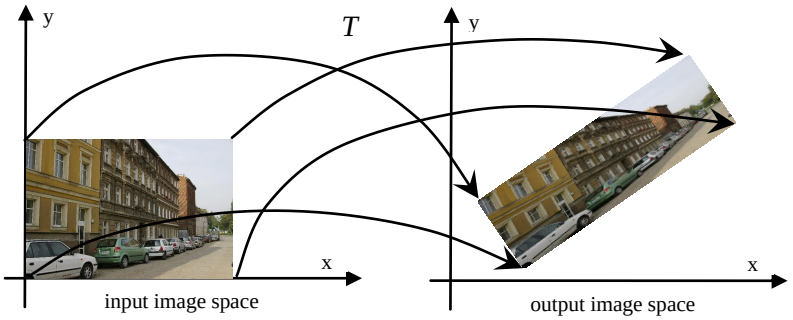


Fig. 5.2. The example of a raster image transformation

For many practical applications it is sufficient to assume that the transformation applied to images belongs to the family of affine transformations. The affine transformation is defined as a transformation that preserves collinearity of points. In result, a line in mapped onto a line by the affine transformation. If the image points p in R^2 are represented by column vectors, i.e. $p = (x, y)^T$ then the affine transform is defined by the formula:

$$p' = Ap + B, \tag{5.1}$$

where A is *2x2* matrix and B is a column vector in R^2. We will be interested in affine transformations that are combinations of *elementary transformations*: scaling along axes, parallel translation and rotation around *(0,0)* point. The transformations applied to images will be obtained as combinations of elementary transformations. Unfortunately the algebraic representation of the affine transformation as in the formula (5.1) is not convenient from the computational point of view. We would rather need such representation of transformations that the composition of two simpler transformations can be obtained just by multiplication of matrices. Simple scaling can be obviously represented by a scaling matrix:

$$A = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{5.2}$$

Also the rotation matrix for the rotation by the angle α can be derived:

$$A = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{5.3}$$

Unfortunately, translation cannot be represented just by the translation matrix in natural coordinates system. In order to be able to represent all transformations in a uniform way, *homogenous coordinates* are introduced. Homogeneous coordinates are obtained from natural coordinates by extending the point vector $p = (x, y)^T$ with the third coordinate *w*: $p_H = (x', y', w)^T$. The *w* coordinate is a kind of scaling factor. In order to obtain natural coordinates of the point represented by the homogenous coordinates $(x', y', w)^T$ x' and y' coordinates should be divided by *w*, i.e.

$$p = (x, y) : x = \frac{x'}{w}, y = \frac{y'}{w}. \tag{5.4}$$

Formally, due to interpretation of *w* coordinate, it should be nonzero. Observe however that by decreasing the absolute value of *w* so that it converges to 0, we obtain points located on the line $y = \frac{y'}{x'}x$. The smaller is the *w* coordinate, the longer is the distance of the point *p* to the coordinates center *(0,0)*. Therefore, sometimes the point in homogeneous coordinates $(x', y', 0)$ is used to represent the point in infinity located in the direction y'/x'. The representation of a 2D point in homogenous coordinates is not unique. Each 2D point has infinite numbers of representations in homogenous coordinates. The homogenous vectors $(5,2,1)$, $(10,4,2)$, $(50,20,10)$ all represent the same point $(5,2)$ on 2D plane. In homogenous coordinates the translation by the vector (t_x, t_y) can be defined by the translation matrix:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

Indeed, by multiplying the matrix (5.5) by the point vector (x', y', w) we obtain:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} `x'+wt_x \\ y'+wt_y \\ w \end{bmatrix} \tag{5.7}$$

By converting the homogenous point being the multiplication product to natural 2D coordinates, we obtain $(\frac{x'}{w}+t_x, \frac{y'}{w}+t_y)$ what corresponds to the translation by the vector (t_x, t_y). The rotation and scaling matrices in homogenous coordinates can be obtained by simple extension of matrices (5.2) and (5.3). Hence, the transformation matrices for all elementary transformations are defined as follows:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.8}$$

a) for scaling, b) for translation, c) for rotation.

Now any combination of elementary transformations can be described by just single transformation matrix. Let us consider the transformation *T* that is the combination of simpler transformations T_1 and T_2 defined by two transformation matrices M_1, M_2 correspondingly. It means that the point being transformed is first transformed using T_1, what results in the temporary point $P_1 = M_1 P$. Then the temporary point P_1 is subject to

T_2 transformation resulting in the ultimately transformed point $P_2 = M_2 P_1$. By substituting P_1 in the latter equation we get:

$$P_2 = M_2 P_1 = M_2 (M_1 P) = (M_2 M_1) P = MP \ , \tag{5.9}$$

where $M=M_2M_1$ is the matrix of the compound transformation obtained by multiplying transformation matrices of the component transformations. The order of matrices in the product depends on how the homogenous point is represented as the vector. Here we assumed that the point is represented by a column vector $(x', y', w)^T$. Therefore if the matrix M is the product of simpler transformation matrices $M_1,M_2,...,M_n$ then multiplying the point P by such a matrix:

$$P' = MP = (M_1 M_2, ... M_{n-2} M_{n-1} M_n) P = (M_1 M_2, ..., (M_{n-2} (M_{n-1} (M_n P))) ...)$$

corresponds to the application of transformations in reversed order, i.e. the transformation represented by M_n is applied as the first one, while the transformation corresponding to M_1 is the last one. The logical order of transformations will be opposite if the point is represented by the row vector (x', y', w) and the transformation is denoted as:

$$P' = PM = P(M_1 M_2, ... M_{n-2} M_{n-1} M_n) = ((...((PM_1) M_2), ..., M_{n-2}) M_{n-1}) M$$

If all homogenous matrix elements are multiplied by the same nonzero constant then the transformation defined by the matrix is not changed. This is because the scaling component w is multiplied by the same factor than remaining components of multiplication product vector. In result the vector represents the same 2D point. For this reason, it is convenient to store the matrix in the normalized form where the coefficient m_{22} (lower-right element of the matrix) is equal to one.

As an example let us consider the representation of 2D homothety transformation by the transformation matrix in homogeneous coordinates.

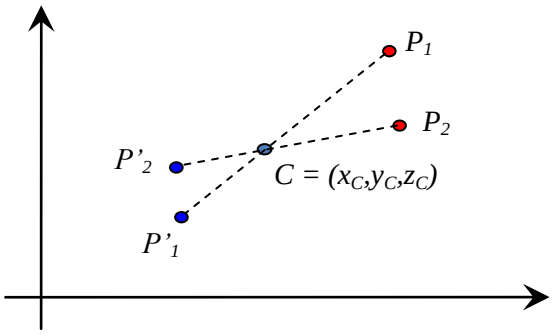


Fig. 5.3. The homothetic transformation with the center C and scaling factor $s = -0.6$

The homothety is specified by the homothety center A (being the point in 2D) and the scaling factor s. The homothety maps each 2D point P onto the point P' according to the formula:

$$P' = C + s(P - C) \, . \tag{5.10}$$

For the point being mapped, the vector connecting it with the center point C is determined. Then the vector is scaled by the scale factor c and the scaled vector is used to translate the

point *C*. Points *P*, *P'* and *C* are always collinear. The homothetic transformation is shown in Fig. 5.3. The transformation matrix for homothety can be obtained either by analyzing the formula (5.10) that defines the transformation or by decomposing it to series of elementary transformations. By rewriting the formula in scalar form we obtain the equations for 2D coordinates of the transformed point:

$$\begin{aligned} P' &= C + s(P - C) = P + (1-s)C \\ x' &= x_C + s(x - x_C) = sx + (1-s)x_C \\ y' &= y_C + s(y - y_C) = sy + (1-s)y_C \end{aligned} \quad . \tag{5.11}$$

It is clear that the above equations correspond to scaling and to translation. Therefore the transformation matrix can be constructed as follows:

$$\begin{bmatrix} s & 0 & (1-s)x_C \\ 0 & s & (1-s)y_C \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.12}$$

The same result can be obtained by observing that the homotety can be obtained as a combination of three elementary transformations:

- translation by the vector *-A*,
- scaling by scaling factors $s_x=s_y=s$,
- reverse translation by the vector *A*.

Hence, the transformation matrix can be obtained as the product of corresponding elementary transformation matrices:

$$\begin{bmatrix} 1 & 0 & x_C \\ 0 & 1 & y_C \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -x_C \\ 0 & 1 & -y_C \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s & 0 & (1-s)x_C \\ 0 & s & (1-s)y_C \\ 0 & 0 & 1 \end{bmatrix} \tag{5.13}$$

Technical aspects of homogenous coordinates

The ability to consistently represent any transformation being the superposition of scaling, translation and rotation and to unify the computations related to transformation composition is of crucial significance for the efficiency of many CG applications, both for 2D and 3D graphics. The operations necessary to combine transformations and to transform points are especially appropriate for hardware implementation due to their logical simplicity. The main advantages of homogenous coordinates related to their application in CG code or in hardware supported implementation are as follows:

- unification of transformation algorithm – the method of coordinate calculation does not depend on the transformation itself and can be implemented with common, simple algorithm consisting just in multiplication of the matrix and vector; it can be easily implemented as a specialized fast hardware module,
- simplicity of transformation composition - any superposition of transformations can be represented by a single matrix of compound transformation that can be created by simple matrix multiplication,

- savings in transforming large sets of points – single transformation matrix is calculated only once and then applied to many points (instead of multiplying each point by many matrices corresponding to transformation stages),
- calculations can be efficiently implemented in hardware (both for 2D and 3D graphics),
- necessary calculations can be executed in parallel – in this way significant speed-up can be achieved, especially in 3D hardware.

Affine transformation support in Java2D

Java2D supports affine transformations by providing `AffineTransform` class that encapsulates the homogenous transformation matrix for 2D and the set of useful utilities for constructing affine transforms. The `AffineTransform` class implements matrix by vector multiplication assuming that the vector is in column form. The object representing appropriately defined affine transformation can be then used to transform 2D shapes that can be derived from `Shape` class. Alternatively, it can be directly applied to the drawing context of a raster image or to a window content pane. In the latter case, all drawing operations executed through the drawing context are subject to the transformation currently bound to it. The most useful methods of `AffineTransform` class are:

Constructors:

`public AffineTransform()`
The default constructor creates the matrix corresponding to the identity transform defined by unit transformation matrix.

```
public AffineTransform(
    double m00, double m10,
    double m01, double m11,
    double m02, double m12)
```

This constructor creates the transformation matrix containing explicitly specified elements. The elements are passed in column-by column order, i.e. the created matrix is arranged according to the following pattern:

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix}$$

`public AffineTransform(double[] m)`
This constructor creates the transformation matrix by copying 4 or 6 matrix elements of the passed array depending on its actual size. The order of elements is: m_{00}, m_{10}, m_{01}, m_{11}, m_{02}, m_{12}.

`public AffineTransform(AffineTransform Mtx)`
The constructor creates the new transformation matrix by copying elements of another `AffineTransform` object passed as a parameter.

Methods that define the transformation matrix for specified elementary transform:

These methods arrange transformation matrices within `AffineTransform` object so as they correspond to the elementary transformation specified by the method name. The contents of the matrix at the moment of the utility call do not affect the final result.

public void setToScale(double sx, double sy)
This method sets the scaling matrix with scaling factors `sx, sy`.

public void setToRotation(double alpha)
public void setToRotation(double alpha, double x, double y)

The first method sets the rotation matrix that rotates points around the center of coordinates system. The second version of the method creates the matrix that rotates around the point (`x,y`). It is equivalent to concatenation of: translation, rotation around (0,0) point and translation in the opposite direction. The rotation angle in radians is determined by the parameter `alpha`.

public void setToTranslation(double tx, double ty)
Sets the translation matrix where the translation vector is (`tx, ty`).

public void setToIdentity()
Sets the identity transformation matrix.

Methods that define the transformation matrix for compositions of transformations:

The methods from this group multiply the existing transformation matrix by the matrix created for the elementary transformation. The elementary transformation matrix is used as the right operand in matrix multiplication operation. Therefore the last applied method defines the first transformation that will be applied to the point being transformed.

public void scale(double sx,double sy)
This utility multiplies the matrix represented by the object by the scaling matrix with scaling factors `(sx, sy)`.

public void translate(double tx, double ty)
This utility multiplies the matrix represented by the object by the translation matrix for the translation vector `(tx, ty)`.

public void rotate(double alpha)
public void rotate(double alpha, double x, double y)
The first method multiplies the existing matrix by the rotation matrix that rotates around the point (0,0). The second variant implements rotation around the point (`x,y`). Rotation angle is specified by `alpha` parameter.

public void concatenate(AffineTransform Mtx)
This utility multiplies the matrix of the target object by the matrix of the object passed as a parameter. The matrix of the object passed as a parameter is the right operand of the multiplication.

Application of `AffineTransform` objects in image geometry transformations

There are two modes in which the `AffineTransform` object can be used when creating a vector or raster image. The first mode can be applied if the image is constructed with 2D shapes derived from `Shape` superclass (lines, polylines, rectangles ellipses - see assignment 2 for details). The method:

public Shape createTransformedShape(Shape org_shape)

of `AffineTransform` class creates the new shape which is the result of application of the transformation defined by object to the shape passed as the parameter `org_shape`. The created shape can be further manipulated or displayed in the ordinary way. The following fragment of code creates the axis aligned rectangle and then creates the new shape which is the same rectangle rotated by 45 degrees. Both rectangles are then displayed. The primary rectangle is drawn with black pen. The transformed shape is drawn with blue pen.

```
Rectangle2D.Double  rect;
Shape               rotated_rect;
AffineTransform     transform;

// Create axis aligned rectangle
rect = new Rectangle2D.Double(100, 100, 60, 90);
// Draw it with default (black) pen
g2d.draw( rect );

// Set affine transformation to rotation by 45 degrees
// use upper left rectangle vertex as the rotation center
transform.setToRotation( Math.PI/4.0, 100, 100 );
rotated_rect = transform.createTransformedShape( rect );

// Set the pen color to blue and display the rotated rectangle
g2d.setColor( new Color( 0, 0, 255 ) );
g2d.draw( rotated_rect );
```

The alternative method is to assign the transformation to the drawing context. All drawing operations executed with this context will be then subject to the bound affine transformation. In order to bind the affine transform matrix to the drawing context use the method:

public void setTransform(AffineTransform Mtx)

defined in `Graphics2D` class. It replaces the current transformation matrix with the new one. This transformation matrix will be applied to all subsequent drawing operations, including activities related to the rendering of UI controls. Therefore, after the drawing sequence is completed, the default transformation matrix of the drawing context should be restored. The transformation matrix assigned to the context can be obtained with the method:

public AffineTransform getTransform()

also defined in the `Graphics2D` class. The example below shows how to manipulate the transformation assigned to the drawing context so as to display the raster image rotated by the angle of 45 degrees. It is assumed that the image to be displayed is already loaded into the `BufferedImage` object referenced by `image` variable. The whole code in contained in `paintComonent()` method.

```
1:   public void paintComponent( Graphics g)
2:   {
3:      Graphics2D  g2d = (Graphics2D)g;
4:
5:      AffineTransform   transform;
6:      AffineTransform   def_transform;
7:
8:      // Get the reference to default drawin gontext transform
```

```
 9:      def_transform = g2d.getTransform();
10:
11:      // Paint the image in a axis aligned rectangle
12:      g2d.drawImage( input_image, 150, 50, 150, 150, null );
13:
14:      // Prapare rotation matrix, rotation center at the upper left
15:      // vertex of displayed image rectangle
16:      transform = new AffineTransform();
17:      transform.setToRotation( 3.1415/4, 150, 50 );
18:
19:      // Bind affine transformation to drawing context
20:      g2d.setTransform( transform );
21:
22:      // Paint the rotated image again
23:      g2d.drawImage( input_image, 150, 50, 150, 150, null );
24:
25:      // Add translation to the transformation
26:      transform.translate( 80, 80 );
27:      g2d.setTransform( transform );
28:
29:      // Paint the rotated and translated image one more time
30:      g2d.drawImage( input_image, 150, 50, 150, 150, null );
31:
32:      // Restore the dafault transformation to prevent correct
33:      // display of other window elements
34:      g2d.setTransform( def_transform );
35:   }
```

Listing 5.1. The example of `AffineTransform` application to raster image display

Assignment scope

Implement the program that supports a poster composition. Raster images, texts, rectangles and circles filled with colors can be placed on the poster surface. The program should be implemented as an interactive application with four panels. The upper left panel displays miniatures of images loaded from the directory. The images can be dragged onto the surface of the poster. The lower left panel contains the gallery of 2D shapes that can be added to the poster (squares and circles only). The right part of the screen is occupied by the work area, where the poster is being assembled. The user can drag any image miniature or a shape from the gallery onto the poster surface. The dropped element is displayed at the drop position. By dragging centers and vertices of the elements in the work area the user can modify the poster layout. Dragging of the vertex causes that the element size is rotated or resized. Dragging of centers causes that the element is translated. Precise layout correction is implemented by pushbutton actions. The set of pushbuttons is located in the bottom panel. The following fine tuning operations should be implemented with pushbuttons:

– translate to left/right/top/bottom by single pixel
– rotate left/right by small angle

The program should make it possible to create images similar to this one presented in Fig. 5.1. Use individual `AffineTransform` objects to determine the actual position and size of each poster element with respect to the poster rectangle. Apply `rotate()` and `translate()` methods that accumulate transformations in the case of fine tuning with pushbuttons. Apply methods that define transformations from scratch in the case of elements positioning with drag and drop operations.

Implement the following functionalities of the program:

- insertion of the new element to the poster with drag and drop operation,
- modification of poster element layout (position, size, rotation) by dragging of poster element centers and vertices,
- deletion of poster elements with right mouse button click,
- change of the elements order (bring to top, push to bottom),
- fine tuning of an element position and rotation angle with pushbuttons,
- store/load operations of all affine transforms, so as all results of work with a poster can be stored in the text file and then retrieved,
- the ability to store the image in a graphic file of arbitrary resolution (not limited by the actual screen window resolution).

Assignment 6
Bilinear interpolation and Gouraud shading of 2D triangles

Aim

The aim of this assignment is to get acquainted with bilinear interpolation technique and its application in 2D and 3D graphics. The concept of Gouraud shading is also explained here. By implementing Gouraud shading entirely in software the students can estimate the attainable shading rate and compare it with the shading rate that can be obtained with graphic hardware accelerators.

Theoretical fundamentals

Origins of bi-linear interpolation

Linear and bi-linear interpolation techniques have many important applications in 2D and 3D computer graphics. In 2D graphics they are used in image transformations (in particular in image resizing). In 3D graphics, the bi-linear interpolation concept is most commonly used in *Gouraud shading*. In 3D graphics the term "shading" denotes the process of filling the visible surface with colors or shades of gray according to local lighting conditions. Although the concept of Gouraud shading was originally introduced by Henri Gouraud ([2]) as a color interpolation method in 3D, it is essentially 2D technique and moreover, can be applying to any attribute, not necessarily to the color. We will use here the wider meaning of "shading" and we will call with this word the process of shape filling with colors. The shading is *smooth* if it avoids abrupt changes of visual attributes between adjacent areas of shapes being shaded.

Bilinear interpolation in 2D image resizing

Let us consider the procedure of 2D image resizing, in particular such a case that results in significant increase of the image resolution. Let the input image resolution is (x^I_{res}, y^I_{res}) and the required output image resolution is (x^O_{res}, y^O_{res}).

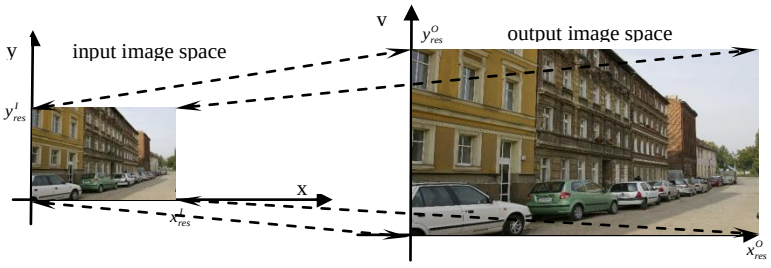


Fig. 6.1. Image resize as a particular case of affine transformation

The resizing is a particularly simple case of geometric affine transformation corresponding to scaling. For the sake of scaling algorithm we will apply, it will be easier to define the transformation that maps the rectangle of the output image onto the input image rectangle. It corresponds to affine scaling transformation where the scaling factors are: $s_x = \frac{x_{res}^I - 1}{x_{res}^O - 1}, s_y = \frac{y_{res}^I - 1}{y_{res}^O - 1}$. The procedure of resided image creation browses the output image rectangle pixel by pixel and for each visited pixels it finds the corresponding position in the input image by applying the scaling transformation.

```
load the input image;
initialize the output image to the desired resolution;

// for all pixels of the output image
for (y = 0; y < y_res; y++)
   for (x = 0; x < x_res; x++)
   {
     find the corresponding position p=(x',y') in input image
        x' = x*sx; y' = y*sy;
     calculate the color of the pixel (x,y) using colors
        of pixels in the vicinity of (x',y') in the input image;
     put calculated pixel color to the output image raster array;
   }
```

The only problem that needs to be solved is how to find the color of the output image pixel. In most cases application of formulas: $x' = xs_x$; $y' = ys_y$ lead to (x',y') coordinates that are not integers and therefore does not determine the exact pixel centers in the input image. The point $p' = (x', y')$ is usually located inside the square defined by adjacent pixel centers in the input image. Bi-linear interpolation is the method in which the final pixel color of the output image pixel is calculated using colors of four adjacent pixels in the input image. The color changes smoothly when the point p' is moved between adjacent pixels in the input image. In result, the visual attributes continuity is preserved in the output image in the case of large magnification of the image being resized.

Let: $j = (\text{int})x'; i = (\text{int})y'$ denote the indices of the upper left pixel center close to the point p'. Let A,B,C,D denote colors of pixels surrounding the point p' in the input image, i.e.:

- A is the color or the pixel (i,j),
- B is the color of the pixel $(i,j+1)$,
- C is the color of the pixel $(i+1,j)$ and
- D is the color of the pixel $(i+1,j+1)$.

We will use two interpolation coefficients: $\alpha = x' - j; \alpha \in (0,1)$ is the interpolation coefficient along x axis, $\beta = y' - i; \beta \in (0,1)$ is the interpolation coefficient along y axis. The procedure first interpolates linearly between A and B as well as between C and D using α coefficient. Two auxiliary colors X_{AB} and X_{CD} are calculated:

$$X_{AB} = \alpha B + (1 - \alpha)A, \quad X_{CD} = \alpha D + (1 - \alpha)C \tag{6.1}$$

Then the final color is calculated by interpolating between results of the first stage interpolation using β coefficient:

$$X = \beta X_{CD} + (1 - \beta) X_{AB} . \quad (6.2)$$

The concept of the bi-linear interpolation in image space is shown in Fig. 6.2.

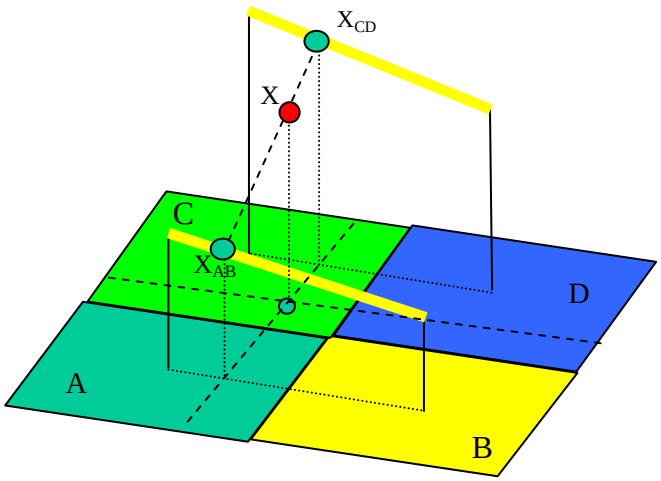


Fig 6.2. Bilinear interpolation between adjacent pixels in the input image

Interpolated shading of triangles

The similar concept can be applied to 2D triangle shading. Let us consider a triangle on the plane. For each vertex its location on the plane and its attribute vector is specified. Any numerical attribute can be processed but we will assume that the attribute is a vertex color. The aim of the interpolated shading is to fill the triangle area with colors so that the colors change gradually from pixel to pixel and that colors at vertices are equal to explicitly specified colors. In a result of interpolated shading the triangle image as shown in Fig. 6.3. should be rendered. The method applied here was proposed by Gouraud ([2]) as a method of shading of a 3D polygon projection onto the projection plane. The proposed concept can be however in any application where smooth shading of a triangle on the plane is required.

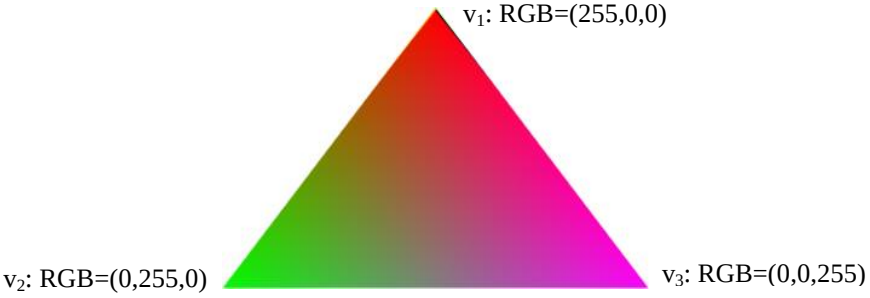


Fig. 6.3. Gouraud-shaded triangle on 2D plane

For the sake of simplicity initially assume that the triangle is shaped as in Fig. 6.3., i.e. bottom edge of the triangle is horizontal. Let the triangle be specified by positions and attributes of its vertices: $v_1=(x_1,y_2,a_1)$, $v_2=(x_2,y_2,a_2)$, $v_3=(x_3,y_3,a_3)$, where v_1 is the topmost

vertex, v_2 is the bottom left vertex and v_3 is the bottom right one. The general idea of interpolated shading is outlined in Fig. 6.4.

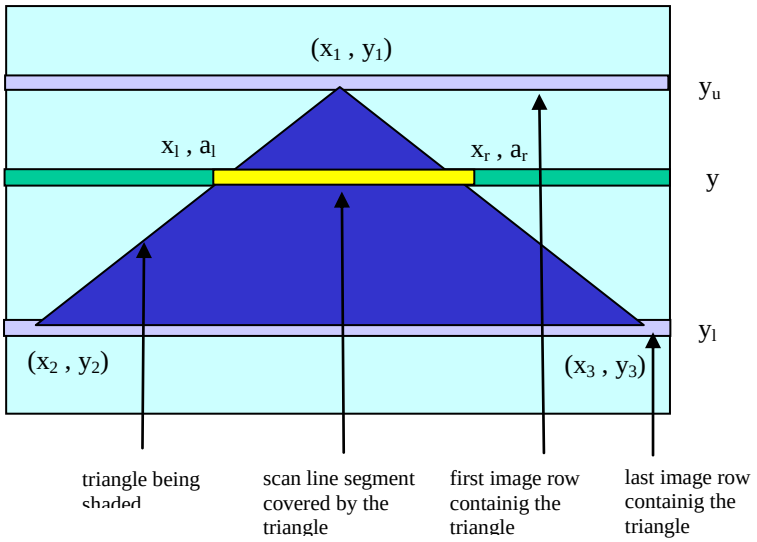


Fig. 6.4. Interpolated Gouraud shading of a triangle

The interpolated Gouraud shading processes the triangle by scanning it with horizontal scan lines (rows of the raster image in which the triangle is being rendered). The subset of scan lines covering the triangle is defined by line index range $<y_u,y_l>$. The scan lines set that covers the triangle is processed in ordered fashion from top to bottom. In each scan line its segment that covers the triangle is determined by x coordinates x_l, and x_r the leftmost and rightmost pixel coordinates of the covering segment can be quickly found with linear interpolation. Let β denotes the interpolation coefficient based on y coordinate of the scan line currently being processed:

$$\beta_y = \frac{y - y_u}{y_l - y_u}. \tag{6.3}$$

The end point x coordinates can be computed as:

$$\begin{aligned} x_l &= \beta_y x_1 + (1-\beta_y)x_2 \\ x_r &= \beta_y x_1 + (1-\beta_y)x_3 \end{aligned} \tag{6.4}$$

The attributes a_l and a_r at pixels (x_l,y) and (x_r,y) can be calculated in analogous way by interpolations between a_1 and a_2 on the left edge and between a_1 and a_3 on the right edge:

$$\begin{aligned} a_l &= \beta_y a_1 + (1-\beta_y)a_2 \\ a_r &= \beta_y a_1 + (1-\beta_y)a_3 \end{aligned} \tag{6.5}$$

Having calculated x coordinates and attributes at the ends of the line segment covering the triangle, it can be filled with attribute values by applying linear interpolation again. This time linear interpolation will be applied to all pixels $(x_i, y) : x_l \le i \le x_r$. Now the interpolation coefficient is based on x coordinate of the pixel being shaded:

$$\alpha = \frac{x - x_l}{x_r - x_l}, \tag{6.6}$$

and the attribute at the current pixel *(x,y)* can be calculated as

$$a(x, y) = \beta_x a_l + (1 - \beta_x) a_r . \tag{6.7}$$

The same result can be obtained with less arithmetic operations per pixel. Instead of combining a_l and a_r at each pixel with two multiplications, one addition and one subtraction, it can be achieved with only one addition. Before pixel interpolation in the current scan line begins, calculate the attribute increment: $\Delta = \frac{a_r - a_l}{x_r - x_l}$. Start with current attribute equal to a_l and add Δ to it in every iteration of the loop that fills pixels in the scan line. The complete interpolated shading algorithm outline is summarized below:

```
find indices yu, yl of upper and lower image rows covered by a triangle;
for  y = yu to yl step 1
   find columns xl, xr containing ends of the scan line segment covered by
      the triangle on the image;
   calculate attributes al, ar for terminal pixels of the scan line segment;
   for x = xl to xr step 1
      calculate attribute of pixel p[y,x] by interpolating between al and ar
      put calculated attribute into the image;
```

In case of arbitrarily shaped triangle it can be divided into two triangles having one horizontal edge. The split line is the horizontal scan line containing the vertex of intermediate y coordinate as shown in Fig. 6.5. The attribute and x coordinate of the additional vertex v_3 can be found by interpolating between v_1 and v_L using the interpolation coefficient β calculated according to the formula 6.3 where $y=y_2$. The described above procedure can be directly applied to the upper triangle. For lower triangle use vertex v_L in place of v_1.

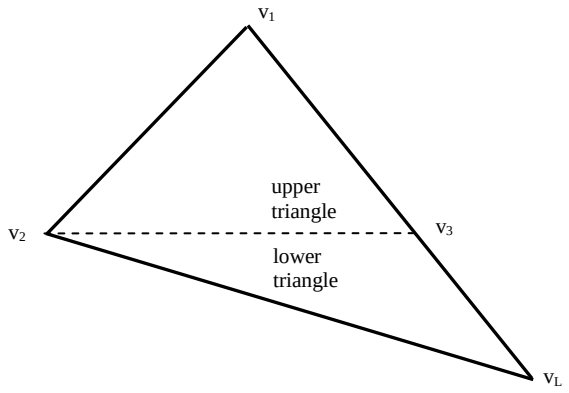


Fig 6.5. Gouraud shading of arbitrarily shaped triangle

Assignment scope

1. Write the program for image resizing with bilinear interpolation. Names of input and output files and the required resolution of the output image are command line arguments. The input file is read from the graphic file, the image is then resized and the

resized image is stored in another graphic file. The image does not have to be displayed on the screen. Optimize your implementation for speed. Compare the execution time necessary to obtain a big image (e.g. of the resolution 3000x2000) with the time necessary to perform the analogous operation in commercial or popular freeware or open source programs. If significant differences occur - try to explain reasons.

2. Write the program for nonlinear deformations of images. It reads an image from the graphic file and creates the output image of the same resolution. Use nonlinear functions that map the output image rectangle onto the input image rectangle. Make sure that image edges in the output image are mapped onto the corresponding edges in the input image. The mapping can be defined so that x and y coordinate transformations are independent, i.e. x coordinate in the input image depends only on the x coordinate in the output image and y coordinate in the input image depends only on the y coordinate in the output image. Apply the same general algorithm as proposed for image resizing. Use bilinear interpolation to assure smooth transitions of colors in the enlarged image.

3. Implement the class representing a 2D triangle. Create fields that contain vertex coordinates and colors. Implement the method that performs Gouraud interpolated shading of the triangle:
   - in the `BufferedImage` passed to the method as an argument,
   - in the window on the screen.

   Test the implementation using manually defined triangles of different shapes. Do not use any third party components that implement triangle shading, instead use your own code.

4. Use the class implemented in the previous step in the program that paints randomly created triangles in buffered image and directly on the screen. Measure the display rate in triangles/sec and in shaded pixels/sec. Compare the results with shading rates achievable with modern graphic boards.

Assignment 7
Simple rendering with Phong lighting model

Aim

The aim of this assignment is to precisely understand the Phong lighting model and its limitations, to evaluate its practical usefulness, find out which optical phenomena simulation it can be applied to and to experiment with modeling of various materials with Phong lighting model.

Theoretical fundamentals

3D scene visualization basics

Visual perception of surrounding world by humans is a result of light interaction with visible objects. The light emitted by primary light sources illuminates the observed scene. The fraction of light falling onto the object surface is re-emitted (reflected) in the observer direction. It passes through the pupil of the eye and finally falls onto the retina which is a kind of "light sensor". The distribution of light intensity on the retina defines the perceived image. In CG in order to obtain the synthetic image of the virtual scene, the process of light transportation from light sources to the image sensor needs to be modeled. The procedure of 3D scene image synthesis is called *rendering*. All 3D rendering techniques developed in CG perform this modeling in a more or less precise way, depending on the required level or realism that should be achieved. In order to render the very simplified view of the virtual scene it is sufficient to model the geometry of directly visible objects viewing. Fig. 7.1.a. presents an extremely simplified scene image displayed in the mode called *wireframe*. In most cases however a more realistic image is expected, where the visible object surface fragments are filled with naturally looking colors. Fig. 7.1.b. shows the view of the same scene created using more sophisticated image synthesis methods.

Fig. 7.1. Simple scene: a) displayed in wireframe mode b) rendered using advanced lighting simulation

Phong lighting model

The key element of realistic rendering is precise modeling of light transport. In the simplest case, the light path consists just of two segments: from the light source to the observed fragment of a surface, and from the observed surface fragment to the observer's eye. In more advanced rendering techniques much more complex light paths are simulated. Complex optical phenomena like specular reflection on mirror surfaces, light refraction on transparent surfaces and light scattering on diffuse surfaces are taken into account. Advanced simulation techniques as backward ray tracing ([1],[4],[14]), forward Monte Carlo ray tracing ([15]) or radiosity method for multiple light scattering simulation ([7]) are able to render excellent images almost indistinguishable from real world photos. Each of the mentioned above methods apply the model of light interaction with the surface. The light-surface interaction is described by *lighting model* (called also *illumination model*). The lighting model is the recipe (formula, function, procedure) that makes possible to compute light intensity emitted from the surface fragment in the specified direction (to the observer) taking into account local lighting conditions and local surface properties related to light reflection. Depending on the applied rendering algorithm we may be interested in the result expressed in photometric quantities or we may be interested just in the observed color of the surface. We will use here RGB color space. Hence, the lighting model calculates light intensities for R, G and B components independently. In order to render a raster image of the scene, two questions have to be answered at each pixel:

- which scene object surface fragment (if any) is visible through the pixel,
- how is it perceived by the observer located at the specified position.

The first question is answered by the rendering stage called *visibility analysis*. In this assignment we deal only with the second problem of determining the perceived color at the particular pixel of the raster image. We assume that the visible surface fragment corresponding to the pixel of the image is already determined. In order to compute the observed surface color the following data are necessary:

- the description of the geometry of the fragment being observed - its location and surface orientation defined by the direction of the vector perpendicular to the surface (*surface normal vector*) ; it can be found if the complete description of all scene object geometry is given and the observer parameters (observer location in scene coordinates, viewing direction, field of view, rendered image resolution) are determined,
- the specification of light sources that illuminate the observed surface,
- surface properties related to the phenomena of light reflection on the surface,
- the location from which the surface fragment is observed.

The elements used in lighting computations are shown in Fig. 7.2.

The common simplification applied in simple CG rendering methods is that we consider only *point primary light sources*. The light source is the point light source if it is approximated by infinitely small point emitting the light located in the scene space. Only primary light sources will be considered. It means the all light sources are able to emit light independently on its possible illumination by other lights. Typically, primary light sources transform other kinds of energy (electricity, chemical energy) into the light. In reality, each illuminated surface reflects a fraction of light falling on it and in result it illuminates other objects with reflected light. Such surface becomes *secondary light source*. If the lighting model takes into account only the illumination from primary lights, then it is called *local lighting model*. If secondary light sources are considered when computing illumination, the lighting model is called *global lighting model*. Simulation of light transport with global

lighting model is a computationally complex task and this phenomenon will not be taken into account here. Here we will apply only local lighting model.

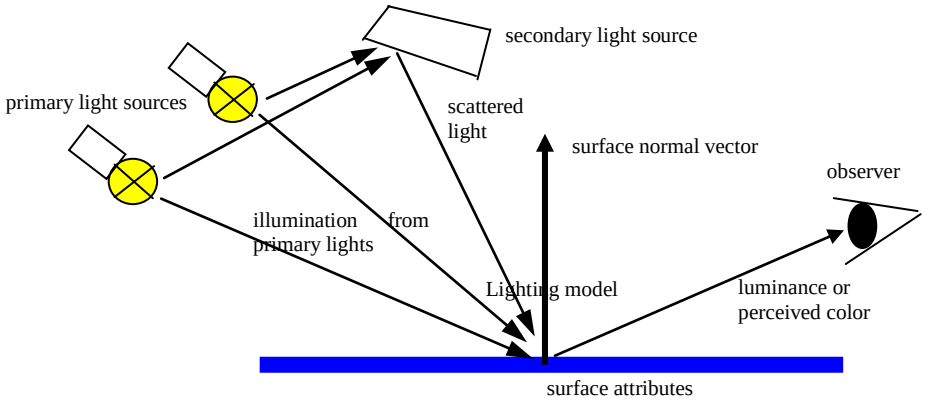


Fig. 7.2. Elements used in lighting computations

Many lighting models were proposed for usage in 3D image rendering. One of most popular lighting models comes from B.T. Phong ([3]).

Despite its simplicity, the Phong model is able to render quite realistic views on relatively simple surfaces like dull materials such as concrete, plaster, paper and on glossy surfaces lsuch as polished plastic.

The Phong model reproduces the most typical optical phenomena: a) diffuse reflection on Lambertian surfaces, b) specular reflection of primary lights on glossy surfaces, c) diffuse reflection of *ambient light* and d) surface self-luminosity. In order to compute the observed color, the reflected light intensity needs to be computed for three RGB components of used color model, based on surface abilities to reflect light by means specific to the reproduced phenomena. Therefore the surface properties will be specified independently for three RGB color components. The following set of properties will be used in Phong model:

- diffuse reflection coefficients: k_{dR}, k_{dG}, k_{dB},
- specular reflection coefficients: k_{sR}, k_{sG}, k_{sB},
- glossiness (shininess) coefficient: g,
- ambient light diffuse reflection coefficients: k_{aR}, k_{aG}, k_{aB},
- self luminance: S_R, S_G, S_B.

The ambient light is the substitute of scattered light component that significantly contributes to the realism of rendered scenes. It is the artificial light that uniformly illuminates all scene surfaces, independently on their locations and orientations. Without the ambient light component, all surfaces that are not directly illuminated by primary lights look unnaturally dark. In real world, surfaces are illuminated also indirectly by secondary light sources. In the local illumination model implemented by Phong formula, secondary lights are substituted by the uniform term of ambient light. The intensity of the ambient light should be set experimentally so as to obtain realistic visual effects. Phong lighting model applies the following formula to compute the visible light intensity:

$$L_C = S_C + k_{dC}\sum_{i=1}^{n} f_{att}(r_i)E_{iC}N \bullet I_i + k_{sC}\sum_{i=1}^{n} f_{att}(r_i)E_{iC}(I_i \bullet O_S)^g + k_{aC}A_C \quad , (7.1)$$

where:

- C - the symbol of color component (R, G or B),
- L_C - luminance at the observer direction for C component of RGB model,
- E_{iC} - *i-th* light intensity for the component C,
- r_i - distance to *i-th* light,
- N - unit surface normal vector,
- I_i - unit vector to *i-th* light,
- O_S - specularly reflected observer unit vector,
- A_C - intensity of ambient light,
- $f_{att}(r)$ - light attenuation as a function of distance.

Remaining symbols correspond to already commented surface attributes. The symbol $\bullet$ denotes the dot product of vectors. The first component of the Phong formula represents the contribution of the surface self-luminance to the observed light intensity. The second component reproduces the effect of diffuse light reflection on Lambertian surfaces. The contribution of each visible light source to the fragment illumination is computed. The diffuse illumination from all point lights is summed and the total illumination is multiplied by k_{dC} factor thus giving the reflected light intensity corresponding to light diffusion. The term $N \bullet I_i$ attenuates the illumination from the point light source with increasing angle between the normal vector and the direction to *i-th* point light. The illumination intensity is highest is the light illuminates the surface perpendicularly. It decreases with increasing angle between N and I_i according to cosine function of this angle. The illumination intensity is also attenuated by the distance r from the illuminated surface to the point light. The formula for $f_{att}(r)$ compensates various nonlinearities and simplifications in the channel of synthetic image visual perception that were not explicitly taken into account in the lighting model. The flexible formula can be applied here, where by tuning coefficients appearing in the formula the most realistic result can be obtained:

$$f_{att}(r) = \min\left(\frac{1}{c_2 r^2 + c_1 r + c_0}, 1\right) \cdot \tag{7.2}$$

The computed attenuation factor is restricted to the interval *(0,1)*. In this way the unwanted effect of lighting amplification (in the case of very close point lights) is avoided. The reader is encouraged to experiment with c_1, c_2, c_3 parameter settings and to observe how they affect rendered image.

The third component of the Phong formula reproduces the effect of glossy near-specular reflection of point lights. The modeled surface is assumed to be not perfect mirror. Therefore, the point light reflection is not rendered as infinitely small point on the reflecting surface but rather it creates a visible highlighted area. The intensity of highlight depends on the angle between mirror reflected observer direction O_S and the direction to the point light I_i. Cosine of the angle $\cos(N, I_i) = N \bullet I_i$ is raised to the power of the glossiness factor g. The higher the value of g, the faster the highlight attenuation term approaches 0 with increasing angle, and in result, the smaller is the highlight diameter. The specularly reflected light is attenuated with the distance to the light with the same formula used for diffuse reflection.

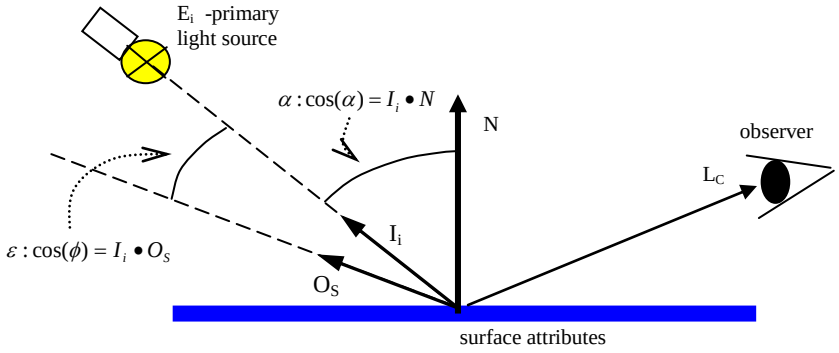


Fig. 7.3. Geometrical elements used in Phong formula

The last component in Phong formula accounts for ambient light reflection. The ambient light intensity *A* is defined uniformly for all elements of the scene. The observed light intensity fraction related to illumination by ambient light is computed by multiplying ambient light intensity *A* by the diffused reflection coefficient k_{aC} of the surface.

Simple rendering by ray casting

The concept of ray casting as a 3D rendering technique was first introduced by Appel in 1968 ([1]). It consists in finding the object visible through the pixel of the raster image by tracing a half-line that starts at the observer point and goes through the pixel center. The half-line corresponds to the path of light which is emitted by the observed object and which is terminated at the observer eye. The line segment from the observed surface fragment to the eye is called *primary ray*. The point on the scene object surface visible through the pixel can be determined by finding the intersection of the half-line with the object of the scene that is closest to the observer along the ray line. The rectangle of the rendered image is located in the scene space, so the coordinates of each pixel center can be easily determined with simple geometrical calculations.

Let use explain the rendering with ray casting using the simplest possible scene consisting merely of single ball centered at the point *(0,0,0)*. The radius of the sphere is *r*. The observer is located on OZ axis in infinity on the negative side of the axis. The projection plane is perpendicular to OZ axis and crosses it at *z=-r*. The upper left vertex of the screen square is located at *(-r,r,-r)*, the lower right vertex is located at *(r,-r,r)*. Due to location of the observer in infinity, all primary rays can be approximated by parallel lines. The scene in orthogonal projection onto ZOY plane is depicted in Fig. 7.4.

Let resolution of the image being rendered is *nxn* pixels. In order to trace primary ray through the pixel at the position *(j,i)* in the raster array the corresponding point on the projection plane has to be found. We will assume that rows of the image are indexed downwards (the convention used in Java2D). We need the following screen-to-projection plane mapping:

Vertex	Pixel coordinates	Coordinates on the projection plane
upper left	*0,0*	*-r,r,-r*
upper right	*(n-1),0*	*r,r,-r*
lower left	*0, (n-1)*	*-r,-r,-r*
lower right	*(n-1),(n-1)*	*r,-r,-r*

The following formulas transform the center of the image pixel *(j,i)* into the point *(x,y,-r)* on the projection plane

$$x = r(\frac{2j}{n-1} - 1), \quad y = r(1 - \frac{2i}{n-1}). \tag{7.3}$$

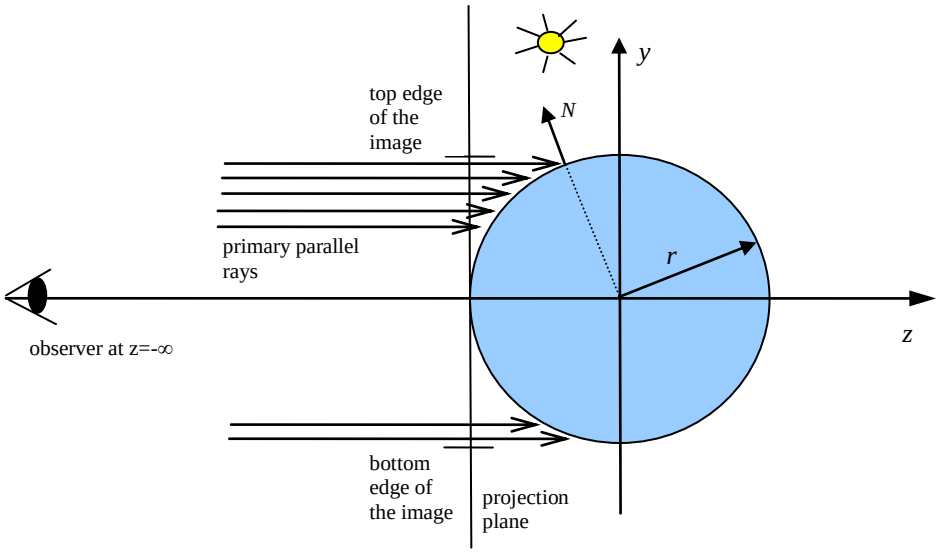


Fig. 7.4. Scene configuration for simplest ray casting

The intersection of the primary ray (0,0,1) containing the point (x,y,-r) on the projection plane with the sphere exists if $x^2 + y^2 < r^2$. Otherwise the primary ray does not intersect the sphere and background color is displayed at the pixel. If the intersection exists then the coordinates of the intersection point P_{int} are:

$$P_{\text{int}} = (x, y, \sqrt{r^2 - x^2 - y^2}). \tag{7.4}$$

Because the normal vector N_{xyz} at the point *(x,y,z)* on the sphere surface is parallel to the line connecting the point with the sphere center then $N_{xyz} = (\frac{x}{r}, \frac{y}{r}, \frac{z}{r})$.

In this way we have found all data necessary to compute the visible colors of the sphere fragments using Phong lighting model and sphere rendering with ray casting can be easily implemented.

Assignment scope

1. Implement the program that renders a sphere with ray casting technique using Phong shading model. Elaborate simple scene description text format that specifies:
   - number of point light sources,
   - position and light intensities E_R, E_G, E_B for each point light,
   - surface parameters of the sphere,
   - intensity of ambient light A_R, A_G, A_B.
   - resolution of the image being rendered,

– file name for the rendered image.

The program should read scene description form the text file specified by a command line parameter. The rendered image should be displayed on the screen and saved to the graphic file.

2. Make experiments with the implemented renderer:
   a) prepare sets of surface attributes that produce images of natural surfaces:
      – white gypsum plaster,
      – gypsum plaster painted with pastel colors,
      – mate plastic in different colors,
      – glossy plastic,
      – others;
   b) make series of experiments with various coefficient values in the light attenuation formula. Suggest parameter settings that give the most realistic results.
3. Modify selected surface attributes procedurally in the way similar to the one followed in Assignment 2. Compute pattern in spherical coordinate system.

Assignment 8
Basic elements of boundary-represented geometry rendering

Aim

The aim of this assignment is to get acquainted with methods of 3D scene projections onto the projection plane. Observer coordinate system is introduced. The students will learn how to construct the transformation that transforms the scene into observer coordinate system.

Theoretical fundamentals

Representing geometry of 3D scenes

The most essential part of scene data is the description of scene elements geometry. Many methods for geometry representation are used in 3D computer graphics. The review of geometry representation methods can be found in [8], [9] and [16]. Display techniques impose some restrictions on the methods of geometry representations. The simplest technique of 3D scene presentation is *wireframe* display. Wireframe display technique inherently requires that scene object surfaces are approximated by sets of polygons. The method of geometry representation by a polygon mesh is called *boundary representation* (BR). In wireframe mode, the boundary represented geometry is displayed in this way that edges of polygons are displayed as line segments. No visibility analysis is being conducted, so all polygon edges are always visible. It resembles the simplified geometry model, where the actual shape is replaced by the frame made of wires. Fig. 8.1. shows exemplary shapes displayed in wireframe mode.

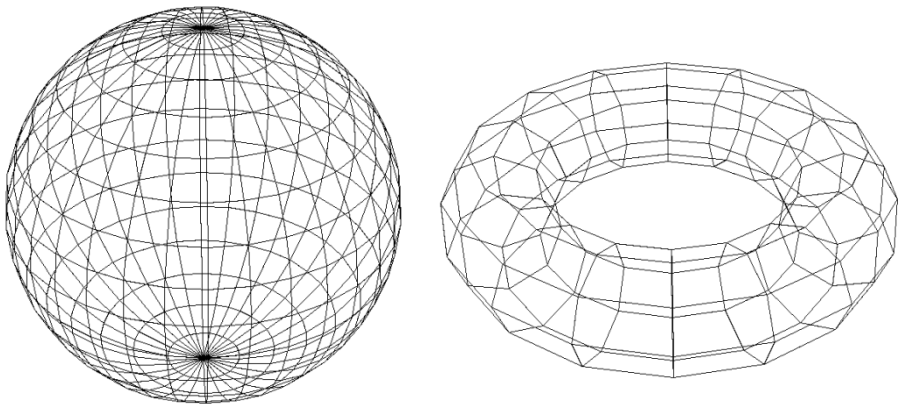


Fig 8.1. Simple scene displayed in wire frame mode

The main advantage of BR is its flexibility and ability to represent any shape. It is especially useful in representing irregular shapes like plants, parts of human or animal bodies, terrain shapes etc. The disadvantage of BR is that the geometry modeled by polygon mesh is always rough. Smooth surfaces like spheres, cylinders etc. cannot be

precisely represented with BR. The polygons in the mesh can be however subdivided so as to obtain arbitrarily precise approximation of any smooth shape.

Although in general BR can contain any polygons, data structures used to store BR are simplified (and in result can be more efficiently used in a program) if BR exclusively consists of triangles. Internally in a program memory BR of the scene can be represented by two arrays. The first array contains 3D coordinates of points being vertices of the triangle mesh. The second array contains records describing triangles. Each triangle record consists of:

– three indices of triangle vertices,
– index of the part the triangle belongs to.

The triangles are grouped into parts. A part is a set of triangles sharing common surface properties or constituting the surface of individual solid. The BR data structure is shown in Fig. 8.2.

Vertex array:

X	Y	Z
x_0	y_0	z_0
x_1	y_1	z_1
	…	
x_{n-1}	y_{n-1}	z_{n-1}

Triangle array:

v_0	v_1	v_2	p
$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	p_0
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	p_1
	…		
$i_{m-1,0}$	$i_{m-1,1}$	$i_{m-1,2}$	p_{m-1}

Fig. 8.2. Boundary representation data structures in two arrays

BR consisting exclusively of triangles can be also stored in simply organized text files. For the sake of this assignment the following text format of BR files will be used. The BR text file consists of three sections. The first section is the array of triangle vertices. The next section is the array of triangles specified by vertex triples. The last section is the array containing part indices for subsequent triangles defined in the previous section. The data are stored in the file in the following defined as follows (Java style comments are to explain the meaning of numbers only; the actual file exclusively consists of numbers):

```
n                 // number of  vertices
x0, y0, z0        // coordinates of the first vertex
x1, y1, z1        // coordinates of the second vertex
. . .
xn-1 yn-1 zn-1    // coordinates of n-th vertex

m                 // number of triangles
i0,1 i0,2 i0,3    // zero-based vertex indices of the first triangle
i1,1 i1,2 i1,3    // vertex indices of the second triangle
. . .
im-1,1 im-1,2 im-1,3 // vertex indices of m-th triangle

p0                // index of the part of the first triangle
p1                // index of the part of the second triangle
. . .
pm-1              // index of the part of m-th triangle
```

Listing 8.1. shows BR of the unit cube in this format, where all cube walls belong to the same part.

```
8
0.0   0.0   0.0
1.0   0.0   0.0
0.0   0.0   1.0
1.0   0.0   1.0
0.0   1.0   0.0
1.0   1.0   0.0
0.0   1.0   1.0
1.0   1.0   1.0
12
0 4 6
0 6 1
1 6 7
1 7 3
3 7 2
2 7 5
2 5 0
0 5 4
4 5 7
4 7 6
3 2 0
3 0 1

0 0 0 0 0 0
0 0 0 0 0 0
```

Listing 8.1. Boundary represented cube shape in simple BR format

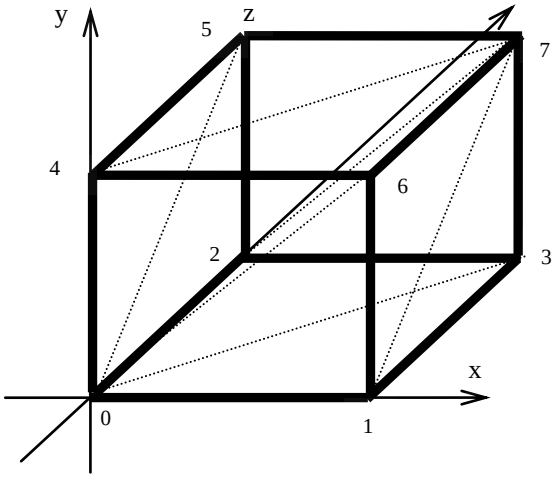


Fig. 8.3. The cube composed of triangles defined by BR in Listing 8.1

Scene geometry transformations in 3D

The scene elements are often designed independently. Coordinates of their triangle meshes are initially defined in local coordinate systems specific to particular scene elements. The complete scene then needs to be assembled from components. The assembling procedure consists in transforming scene elements from local coordinates systems to the common scene (world) coordinates. We will assume that the transformation from local to world coordinates system is a combination of translations, rotations and scaling. The concept of homogenous coordinates described for 2D in Assignment 3 is also used in 3D graphics. 3D points are now represented by vectors (x', y', z', w). Conversion from homogenous coordinates to original 3D coordinates consist in dividing *x',y',z'* coordinates by *w*, i.e.

$$(x, y, z) = (z'/w, y'/w, z'/w)$$

Transformation matrices for translation and scaling are analogous to these ones derived for 2D images:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

a) scaling b) translation

Basic rotations are now defined as rotations about axes OX, OY and OZ. In order to define directions of positive rotations (i.e. rotations by positive angle) consistently we need to determine how three axes directions are related. If we look perpendicularly at XOY plane so as the OX axis points to the right and OY axis points upwards then:

- if OZ axis is oriented towards the observer then the coordinates system is *right-handed*,
- otherwise the coordinates system is *left-handed*.

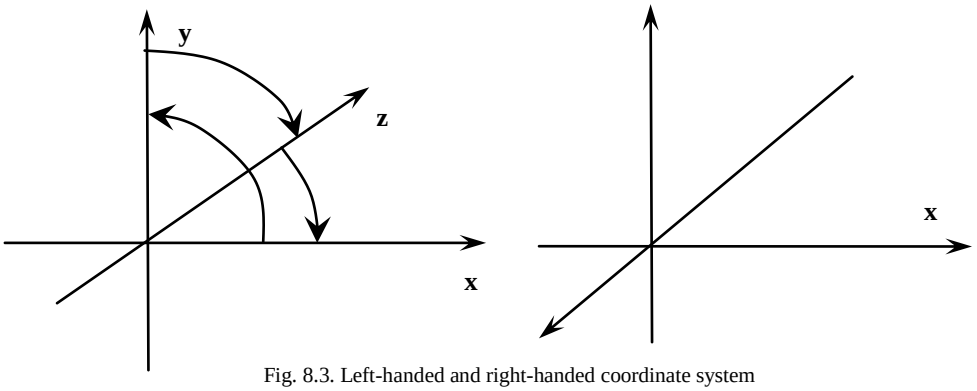


Fig. 8.3. Left-handed and right-handed coordinate system

The orientation of the positive angle rotation about all axes is defined so that when we look in the direction of the rotation axis and remaining axes point out to the right and upwards, then the rotation is counterclockwise in the left-handed coordinates system and clockwise in the right-handed coordinates system. The rotation orientation in left-handed coordinates system is presented in Fig.8.3. The rotation matrices in left-handed coordinates system are:

$$\begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 & 0 \\ \sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OZ axis rotation OY axis rotation OX axis rotation

All essential properties derived for matrix transformations in homogenous coordinates in 2D are valid also for 3D. In particular:

a) point transformation consists just in multiplication of the point vector by the matrix,
b) more complex transformations can be obtained by multiplying transformation matrices of their components.

In 3D graphics, matrix transformations in homogenous coordinates are used not only for scene modeling but also for projections and in many other operations related to shadow rendering and texture mapping ([14], [17]).

Projections

The scene image in created on the plane. In order to create the image, the scene elements are projected onto the projection plane. The projection is defined as transformations that reduces dimensionality. In CG, projections transform 3D scene space points into 2D points on the *projection plane*. Many types of projections have been proposed ([9]). In 3D scenes rendering two types of projection are used most commonly: parallel projection and one point perspective projection.

Parallel projection

Parallel projection is defined by specifying the projection plane and projection direction *u*. The projection plane is specified by its equation

$$ax + by + cz + d = 0. \tag{8.1}$$

The vector $N = (a, b, c)$ constructed of plane equation coefficients defines the plane normal vector. The plane equation can be written in simpler form:

$$N \bullet P' + d = 0. \tag{8.2}$$

The image *P'* of the point *P* in the parallel projection is a point on the projection plane where it is intersected by the parametric line $P' = P + ut$. Vector *u* defines *direction of projection*. By substituting *P'* in the plane equation 8.2 we obtain:

$$N(P + ut) + d = N \bullet P + (N \bullet u)t + d = 0 \tag{8.3}$$

and hence:

$$t^* = -\frac{N \bullet P + d}{N \bullet u}, \tag{8.4}$$

and finally the projection *P'* can be computed as:

$$P' = P + ut^* = P - u\frac{N \bullet P + d}{N \bullet u} \tag{8.5}$$

The specific case of parallel projection often used in CAD software is *orthographic projection* where the projection direction is parallel to one of axes OX, OY, OZ and the projection plane is perpendicular to the projection direction. Finding the coordinates of the projected point on the projection plane consists in this case just in rejecting one of coordinates that corresponds to the direction axis.

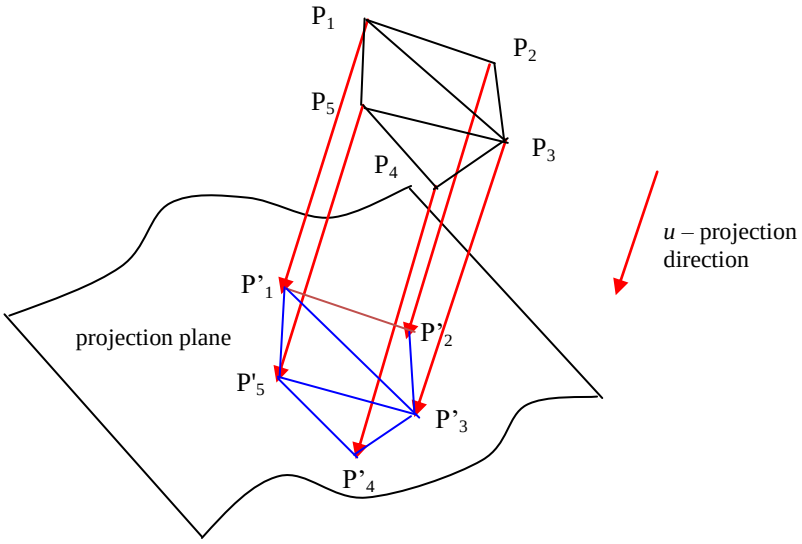


Fig 8.4. Parallel projection of three triangles $(P_1,P_2,P_3),(P_1,P_3,P_5),(P_5,P_3,P_4)$

Perspective projection

The perspective projection more closely corresponds to paths of light that is reflected on the object surface and falls onto the retina in human's eye. Therefore it is typically used in realistic image synthesis. Finding a projection *P'* of the point *P* on the projection plane consists now in finding the intersection of the half-line that starts in the observer position P_O and includes the projected point *P* with the projection plane. In order to find the image of the point being projected, formulas analogous to 8.1, ..., 8.5 can be used, but now the projection direction *u* is the vector defined by P_O and *P*, i.e. $u = (P-P_O)$ and the point *P* in 8.1, ..., 8.5 is replaced by P_O.

The problem however remains how to transform points on the projection plane into coordinates in image space measured by pixel units. In would be convenient to have projection plane parallel to one of coordinate's system axes. Hence the coordinates of projected points could be easily transformed to pixel coordinates using simple linear transformations independently in *x* and *y* directions.

Therefore we introduce the notion of *observer coordinates system*. In observer coordinates system, the observer is located in "standard" position on OZ axis on its negative side. Viewing direction is parallel to OZ axis, so the point *(0,0,0)* is in the center of the field of view. The projection plane is XOY. The observer coordinates in orthographic projection parallel to OY axis are presented in Fig 8.5.

By using similarity of triangles *(O,P,ZP)* and *(O,P'C)* the following proportions can be derived:

$$\frac{x'}{x} = \frac{d}{d+z}; \quad \frac{y'}{y} = y\frac{d}{d+z} \tag{8.6}$$

and hence we get extremely simple formulas for computing *(x'y')* coordinates of *(x,y,z)* point projection onto the projection plane:

$$x' = x\frac{d}{d+z}; \quad y' = y\frac{d}{d+z}. \tag{8.7}$$

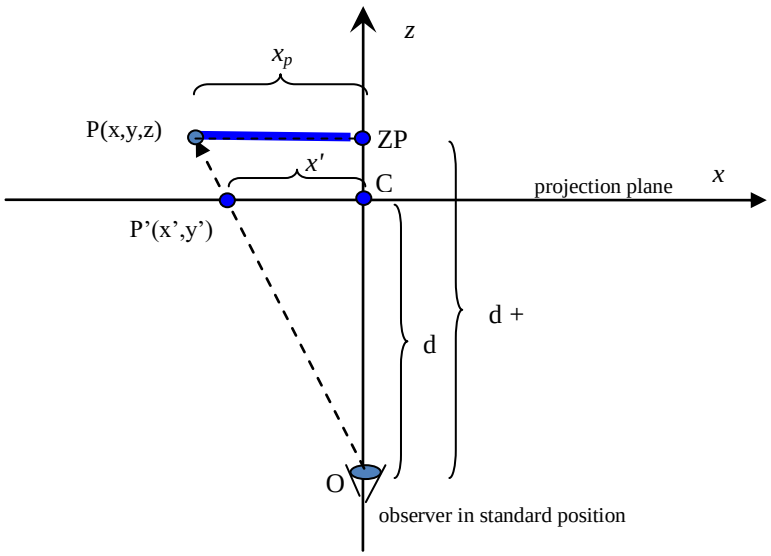


Fig. 8.5. Perspective projection in standard observer coordinates system

Transforming from scene to observer coordinates

The transformation from scene to observer coordinates can be composed of translations and rotations. Let us assume that observer is specified by:

– position of the eye P_O,
– viewing target point C determining the center of the image on the projection plane,
– direction V that defines the vertical (y) axis of the observer coordinates system.

Viewing direction is determined by the vector $u = P_O\text{-}C$. Vectors V and u are assumed to be perpendicular.

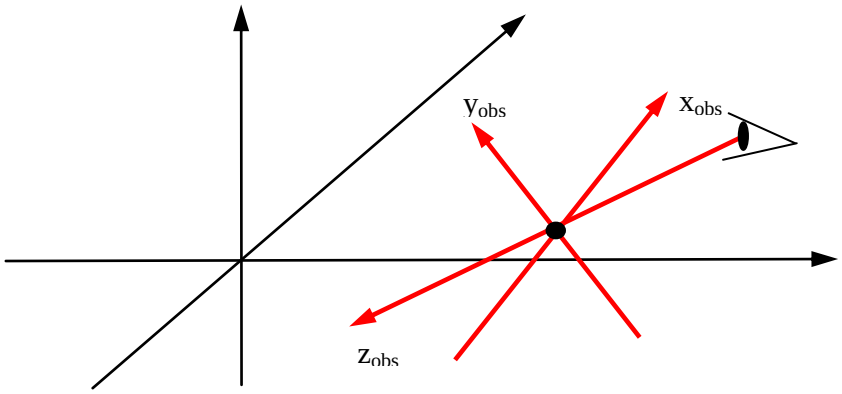


Fig. 8.6. Observer coordinates system (red axes) in scene coordinates (black axes)

The transformation from scene to observer coordinates consists of four steps (left-handed coordinates system is assumed).

1) Translation by the vector -*C*. The image center *C* is now at the point *(0,0,0)*. Use Translation matrix M_T to perform the first step transformation. P_O becomes now $P_O'=(x',y',z')$

2) Rotation about OY axis so as to bring the observer onto ZOY plane. The rotation angle φ (Fig. 8.7) can be programmatically computed using `atan2()` function: $\phi = -(\pi/2 + a\tan 2(x', z'))$, where $P_c'=(x',y',z')$ is the eye point after the first step of transformation. Use rotation matrix M_{YR} to perform the second step of transformation. P_O becomes now $P_O''=(x'',y'',z'')$

3) Rotation about OZ axis so as to bring the observer onto OZ axis. The rotation angle φ' (Fig. 8.8) can be computed as: $\phi' = \pi + a\tan 2(z'', y'')$, where $P_c''=(x'',y'',z'')$ is the eye point after the second step of transformation. Use rotation matrix M_{XR} to perform the second third of transformation. P_O becomes now $P_O'''=(x''',y''',z''')$

4) Rotation about OZ axis so as to bring the vertical direction V''' onto OY axis. In order to find the rotation angle apply the transformation from the steps 1), 2) and 3) to the vertical direction *V* defined in the observer parameters. Let $V'''=(x''',y''',z''')$ denotes the transformed direction *V*. The rotation angle φ'' can be computed as: $\phi'' = \pi/2 - a\tan 2(x''', y''')$ Use rotation matrix M_{ZR} to perform the rotation.

In order to obtain the complete transformation matrix that transforms scene coordinates into observer coordinates, the matrices: M_T, $M_{YR,}$ M_{XR} and M_{ZR} obtained in steps 1) - 4) should be multiplied in the order corresponding to described steps. The resultant product of multiplication is the matrix of the complete transformation.

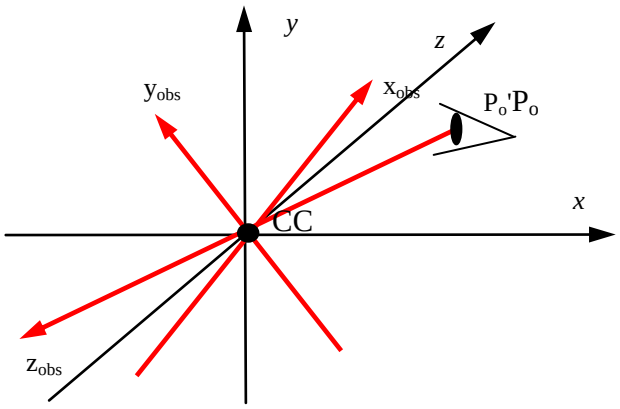


Fig. 8.6. Translated observer coordinates

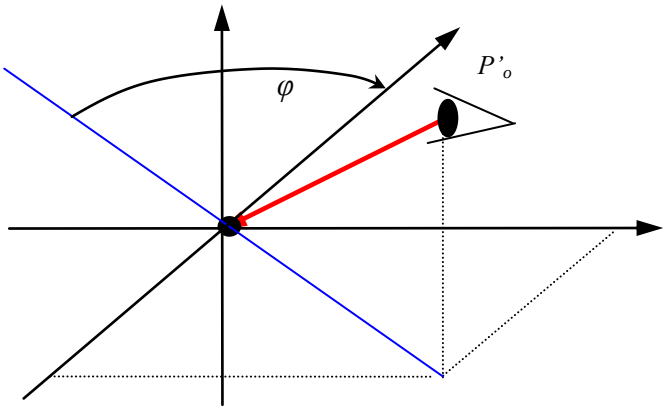


Fig. 8.7. Rotation about OY that brings observer onto ZOY plane

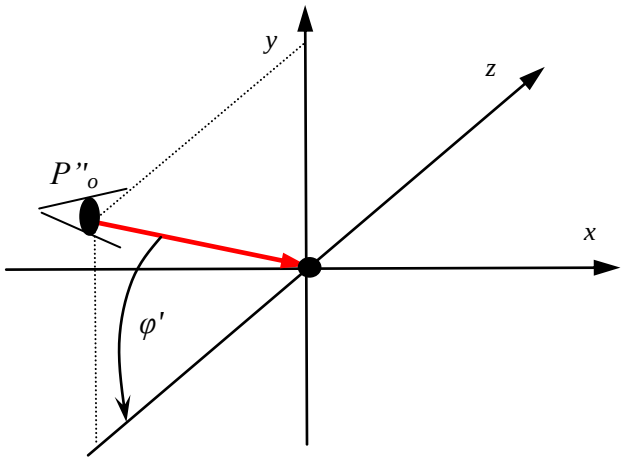


Fig. 8.7. Rotation about OZ that brings observer onto OZ axis

Assignment scope

Implement the program that makes possible to set observer parameters so as to obtain the required view of the scene. The program displays the scene in wireframe mode in four viewports containing perspective view and three orthographic views based on front-elevation, top-elevation and side-elevation projections. Scale orthographic views so as to enclose tightly the scene domain. The scene domain is the smallest axis-aligned box that contains all scene elements.

Two small icons representing the eye position P_O and viewing target point C should be displayed in orthographic viewports and connected with the line symbolizing viewing direction. The user can drag icons over viewports. The program recalculates positions of P_O and C points in scene coordinates on-the-fly and modifies scene-to-observer transformation matrix appropriately. The perspective view is redisplayed in real time when eye or target icons are dragged. Use slider UI control to set field of view.

Set vertical direction V so that it is perpendicular to $(P_O\text{-}C)$ vector and to the cross product of $(P_O\text{-}C)$ and $(0,1,0)$ vector. Select direction of the vector V so as to obtain non-negative value of the dot product $(0,1,0) \bullet V$.

The program should be able to read and display boundary represented geometry stored in the format specified in the section "Representing geometry of 3D scenes". Elaborate the format of the observer file where the complete observer specification is stored. Load the observer file while loading the scene geometry. Use the same basename of the scene and camera file. Use different name extensions, e.g. "geo" for boundary representation of the scene geometry and "cam" for observer definition file.

...

Assignment 9
3D modeling of surface patches and solids

Aim

The aim of this assignment is to learn the methods of curve representation in 2D and 3D, to examine basic properties of Bezier curves and Bezier piecewise curves and to become familiar with their application to define 3D surface patches and rotational bodies. The simple technique of a surface patch triangulation is also presented. By implementing an application that displays created shape in 3D, students will also develop their skills in applying 3D transformations and projections.

Theoretical fundamentals

Curves are widely used in 2D and 3D graphics for modeling complex smooth shapes. In 2D they are used e.g. for defining scalable fonts in text processing and desktop publishing programs, for approximating shapes of plots defined by sets of points or for presenting smooth movement paths in animation systems. In 3D graphics, planar curves are used as a start point for defining surface patches and rotational solids.

Parametric representation of curves

In the parametric curve representation, points on the curve are defined by a vector of functions of a scalar parameter *t* coming from the specified interval $< t_S, t_E >$. For the 2D curve the vector consists of two functions defining *x* and *y* coordinates; for the 3D curve three functions defining *x,y,z* coordinates are used.

$$\begin{aligned} P_{2D}(t) &= (x(t), y(t)) \\ P_{3D}(t) &= (x(t), y(t), z(t)) \end{aligned} \quad . \tag{9.1}$$

Later on, we will focus on 2D curves, but most of conclusions are valid also for 3D curves. By *P(t)* we will mean $P_{2D}(t)$ *or* $P_{3D}(t)$.

The curve segment is defined as the set of points *P(t)* for $t \in < t_S, t_E >$. Typically we expect that the curve segment defined in this way is continuous and smooth. Fig 9.1.a shows a curve segment that is both continuous and smooth. In Fig 9.1.b, the example of a curve segment that is continuous but not smooth is presented. If the vector *P (t)* is continuous we say that the curve is G^0*-continuous.* It corresponds to the "ordinary" continuity of the curve, i.e. it constitutes a single continuous line segment. It can be shown that if functions *x(t),y(t)* and *z(t)* are continuous and have continuous derivatives *dP (t)/dt* within $< t_S, t_E >$ interval then the curve is booth continuous and smooth. We say that such curve is G^1*-continuous* ([9]).

a) continuous and smooth curve b) curve that is continuous but not smooth

Fig 9.1. Continuity and smoothness of curve segments

The curve shape is usually modeled in an interactive process. We expect that curve shape can be formed in an intuitive manner, so that a designer can easily predict the consequences if his design activities. Commonly applied method of curves definition is based on curve control points. The curve is defined in such way that its shape depends on the location and number of control points. In order to create the curve of the desired form, the control points have to be located appropriately. We will therefore define points on the curve as:

$$P(t) = \sum_{i=0}^{n} B_i(t) P_i \cdot \tag{9.2}$$

P_i are the control points that define the curve. The point $P(t)$ on the curve corresponding to certain t value is a linear combination control points. $B_i(t)$ is the weight of i-th control point depending on t. Various shapes and curve properties can be obtained by applying various formulas for weights. Typically, $B_i(t)$ are polynomials. To avoid high computation costs, the weight polynomials should be of relatively low order.

Lagrange interpolating curve

Natural expectation in curve modeling is to define the curve in such way that all control points belong to the curve. The example of a curve that satisfies this requirement is *interpolating Lagrange curve*. In order to define Lagrange curve with $n+1$ control points $P_0, P_1,\ldots,P_n$ we select $n+1$ values of t parameter within the interval $< t_S, t_E >$:

$$t_0 = t_S, t_1, t_2, \ldots, t_n = t_E, \;\; t_i < t_{i+1} \;\; i = 0,\ldots,n-1. \tag{9.3}$$

The points $t_1,\ldots,t_{n-1}$ can be arbitrarily distributed within the interval $< t_S, t_E >$, but uniform distribution can be recommended, i.e. $t_{i+1} - t_i = (t_E - t_S)/n$. The weight polynomials are defined as:

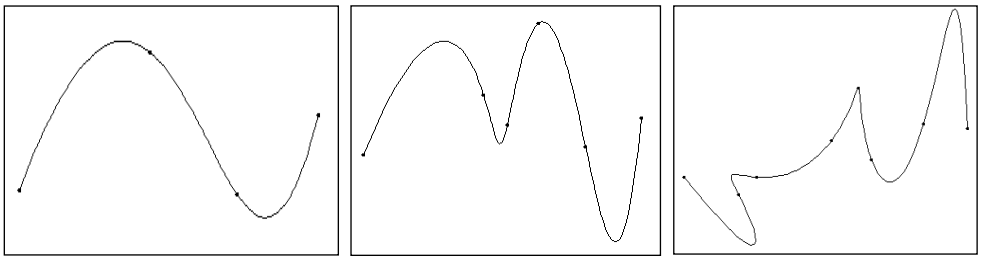
$$B_i(t) = \prod_{j=0, j\neq i}^{n} \frac{t - t_j}{t_i - t_j}, \tag{9.4}$$

and in result the curve point is defined as:

$$P(t) = \sum_{i=0}^{n} P_i \prod_{j=0, j\neq i}^{n} \frac{t - t_j}{t_i - t_j}. \tag{9.5}$$

It is easy to observe that if $t=t_i$ then all weights but the weight $B_i(t_i)$ for the control point P_i are equal to 0, whereas the weight for the point P_i is equal to 1. It means that for each $t_i, i = 0,\ldots,n$ the point defined by the curve formula 9.5 is exactly equal to i-th control point $P(t_i) = P_i$. Thus each control point belongs to the curve and the curve segment begins at the first control point P_0 and terminates at the last control point P_n.

Unfortunately, the shape of the Lagrange curve between control points cannot be easily predicted. It exhibits the tendency to strong oscillation, especially when the order of the polynomial is high. Fig. 9.2. shows examples of Lagrange curves for various polynomial orders. The weight polynomial order is determined by the number of control points. For $n+1$ control points the order of the polynomial is n. Not only it leads to unpredictable shape of the curve but also increases the computational costs of the curve drawing procedure. For these reasons practical usability of the Lagrange curve is limited to modeling of simple shapes where a few control points are sufficient.



a) 4 control points b) 6 control points c) 8 control points

Fig. 9.2. Examples of Lagrange curves obtained for 4, 6 and 8 control points

Bezier curve

In order to effectively form the curve shape with control points it is not necessary to require that the curve goes through all control points. It would be sufficient if shifts of control points cause predictable changes of the curve shape. We would also like to be able to derive some properties of the curve that help to predict the curve shape if control points location is known. Bezier curve is and example of the curve that possesses the desired properties. In Bezier curve, control point weights are defined with Bernstein polynomials:

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i \ . \tag{9.6}$$

To obtain desired properties of the curve segment, t parameter values come from the unit interval: $t \in <0,1>$. The curve is therefore defined by the formula:

$$P(t) = \sum_{i=0}^{n} P_i \binom{n}{i}(1-t)^{n-i}t^i \ . \tag{9.10}$$

For example, for $n+1=4$ control points P_0, P_1, P_2, P_3 the formula can be rewritten in the explicit form:

$$Q(t) = P_0(1-t)^3 + 3P_1(1-t)^2 t + 3P_2(1-t)t^2 + P_3 t^3 . \tag{9.11}$$

Some examples of Bezier curves are given in Fig. 9.3.

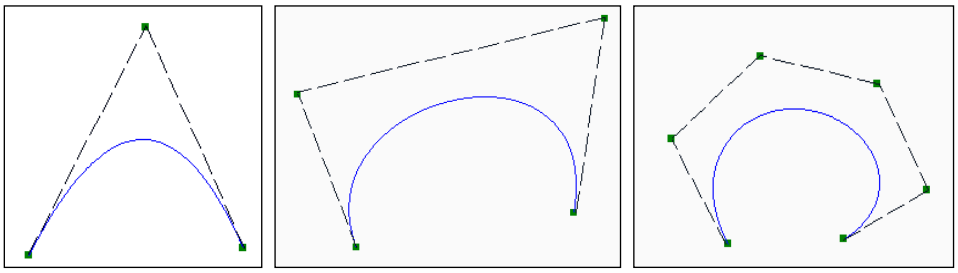


Fig. 9.3. Examples of Bezier curves defined by 3,4 and 5 control points

The following properties of Bezier curves can be derived:

1. The Bezier curve begins in its first control point P_0 and terminates in its last control point P_n, i.e. $P(0) = P_0$ and $P(1)=P_n$. In most cases, remaining control points do not belong to the curve.
2. The curve is tangent to the line segment (P_0,P_1) at the point $P(0)$ and it is tangent to the line segment (P_{n-1},P_n) at the point $P(1)$.

3. The curve is entirely included within the convex hull of all its control points.

As far as a few control points are used, the position of the point on the curve can be computed directly from the defining formula 9.10. However if the number of control points becomes greater, some numerical problems may occur. They can be caused by small and big factors appearing in the formula 9.10. The value of the Newton binominal $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ can exceed the range of integers while the value of *t* raised to the high power *i* or *(n-i)* can be very low. Such problems are unlikely to appear in modern computers for moderate counts of control points. Problems of numerical overflows/underflows can be however avoided by utilizing the observation, that Bezier formula 9.10 can be recursively decomposed into lower order formulas based on reduced sets of control points. De Casteljau algorithm based on such decomposition makes possible to find the position of a point on the Bezier curve just by performing a sequence of linear interpolations between pairs of points. Interested reader can find detailed description of De Casteljau algorithm in [16].

Piecewise Bezier curves

There are two basic disadvantages of Bezier curves. Firstly, the polynomial order increases as the number of control gets greater. In order to model complex shapes, many control points need to be used. It unavoidably leads to high polynomial degree, what in turn decreases computational efficiency of the drawing procedure. Second disadvantage is that each control point impacts the entire shape of the curve. We would rather expect that modification of single control point position influences only the fragment of the curve. In this way the shape being modeled can be tuned locally. One simple way to overcome these disadvantages is to combine the complex and long curve of smaller segments defined independently using low order polynomials. Additionally, we will be able to assure that the curve contains all explicitly specified control points.

Let's consider the sequence of control points $Q_0, Q_1, ..., Q_{n-1}, Q_n$. We will define *n* connected Bezier segments so that the pairs of adjacent controls points $(Q_0, Q_1), (Q_1, Q_2), ..., (Q_{n-2}, Q_{n-1}), (Q_{n-1}, Q_n)$ are used as the first and last control points of subsequent Bezier segments. Third-order polynomials will be used to model Bezier segments, so four control points are necessary for each segment. For *i*-th segment $(i = 1, ..., n)$ $P_0^{(i)} = Q_i$ and $P_3^{(i)} = Q_{i+1}$. Because the last control point of the previous Bezier segment is the same as the first control point of the next Bezier segment, therefore the chain of curve segments constitutes the continuous line. In order to assure G^1-continuity (smoothness) two remaining control points P_1 and P_2 will be fixed automatically. Taking into account the Bezier curve property that the curve is tangent to line segments (P_0, P_1) and (P_{n-1}, P_n), in order to preserve smoothness, the control points $P_2^{(i)}, P_3^{(i)} = P_0^{(i+1)} = Q_{i+1}, P_1^{(i+1)}$ must be collinear. It seems reasonable to put these three points on the line that contains Q_{i+1} and that is parallel to the line defined by adjacent explicitly specified control points Q_i and Q_{i+2}. The distance of automatically created control points $P_2^{(i)}, P_1^{(i+1)}$ to the point Q_{i+1} should be set experimentally, so as to obtain smooth shape of the whole curve. It can be related to the distance of adjacent explicitly specified control points, e.g. $| P_1^{(i+1)} - Q_{i+1} | = k | Q_{i+2} - Q_{i+1} |$ and

$|P_2^{(i)} - Q_{i+1}| = k\,|Q_i - Q_{i+1}|$, where k is the experimentally selected constant. Setting $k=1/3$ seems to be a good choice. A piecewise Bezier curve consisting of two segments is shown in Fig. 8.4.

The proposed above rule for finding positions of automatically added control points is not applicable to the second control point $P_1^{(1)}$ of the first segment and to the third control point of the n-th segment $P_2^{(n)}$. In these cases, the control points can be located so that the four control points of the first and the last segment constitute a symmetric trapezoid.

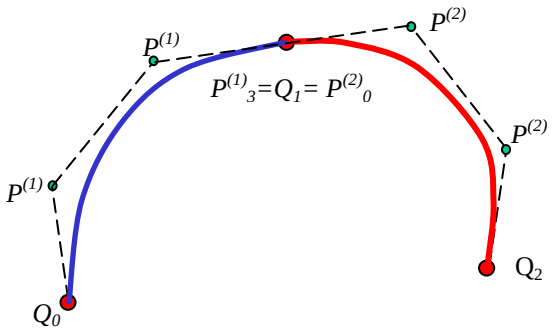


Fig. 8.4. Piecewise Bezier curve consisting of two segments

Drawing curves on the plane

Vector graphics APIs not always provide components for drawing curves directly. Often the curve can be drawn only using simpler drawing utilities as straight line drawing procedure. Any parametric curve can be approximated easily by a polyline, i.e. by the chain of short line segments which end points are located along the curve. In the simplest approach, the end points of the polyline segments can be distributed uniformly in the parameter interval $< t_S, t_E >$. If the number of approximating polyline segments is m then the step in t domain is $\Delta = \frac{t_E - t_S}{m}$ and subsequent vertices of the polyline constitute the sequence:

$$P(t_S), P(t_S + \Delta), P(t_S + 2\Delta), P(t_S + 3\Delta), ..., P(t_E - \Delta), P(t_E). \quad (9.12)$$

2D points on the curve that correspond to uniformly distributed t parameter values do not have to be uniformly distributed along the line. The curve "velocity" can be defined as:

$$\frac{d(\sqrt{x(t)^2 - y(t)^2})}{dt}. \quad (9.13)$$

This derivative determines by what distance measured along the curve the 2D point is moved while t parameter increases by dt. In order to get more uniform approximation of the curve by the polyline, the step Δ can be modified adaptively depending on the value of the local velocity 9.13, so as to get approximately balanced lengths of polyline segments.

Defining 3D shapes with 2D curves

If a curve defined on a plane located in 3D space is rotated about an axis or it is translated along a path in 3D then it "sweeps" a surface. If the end points of the curve are located on the rotation axis and the rotation angle is 2π then sweeping defines a closed rotational solid

volume. A surfaces obtained by rotating a 2D curve about an axis is called *surface of revolution*. A surface obtained by translating a curve by a path in 3D is called *extruded surface* and this modeling operation is called *extrusion*. Modeling 3D surfaces and shapes by sweeping (revolving or extrusion) makes possible to create complex 3D shapes easily and intuitively. Moreover, the swept surface can be easily converted into triangle mesh and in result it can be directly used in virtually any rendering method.

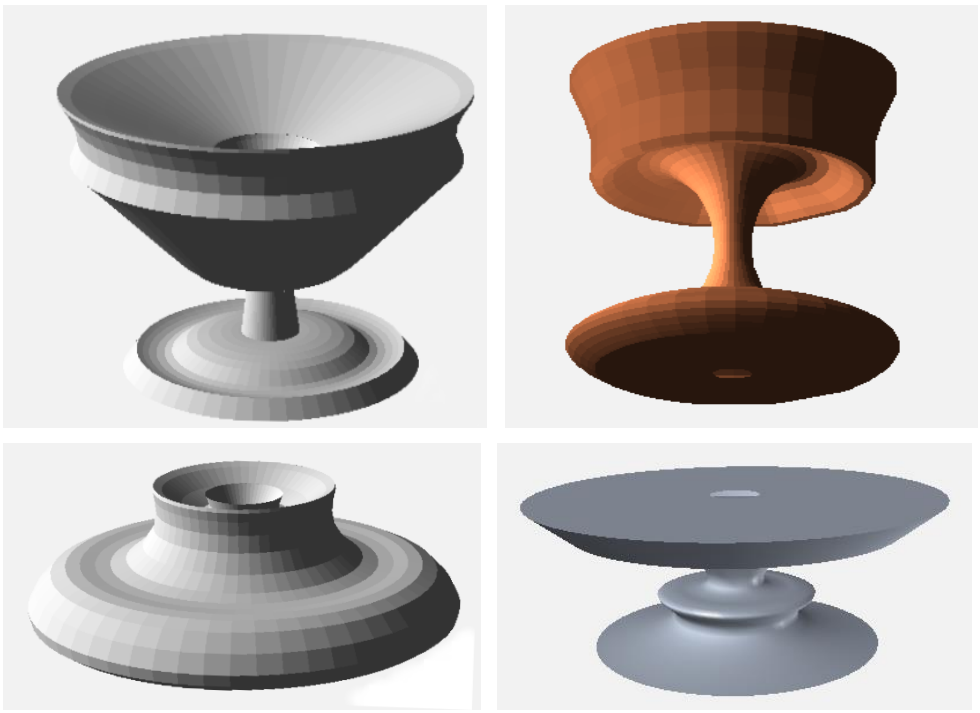


Fig. 9.5. Examples of rotational bodies created by triangulation of surfaces of revolution - first three images are rendered in flat shading mode (polygon mesh visible), fourth image is rendered in smooth shading mode (Gouraud shaded).

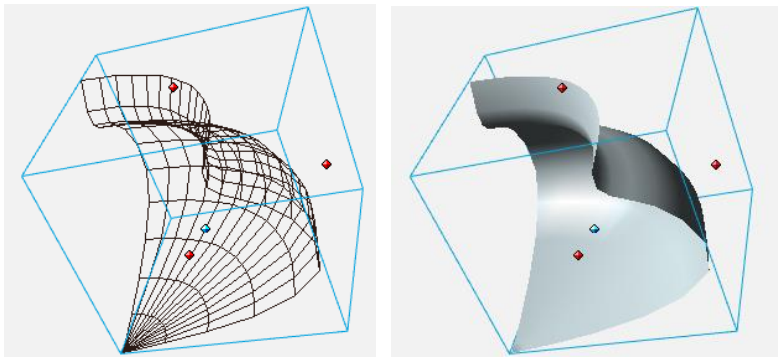


Fig. 9.6. Surface path created by curve revolving by the angle $\pi/3$: left view - polygon mesh of the surface; right view - smooth shaded image of the surface

In order to create the surface patch (or closed rotational surface) by rotational sweeping, the following data must be specified

– 2D curve on the plane located in 3D space (specified by the set of control points and scalar parameter interval $< t_S, t_E >$),

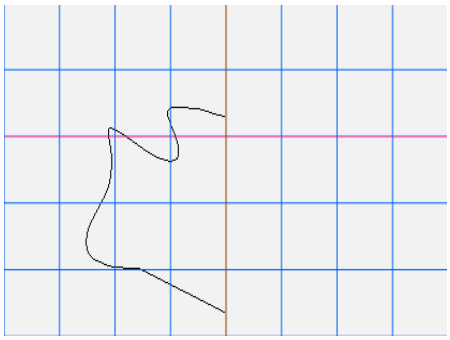
– rotation axis,

– rotation angle φ_E.

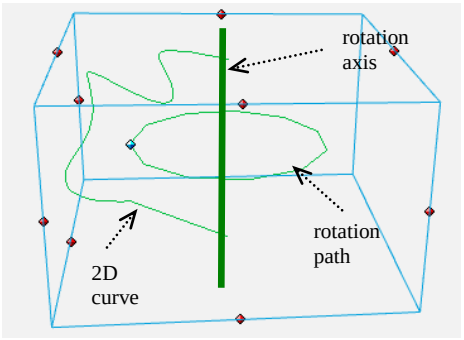
In practice, the surface always needs to be converted to the triangular mesh. In order to convert it to triangles, meshing accuracy must be specified. The density of the resultant triangle mesh is usually defined by two step lengths: Δt and $\Delta\varphi$. They determine how many quad sectors will be created on the surface being triangulated. In order to rotate points on the curve about the axis, the rotation matrix can be used. The following pseudocode outlines the algorithm of surface of revolution triangulation:

```
// set the initial rotation matrix as a identity transform matrix
M_φ-Δφ = I
for (φ = Δφ; φ <= φ_E; φ += Δφ ) {
   set the rotation matrix M_φ for the rotation by the angle φ;
   for ( t = t_S+Δt; t <= t_E; t+= Δt ) {
      create the triangle (P_1,P_2,P_3)
         where P_1 = P(t-Δt)M_φ, P_2 = P(t)M_φ-Δφ, P_3 = P(t)M_φ;
      create the triangle (P_1,P_2,P_3)
         where P_1 = P(t-Δt)M_φ-Δφ, P_2 = P(t)M_φ-Δφ, P_3 = P(t-Δt)M_φ;
   }
   // store the rotation matrix M_φ for the next iteration
   M_φ-Δφ = M_φ;
}
```

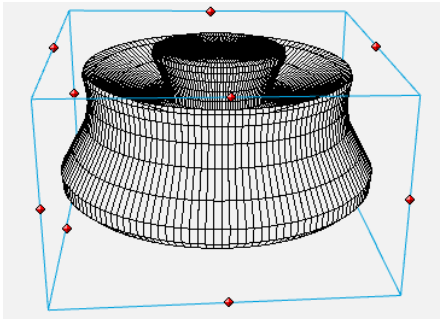
Fig. 9.7. presents the complete sequence of steps necessary to obtain triangulated model of the rotational shape. It begins with modeling of a curve on 2D plane. Then the plane is located in 3D space and the rotation angle is determined (in this case the rotation path corresponds to the rotation by 2π). In the next step the polygon mesh is created and finally the shape image is rendered.



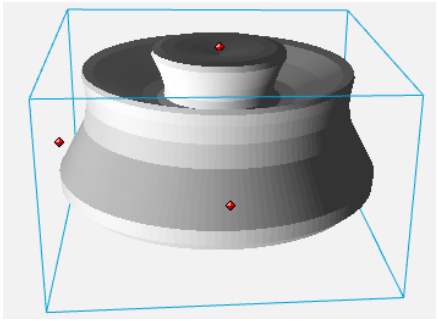
a) a curve is modeled on 2D plane



b) the planar curve is located in 3D space and the rotation axis is determined



c) polygon mesh is created



d) the object is displayed in the flat shading mode

Fig. 9.7. Steps of rotational object creation by 2D curve rotation

Assignment scope

Implement a surface of revolution modeler. The program makes possible to create 2D curve and then to build a triangle mesh by revolving the curve about OY axis. Two panels should be displayed in the program window. The left panel is the curve design area. The right panel contains 3D preview of the created surface.

Operations in the curve design area:

OX and OY axes are fixed in the middle of the curve design panel. The user should be able to:
– add a new curve control points with left mouse button click,
– reposition existing control points by mouse dragging with the left button pressed,
– delete an existing control point with right mouse button click,
– move all control points in parallel by mouse dragging with the right button pressed.
The curve defined by the set of control points should be constantly displayed and updated after each move of a control point. The curve should be updated on-the-fly while a control point is dragged.

Operations in 3D preview panel

After a curve is completed, the surface patch can be created. The curve is always rotated about OY axis. The rotation angle is specified by the slider located in the bottom part of the 3D preview panel. The slider range is $<0,2\pi>$. The triangulation accuracy should be defined by two numbers n_t and n_φ which determine counts of quad sections in the mesh. Use the spin button control for n_t and n_φ setting.

Once the 2D curve is completed, the triangle mesh of the surface can be created. Use a push-button in the 3D preview panel to create the mesh and display it. Apply parallel projection. Use the projection direction parallel to OZ axis and fix *(0,0,0)* point in the center of the preview panel. For the sake of 3D display only, implement rotation of the created mesh about OX and OZ axis. Dragging in the preview panel with the left mouse button pressed should cause the object rotation. If *x* cursor coordinate is changed while dragging then the rotation about OX is executed. If *y* cursor coordinate is changed then the rotation about OZ is executed. Create the shape transformation matrix and initialize it to represent identity transformation. Then, each time the mouse drag operation is detected multiply the transformation matrix by the rotation matrix about OY or OZ axis, depending on which cursor coordinate is changed. It is possible that both rotations should be applied as a result of single cursor movement.

Implement triangle mesh write procedure. The created triangle mesh should be stored in the text format of BR file which was described in Assignment 8. Save the original triangle mesh obtained by curve rotation. Do not store the data transformed in result of view setup operations in 3D preview panel.

Elaborate also a simple text format to store 2D curve control points. Store it in a file each time the triangle mesh is stored. Implement the complementary "Load" operation that makes possible to load the previously created curve into the curve design area.

Create several interesting shapes with your modeler. Store them as triangle meshes with various triangulation accuracies. They will be used in later assignments for experiments with image quality of smooth and flat shaded surfaces.

Assignment 10
Software implementation of visible surface determination with Z-buffer

Aim

The aim of this assignment is to deeply understand the principles of visibility analysis with Z-buffer, experiment with software implementation of this method, evaluate its performance and compare it with the hardware implantation of the same technique available in PC graphic boards. Assignment 6 is the prerequisite for this assignment because interpolated triangle shading algorithm described there is an important element of this exercise.

Theoretical fundamentals

Correct visible surface determination (called also *visibility analysis*, *visibility determination* or *hidden surface elimination*) is one of the most crucial elements of the rendering pipeline which contributes significantly to the realism of the synthesized image. Many visibility visible surface determination (VSD for short) methods were proposed during forty years of CG development ([1],[9],[16]). If the geometry of scene object is complex, VSD is a time consuming process. Therefore VSD methods used at successive stages of CG evolution were always adapted to hardware computational capabilities. Since early nineties of XXth century, rendering algorithms are supported by specialized graphic processors. Therefore nowadays such VSD algorithms are most commonly used which can be efficiently implemented in hardware. Z-buffer algorithm is one of the most popularly used methods due to its simplicity, robustness and ability to parallel implementation. Z-buffer algorithm (known also as *depth buffer algorithm*) is credited to E.Catmull who presented Z-buffer idea in 1975. In general, the algorithm proposed by Catmull is applicable to many geometry representations, but most efficient implementation can be obtained for polygons display. Here we will describe the version of Z-buffer algorithm tailored to the triangle mesh display.

Visible surface determination as the rendering pipeline element

VSD is the final element of the typical *rendering pipeline* ([11],[17]). The rendering pipeline is the sequence of activities that need to be executed in order to render the image. The rendering pipeline processes elements of scene geometry one after another by passing then through subsequent pipeline stages. Each stage can be implemented as an autonomic processor. Therefore there can be several geometry elements in the pipeline at various stages of processing. Additionally, the operations executed at the single stage often can be also executed simultaneously, provided that appropriate parallel hardware architecture is available. The architectures of the rendering pipeline presented in various articles differ slightly, depending on the software package or hardware that implements it. For the sake of our considerations we will assume the rendering pipeline architecture as presented in Fig. 10.1.

The first stage of the pipeline consists in modeling transformation of scene elements. The aim of this stage is to transform models of scene elements defined in their local coordinates

system to the common scene (world) coordinates. The next stage consists in calculating vertex properties that will be used in further stages. If interpolated color shading (e.g. Gouraud shading described in Assignment 6) is applied, then final vertex colors have to be calculated taking lighting conditions in the scene into account. Here also other vertex attributes as texture coordinates or averaged vertex normals can be calculated. Next, geometrical elements of the scene are transformed into observer coordinates system for more efficient projection onto the projection plane. Both modeling and world-to-observer transformations can be represented by transformation matrices in homogenous coordinates. Therefore they can be combined into single transformation matrix that transforms geometry elements directly into the observer coordinates. If this approach is followed then modeling and view transformations can be executed within the single stage and lighting is moved to the successive position in the pipeline. Because now lighting is performed in observer coordinates, all light source geometry-related parameters (positions, direction) must be transformed to the observer coordinates as well.

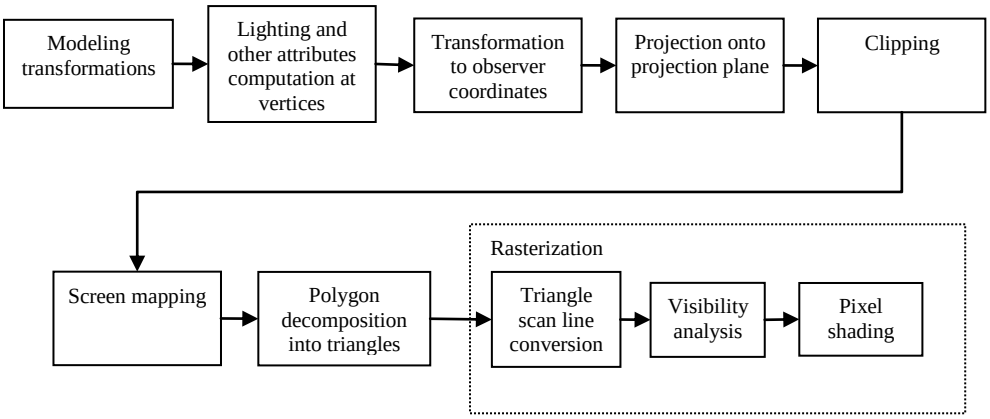


Fig.10.1. Rendering pipeline architecture

The next stage consists in projecting polygon vertices onto the projection plane using appropriate projection method. Usually one point perspective projection is applied but other projection methods can be also used. The projection maps 3D points onto infinite projection plane. Only the fragment of the projection plane will be however mapped onto the raster image being rendered. Therefore a rectangular "window" of the projection plane is defined that restricts the observer view. Some of polygons projected onto the projection plane can be entirely or partly out of the image window. In the next stage clipping is applied. Clipping consists of finding fragments of projected polygons that are inside the image window. If the polygon projection is entirely out of the window then it is not passed to further processing stages at all. It only a fragment of the projected polygon is visible through the window then the polygon is clipped, as shown in Fig. 10.2. New polygon is created which is a common part of the projected polygon and the window rectangle. Attributes at new vertices of the created polygon are computed from attributes at vertices of the original polygon.

Clipping can be also implemented in 3D by intersecting 3D polygons in observer space with a *viewing frustum*, as shown in Fig. 10.3. Viewing frustum is the volume fragment of the observer space containing the subset of scene elements than can be visible in the image window. For technical reasons the infinite pyramid defined by the observer position and vertices of the view window on the projection plane is clipped by two *clipping planes* parallel to the projection plane. This clipping can be utilized to restrict the view volume but

mainly it is introduced due to limited accuracy of data structures used in VSD operations. 3D clipping is carried out before the projection stage.

The clipped polygon is still defined by its vertices location in observer coordinates (with *z=0*, if the projection plane is XOY plane). In order to draw the polygon in the raster image, coordinates of polygon vertices must be converted to pixels. It is achieved at the screen mapping stage. Simple linear transformations that convert *x* and *y* coordinates independently of each other can be applied here. Ordering of raster image row indices used in applied API must be taken into account. Sometimes *y* coordinates of a displayed image increase upwards, while in other APIs they increase downwards. In the case where the upper line of the image is indexed with 0 the screen mapping transformation should map:

– the upper left vertex of the image window on the projected plane onto the image pixel (0,0),
– the lower right vertex of the image window onto the pixel $(x_{res}-1, y_{res}-1)$.

As the result of clipping, the triangle can be replaced by polygons of the more complex shape (as is the case shown in Fig. 10.2) . Because the final rasterization stage works most efficiently if simplest polygons possible (triangles) are applied to it, the clipped polygon is decomposed into triangles in the next stage.

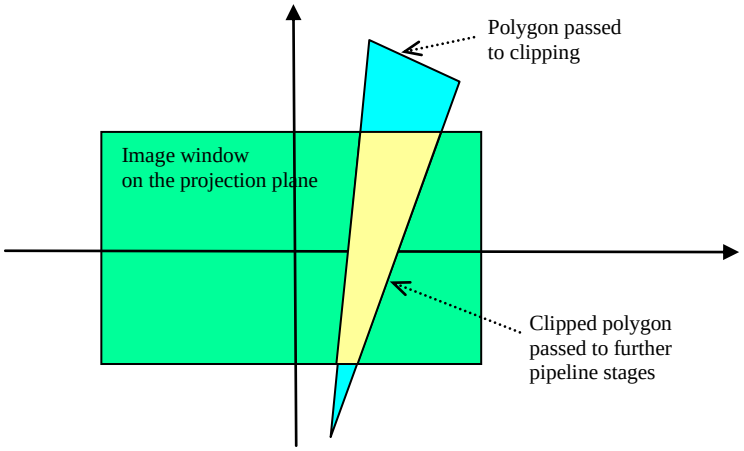


Fig. 10.2. Polygon clipping by a image window on the projection plane

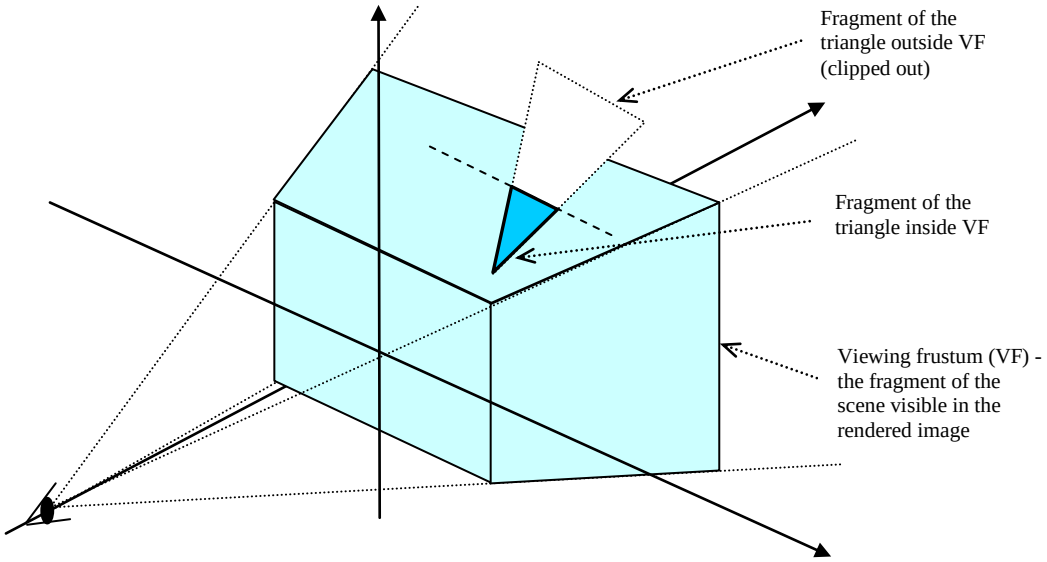


Fig. 10.3. Polygon clipping in 3D observer coordinates

Polygons transformed to raster image coordinates are ready to be *rasterized*. By *rasterization* we mean the procedure of finding the area of the raster image covered by the polygon projection, determining through which pixels the polygon is actually visible and filling it with appropriate colors. This is the stage where visibility analysis is being conducted. Logically, this stage can be subdivided into three substages. It starts with triangle *scan line conversion*. It consists in finding scan line fragments (fragments of raster image rows) covered by the triangle. The interpolation procedure described in Assignment 6 can be applied here. Then individual scan lines can be processed by independent threads running on specialized rasterizing processors. The visible fragment determination is conducted individually for each pixel in the scan line. Z-buffer algorithm is used here. Details will be described in the next section. Finally, if the triangle fragment covered by the currently processed scan line pixel is closest to the observer amongst all other fragments of triangles processed so far then *pixel shading* is applied. Pixel shading consists in determining the color of the fragment. It can be done using interpolated shading based on colors computed for triangle vertices or the complete lighting model can be used. The latter approach is followed in graphic processing units (GPUs) that support execution of short programs (*pixel shaders*) for each shaded fragment. Finally, the calculated color is stored in the raster image array. Pixel colors stored in this array can be overwritten by subsequent triangles. Therefore the image array used in the pipeline is called *color buffer* or *frame buffer*.

Z-buffer algorithm

Visible surface determination is carried out at the rasterization stage. In consists in finding out which triangle fragment (if any) is visible through the center of a pixel image. The general principle of the rendering pipeline is that triangles are passed through the pipeline one by one and that each triangle is passed only once (unless the rendering pipeline is used in a tricky way to obtain some special effects). The state of the rendering module is determined by all triangles passed so far by at the current stage we know nothing about the remaining triangles. It means that when a triangle fragment is being processed we cannot

decide if it will be finally visible until we know all subsequent triangles waiting for display. Therefore when processing a triangle fragment we can only determine if it would be visible provided there were no more triangles to display. The fragment is visible if the distance to it from the observer is shorter than the distance to fragments of all other triangles displayed so far. In the situation presented in Fig. 10.4 distances to fragments of T_1, T_2 and T_3 triangles at the pixel pointed out are d_1, d_2, d_3 correspondingly. Because $d_3 < d_2 < d_1$ then T_3 triangle is visible. Finding exact distance from the triangle fragment to the observer along the line crossing the pixel center (pixel observation line) is computationally complex. If the observer is located in its standard position on z axis ($z_O = -d_O$) and the projection plane is XOY then actual distances to fragments of triangles visible through the fixed pixel are in the same "<" relation than z-coordinates, i.e. $d_i < d_j \Leftrightarrow z_i < z_j$. Therefore instead of comparing distances it is sufficient to compare z-coordinates of triangle fragments.

The idea of depth buffer algorithm is to associate the additional array called *Z-buffer* (or *depth buffer*) having the same sizes as the frame buffer. Each pixel in the frame buffer has its counterpart in the depth buffer. Depth buffer contains z-coordinates of the triangle fragments seen through the corresponding pixels. It is assumed to be a fragment belonging to this triangle processed so far, which is closest to the observer. The triangle scan line conversion stage determines all pixels covered by the triangle. They are traversed in certain order and each pixel is being processed. For each pixel, z-coordinate of the fragment visible through the pixel is determined. Found z value of the fragment is compared with z value stored in the depth buffer at the pixel position. If z-coordinate of the triangle is greater than the stored value then other triangle already processed is closer to the observer along the pixel observation line. Hence, the current fragment is behind the other triangle and is not visible. Neither the frame buffer not the depth buffer is modified. Otherwise, it means that the fragment is closer to the observer than any other fragment of triangles processed at this pixel. In this case, the value in the depth buffer is replaced by z-coordinate of the current fragment and the color in the frame buffer is replaced by the current fragment color. After the processing of all triangles is completed the value at each depth buffer position corresponds to the z coordinate of visible fragment and the color buffer contains its color. The ultimate scene image is created in the color buffer.

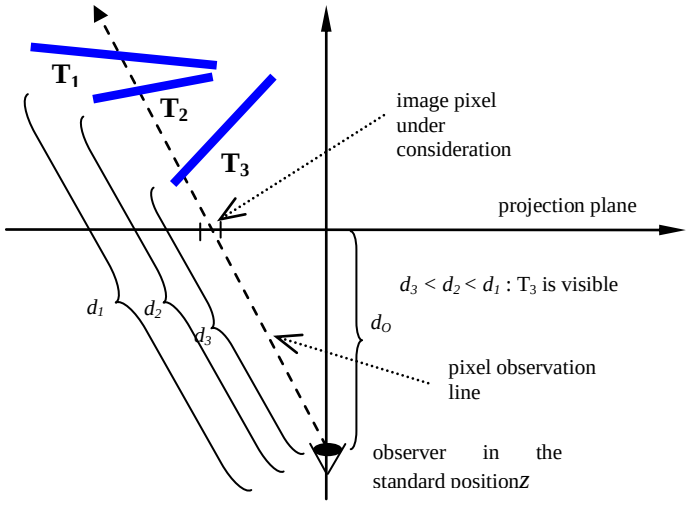


Fig. 10.4. Visibility analysis with depth test

In order to obtain correct results, both buffers should be appropriately initialized. Then to allow depth and color buffers unconditional modifications by the first triangle being displayed, the depth buffer should be initially filled with z values that are greater than z values of any scene element. Therefore the maximal value of z-coordinates of all triangle vertices in observer coordinates should be found and the z value that exceeds it should be used to initialize the depth buffer. Color buffer should be initialized with the background color or filled with the background image or pattern.

The fragment color stored in the frame buffer can be calculated in various ways, depending on applied shading method. Gouraud shading is typically implemented in hardware supported graphics. The pixels are shaded using colors interpolated from colors explicitly specified at vertices. More realistic rendering techniques perform complete lighting model computation in each shaded fragment. The color needs to be computed for each fragment for which *z*-test passes. It is often being overwritten by other triangles. So if lighting calculations are complicated and computationally costly, then instead of computing colors for each processed fragment, sometimes the better idea is to store the visible triangle index and to postpone lighting calculations until the visibility analysis is completed.

The whole visible surface determination algorithm combined with pixel color calculation can be carried out according to the following pseudocode:

```
find zmax - the maximal z coordinate of vertices in observer coordinates;
fill depth buffer with zmax;
fill frame buffer with background color or pattern;
for each triangle t in the scene model
   for each pixel (i,j) of the triangle projection on the image plane
   {
      calculate z coordinate of the triangle fragment visible
         through pixel (e.g. by using interpolation);
      if ( z < depth_buffer[i,j] )
      {
         depth_buffer[i,j] = z;
         calculate color c of the fragment at pixel (i,j)
           by using Gouraud shading – interpolate color directly or
           by applying the lighting model the fragment;
         frame_buffer[i,j] = c;
      }
   }
}
```

The last thing that needs to be explained is how to find *z*-coordinate of the triangle fragment covered by the current pixel. The easiest and probably quickest way is to apply the same interpolation scheme as we applied in color interpolation in Assignment 6. The depth of a fragment can be treated as any other vertex or fragment attribute and can be interpolated in exactly the same way as colors were interpolated for internal pixels of triangles. The problem is however that interpolation is linear in the image space and the perspective projection is not linear. It means that z-coordinates interpolated linearly in the image space do not correspond exactly to actual coordinates of fragments visible through the triangle projection interior pixel. This problem does not appear if parallel projection is used and can be neglected in the case of narrow angle perspective projection. Otherwise, to avoid significant inaccuracies, perspective correction must be applied to interpolated *z*-coordinates. Several methods have been proposed to obtain perspective-correct interpolation. One approach presented in [12] is based on the observation that 1/z (on the contrary to z), is linearly interpolated in the image space. Therefore instead of linearly interpolating *z*-coordinate one can linearly interpolate 1/z.

In some implementations, the rendering pipeline transforms coordinates to the observer coordinate system so that the viewing frustum is mapped onto the *canonical view volume* ([11],[16]). Canonical view volume is an axis aligned bounding rectangle in the observer coordinates system, containing the world fragment that is to be rendered. With this transformation, the perspective projection is replaced with parallel orthographic projection. The transformation introduces nonlinear mapping of *z*-coordinates that compensate the nonlinearity of perspective projection. This approach is followed in hardware-implemented rendering pipelines. More general approach to perspective-correct interpolation of any other vertex attribute (e.g. texture coordinates) in presented in [6]. The concept is similar to the one presented in [12], but when connected with transformations to canonical view volume it makes possible to interpolate also other attributes defined at vertices.

Another method to avoid *z*-coordinate interpolation problems is not to interpolate it but rather calculate exact intersection of the pixel observation line with the triangle plane. Formulas (8.1)-(8.5) can be easily adapted to current purposes. The method requires however additional "per fragment" computations that slow down the rendering process.

Rendering transparent objects with Z-buffer

The rendering method presented in the previous section does not impose any restriction on the order of triangles passed through the rendering pipeline. The result of visible surfaces determination is correct independently on the triangle order. Things are, however, not that simple if partly transparent objects appear on the scene. Transparent object attenuates the light that is transmitted through the surface and it usually adds additional intensity to the light emitted in the direction of the observer. It is the result of light reflection that may appear on the transparent surface. If the transparent surface is processed in unfortunate order, it can block rendering of other surfaces visible through it. Consider again the scene presented in Fig. 10.4. Let us now assume that the triangle T_1 is opaque and triangles T_2 and T_3 are transparent. Let triangle display order is: T_3, T_2, T_1. If T_3 is displayed as the first one, the distance to it (d_3) is written to the depth buffer. When the triangle T_1 is then being displayed, the value d_3 stored in the depth buffer prevents T_1 triangle fragment from being displayed, despite the previously displayed fragment of T_3 was transparent. Depth test principle must be therefore modified to avoid such situations. Additionally, the way in which the frame buffer is modified if depth test passes must be changed for transparent surfaces. In order to model transmitted light attenuation and to allow rendering of lighting effects on the transparent surface ,the color blending formula can be used:

$$c_{out} = k_t c_{in} + c_t \,, \tag{10.1}$$

where k_t is the surface transmittance coefficient, c_{in} is the color stored in the color buffer and c_t is the result of lighting model application to the fragment of the transparent surface. k_t determines how much light falling on the surface is transmitted to its opposite side. The color stored in the frame buffer before a transparent fragment is processed corresponds to light intensity from the surface located behind the transparent one. Application of this formula causes that the color stored in the frame buffer is blended with the color of the transparent surface and the mixture is stored again in the color buffer. To correctly handle the case, where many transparent surfaces are visible one through another, the formula (10.1) must be applied in the order corresponding to decreasing distance of surfaces. Therefore the transparent triangles should be displayed in this order, starting from the most distant elements.

The rendering of scenes containing transparent objects will be organized as a two stage process. At the first stage, all opaque triangles are displayed using the method described in the previous section. Triangle order is unimportant at this stage. Transparent triangles are displayed at the second stage in decreasing order to the observer. This, however, does not guarantee correct fragment processing order at each pixel, where the least distant fragments are expected to be processed at the end. Otherwise processing of some more distant fragments can be still blocked. In order to prevent transparent triangles to block the light transportation from other fragments located behind them, the depth buffer modification is disabled at the second stage. Depth buffer test is, however, still carried out to prevent transparent surfaces located behind opaque fragments from being displayed.

The complete display algorithm consists of the following steps:
– divide the set of triangles into opaque and transparent subsets,
– display opaque triangles by passing them through the pipeline in arbitrary order,
– disable z-buffer modification but preserve z-buffer test,
– sort transparent triangles by its distance from the observer,
– display transparent triangles in decreasing distance order using color buffer modification formula (10.1).

It should be pointed out that this rendering method is not quite physically accurate and the method of transparency handling is just a rough approximation of the actual physical phenomenon. For physically accurate rendering of transparent surfaces other methods should be used. Backward ray tracing ([8],[14]) is one of the most accurate rendering methods for transparent surfaces.

Assignment scope

1. Implement the software component (class) that implements visible surface determination with depth buffer. The following methods should be implemented:
   - buffers initialization, the following parameters should be passed to this utility
     - buffers resolution,
     - z-coordinate interval (z-coordinates of the front and back clipping planes),
     - distance of the observer from the origin (0,0,0),
     - horizontal field of view (the angle between the viewing direction and the leftmost pixel in the central horizontal scan line of the image),
   - color buffer initialization by filling it with a constant color,
   - color buffer initialization with the raster image passed as a parameter of the initialization method,
   - Gouraud display of a triangle, vertex positions in the observer coordinates system and vertex colors are passed as the method parameters,
   - depth test enabling/disabling,
   - frame buffer acquisition as a raster image.

   Use the implemented module to extend the program from Assignment 6, which displays randomly generated triangles. Now draw randomly 3D coordinates of triangle vertices and display random triangles with visible surface determination. Evaluate the display rate in triangles/sec and in shaded pixels/sec. Compare your program performance with the performance of modern hardware boards. Use benchmark programs or find these data in product specifications.

2. Extend the module implemented in 1) so as to obtain 3D geometry renderer. Add a method that specifies the observer parameters in scene space. The component should create scene-to-observer transformation matrix. Modify the utility that displays Gouraud shaded triangle so that now passed vertex coordinates are defined in scene space (not as previously - in the observer space).

3. Use the module implemented in 2) to improve the shape modeler elaborated in Assignment 9. Replace wireframe display of the triangle mesh with Gouraud shading with correct visibility analysis. Experiment with various densities of triangle meshes displayed in flat shading mode (i.e. without averaging normals at vertices). Observe if increasing of the mesh density severely impacts the display rate. Use Phong lighting model. Set surface properties to obtain glossy, plastic-like material.

4. Extend the module implemented in 2) to full-fledged scene renderer. Add utilities for:
   - defining the set of primary light sources,
   - defining the material properties that will be used for subsequent triangles passed to the rendering procedure.

   Now the utility that displays the triangle is called only with triangle vertex positions as parameters. Vertex colors are calculated internally.

   Implement correct handling of transparency according to recommendations given in the previous section.

5. Use the module implemented in 4) to extend the program implemented in Assignment 8. Replace wireframe display of the perspective view with Gouraud shading display with correct visible surface determination.

...

Assignment 11
3D rendering with OpenGL

Aim

The aim of this assignment is to learn basic principles of image rendering with OpenGL and to exercise it in a practice. With created application, students will be able to render images utilizing hardware support delivered by the installed graphic boards and to compare the performance of hardware-supported rendering with the performance of similar rendering procedures implemented in software in the scope of earlier assignments. Assignment 10 is a prerequisite for this exercise because it introduces the concept of the rendering pipeline utilized in OpenGL. Understanding of the rendering pipeline concept is essential for OpenGL understanding.

Theoretical fundamentals

Modern software engineering practices recommend possibly widest usage of "off-the-shelf" components that can be easily configured and incorporated into the program being developed. In this way, typical functionality can be re-used with much lesser probability of introducing serious errors. In CG programs which utilize hardware support, application of standard components isolates programmers from the hardware layer and makes programs independent on hardware interfaces.

OpenGL is one of the earliest libraries providing hardware graphics functionalities to programmers in consistent and easy to use way. The archetype of OpenGL was the library called IRIS GL (Graphics Library) primarily elaborated by Silicon Graphics company for use in their personal workstation Personal IRIS in late eighties. In 1992, Silicon Graphics released the modified specification of their GL library called OpenGL. It was then re-implemented by many hardware and software manufacturers and became the industry standard. Despite its twenty years of history, OpenGL is still very popular and probably the most commonly used graphics library that supports both 2D and 3D graphics.

OpenGL is a set of low level utilities that provide graphic hardware services to end user CG application programmer. Because of its low-level nature, OpenGL is more difficult to master and efficiently use than other CG libraries. It however provides the programmer with enormous flexibility that makes possible to implement hundreds of visual effects with moderated programming effort.

Basic principles of OpenGL usage

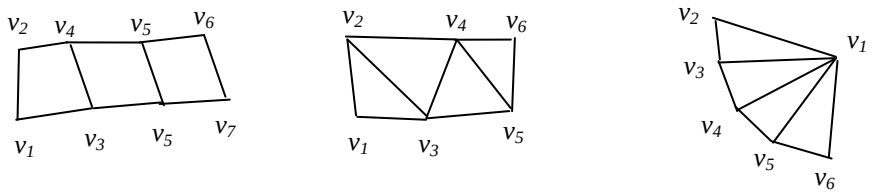
OpenGL defines a kind of display state machine. The central functionality of OpenGL consists in drawing a geometric primitive in the frame buffer. OpenGL implements the rendering pipeline similar to this one described in Assignment 10. Graphic primitives passed by the application to OpenGL undergo stages of the rendering pipeline and finally are drawn into the frame buffer. The way in which the primitive is drawn depends on the current state of the display machine. The state vector consists of great number of state variables determining how a primitive is being displayed. The most important display machine state variables correspond to:

– transformations used to transform vertices of the primitive to the image space,

– shading mode that defines how colors of primitive fragments should be computed,
– lighting conditions - OpenGL can apply its own internal lighting model to calculate colors of displayed fragments,
– current surface attributes - used to compute displayed fragment colors if internal lighting mode is enabled,
– variables determining how visible surface determination with z-buffer should operate,
– variables determining how the color calculated for a fragment being displayed is blended with the color in the frame buffer,
– texturing control data - texturing is the technique of applying raster image data to modify the color calculated for a fragment being displayed.

In order to display the geometric primitive, the application has to specify the set of vertices that define the primitive. The set of vertices can be drawn as:
– set of individual points,
– set of not connected line segments,
– polyline,
– closed polyline,
– single polygon,
– set of quads,
– set of triangles,
– set of polygons that share vertices in the specific way (quad strip, triangle strip, triangle fan).

Fig. 11.1. presents the geometrical structures composed of polygons, defined as a single primitive. Defining the complete structure as a single primitive makes possible to avoid superfluous geometric transformation repeated for the same vertices, appearing in simple polygons defined individually.



a) quad strip b) triangle strip c) triangle fan

Fig. 11.1. Compound OpenGL geometric primitives

Before the program begins displaying scene geometry with OpenGL by pushing geometry primitives to the rendering pipeline, the state variables should be set up to obtain required visual effects. At least, transformations and shading model have to be defined. If internal lighting mode is to be used, required number of lights should be defined as well. After initial setup is completed, the application pushes subsequent geometry primitives to the OpenGL pipeline. The framework of typical operation sequence is presented in the following pseudocode:

```
initialize a window for OpenGL display and bind it with OpenGL rendering
   pipeline output;
set basic rendering parameters (transformation matrices for modeling and
   projection, shading mode, lights);
for all scene objects
{
   set modeling matrix for the current part;
```

```
   define material attributes of the object (for internal lighting mode);
   for all geometric primitives constituting the object
     display the current primitive;
}
flush the rendering pipeline to make sure that all primitives have been
  displayed;
```

If the image is to be displayed in a screen window, the link of OpenGL output with the GUI element controlled by GUI manager has to be established. OpenGL defines a *rendering context* similar to 2D drawing context explained in Assignment 2. In order to direct the output of OpenGL to the window on the screen, these contexts must be linked. The properties of the window pane used as OpenGL output (in particular the way in which pixels are represented) must be compatible with OpenGL rendering context. This operation of context linking is not standardized and can be different in various GUI environments. If the application can be arranged so that:

– OpenGL output does not have to be tightly bound to other GUI elements of the application,
– OpenGL image display and interaction can be carried out in a separate window

then the process of OpenGL output window initialization can be simplified by using `aux` library that accompanies OpenGL in the many of development environments. This library can be also used to organize simple user interaction with the displayed scene by simple mouse and keyboard event handling. `Aux` can be also used to display complete polygon meshes of typical shapes (spheres, cubes, toruses) for testing purposes. We will use aux library in an example presented in the later part of this chapter.

The application informs OpenGL that the new primitive definition begins by the call of `glBegin()` function. The parameter of this function is a constant determining how the set of vertices should be interpreted and drawn. Possible values are: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_POLYGON`, `GL_TRIANGLES`, `GL_QUAD`, `GL_TRIANGLE_STRIP`, `GL_QUAD_STRIP`, `GL_TRIANGLE_FAN`. Then subsequent vertices constituting the primitive should be passed to OpenGL using one of `glVertex*()` functions[1]. After all primitive vertices are passed, the application calls `glEnd()` function indicating that the complete set of primitive vertices is ready to be displayed. Calling `glEnd()` does not necessarily result in the immediate display of the primitive. It only means that the complete primitive can be passed to subsequent stages of OpenGL rendering pipeline. For example, to define a polygon consisting of five vertices the following sequence of calls should be issued:

```
glBegin(GL_POLYGON);
   glVertex3f(0.0, 0.0, -10.0);
   glVertex3f(3.0, 0.0, -10.0);
   glVertex3f(3.0, 3.0, -10.0);
   glVertex3f(1.5, 4.0, -10.0);
   glVertex3f(3.0, 0.0, -10.0);
glEnd();
```

[1] OpenGL provides many variants of functions accepting different types of arguments. The asterisk character stands for the function name suffix which determines the type of arguments. b/s/i/f/d/ub/us/ui suffices can be applied. The suffix may also contain a digit defining the dimensionality of parameters. In the further part of this chapter we will use the most convenient versions of OpenGL functions.

How colors of the polygon fragments will be determined, depends on the display machine state variables. Some OpenGL state variables define individual vertex properties (the vertex color). The general OpenGL principle is that the property once defined by a call of appropriate function remains valid for all subsequent display operations. If the property related to a vertex is defined then it is applied to all vertices defined by `glVertex*()` calls until the new value of this property is redefined. The simplest way to obtain colored polygon is to explicitly define colors of vertices using `glColor*()` function:

```
glBegin(GL_POLYGON);
   // Define red color for two first vertices
   glColor3f(1.0, 0.0, 0.0);
   glVertex3f(0.0, 0.0, -10.0);
   glVertex3f(3.0, 0.0, -10.0);

   // Define green color for two next vertices
   glColor3f(0.0, 1.0, 0.0);
   glVertex3f(3.0, 3.0, -10.0);
   glVertex3f(1.5, 4.0, -10.0);

   // Define blue color for the last vertex
   glColor3f(0.0, 0.0, 1.0);
   glVertex3f(3.0, 0.0, -10.0);

glEnd();
```

Observe that if the color is defined using *3f* variant of `glColor*()` function then color component intensities are within the range <0.0,1.0>. Various vertex properties can be defined individually by calls of appropriate functions. The most useful properties are listed in Table 11.1.

Command	Purpose of Command
`glVertex*()`	set vertex coordinates explicitly
`glColor*()`	set current color
`glNormal*()`	set normal vector coordinates (for internal lighting mode)
`glEvalCoord*()`	generate coordinates of vertices on the Bezier surface patch
`glTexCoord*()`	set texture coordinates
`glMaterial*()`	set material properties (for internal lighting mode)

Table 11.1. Selected properties that can be assigned to vertices

Internal and external lighting modes

OpenGL can display vertex primitives using vertex colors calculated by internal illumination model or vertex colors can be explicitly specified as in the code example presented in the previous section. The first mode will be called *internal lighting mode* while the second one will be termed *external lighting mode.*

In external lighting mode, colors are calculated by the application that controls OpenGL. The mode must be set by switching internal lighting off. Internal lighting switch is one of OpenGL state variables of Boolean type. Boolean state variables can be set using `glEnable()/glDisable()` functions, which take a constant determining switched property as a parameter. In order to select external lighting mode and to apply interpolated

shading of polygons, the state variable GL_LIGHTING should be set off and GL_SMOOTH variable should be set on. The code framework for displaying smooth shaded polygons in the external mode is as follows:

```
glDisable(GL_LIGHTING);     // swith internal lighting off
glShadeModel(GL_SMOOTH);    // switch color interpolation on
for all polygons
{
   glBegin(GL_POLYGON)
      ...
   glColor3f( red, gre, blue); // set vertex color explicitly
   glVertex3f( x, y, z );      // set vertex coordinates
   ...
   glEnd();
}
glFlush();
```

In order to use external lighting mode, three elements should be defined:
– primary light sources (usually defined for the whole scene before geometry display begins),
– vertex material properties,
– vertex normal vectors.
Having these data defined, the vertex color can be calculated using internal illumination model.

In single pass OpenGL is able to apply only limited number of lights. Each OpenGL implementation defines the light count limit, but at least eight lights can be used. OpenGL allows point lights located at specified positions in the scene space or parallel lights that are located in infinity. Point lights are defined as spot lights that illuminate scene objects located inside a spotlight cone. The cone angle can be however set to 180 degrees which result in the light that uniformly illuminates surrounding objects.

If the light is to be switched on, it must be activated by calling `glEnable()` with the parameter that is the light identifier: GL_LIGHT0, GL_LIGHT1, ... , GL_LIGHT7. Initially the light has all its attributes set to default values (see [20] for detailed default attribute values). `glLight*()` function can be used to define individual light attributes. Single call to this function defines only one attribute of the light. `glLight*()` is declared as:

`void glLight{if}[v](GLenum light, GLenum pname, TYPE param)`[2]
The first argument: `light`, specifies the constant identifying the light affected by this call. It can be one of GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is defined by `pname` argument. It is the constant that defines the light attribute being modified. Constants that can be used here and their meanings are specified in Table 11.2. The last argument (`param`) is the attribute value to be set.

The light attributes may seem somewhat unusual and not physical. This is because light color and intensity is defined independently for each of phenomena reproduced by internal lighting model, i.e. for specular, diffuse and ambient light reflection. Moreover, each point light can emit ambient light that uniformly illuminates all scene objects. It gives extra

[2] The specification of OpenGL functions presented here is excerpted from the book: Neider J., Davies T., Woo M. - OpenGL Programming Guide. Official Guide to Learning OpenGL, Addison-Wesley, 2007

flexibility in scene lighting definition and can sometimes be useful. If it is not necessary, just set light color and intensity to be equal for specular and diffuse reflection and set nonzero ambient light intensity only for single light in the scene.

Parameter Name	Meaning
GL_AMBIENT	ambient RGBA intensity of light
GL_DIFFUSE	diffuse RGBA intensity of light
GL_SPECULAR	specular RGBA intensity of light
GL_POSITION	(*x*, *y*, *z*, *w*) position of light
GL_SPOT_DIRECTION	(*x*, *y*, *z*) direction of spotlight
GL_SPOT_EXPONENT	spotlight exponent
GL_SPOT_CUTOFF	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	constant attenuation factor
GL_LINEAR_ATTENUATION	linear attenuation factor
GL_QUADRATIC_ATTENUATION	quadratic attenuation factor

Table 11.2. Attributes of OpenGL lights

Fig. 11.2. presents the meaning of geometrical attributes of lights. Light position is determined by the point represented in homogenous coordinates. The default position is (0,0,1,0). *w*-coordinate is equal to 0, what corresponds to the light in infinity (directional light like the sun). Spotlight direction is the direction of the cone illuminated by the light. The spotlight cutoff angle is the angle between the spotlight direction and the line on the edge of the cone. Its default value is 180 degrees what corresponds to uniform point light. The spot exponent parameter defines how the spot light intensity attenuates with increasing angle between the illumination direction and the axis of the light cone. The attenuation factor is the cosine of this angle raised to the power of spot exponent attribute.

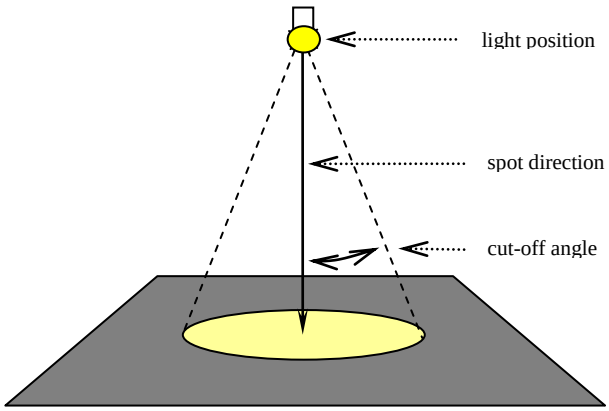


Fig 11.2. Geometrical parameters of OpenGL lights

The formula used to determine the light attenuation with the distance to illuminated surface can be flexibly modeled with three factors: constant, linear and quadratic. The attenuation formula is defined as:

$$f_{acc} = \frac{1}{f_q r^2 + f_l r + f_c}, \tag{11.1}$$

where r is the distance to the point light and f_c, f_l and f_q are attenuation factors defined as GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION and GL_QUADRATIC_ATTENUATION light attributes correspondingly.

It should be emphasized that lights definition is also the element of display machine state. The current definition is used in lighting calculations for vertices being defined. If light attributes are changed in the middle of geometry display process then subsets of vertices will be rendered in varying lighting conditions.

Material attributes once defined will be applied for lighting computations at subsequent vertices. Material attributes are defined by `glMaterial*()` function. OpenGL lighting model assumes that material properties can be defined independently for front and back sides of each polygon. `glMaterial*()` function is defined as follows:

```
void glMaterial{if}[v]( GLenum face, GLenum pname,
                        TYPE param )
```

The `face` parameter can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK. It indicates to which face of the object the material property should be applied. The particular material property being set is identified by `pname` parameter and the desired value for that property is given by `param`. The attributes that can be set are listed in Table 11.3.

Parameter Name	Meaning
GL_AMBIENT	ambient color of material
GL_DIFFUSE	diffuse color of material
GL_AMBIENT_AND_DIFFUSE	ambient and diffuse color of material
GL_SPECULAR	specular color of material
GL_SHININESS	specular exponent
GL_EMISSION	emissive color of material

Table 11.2. Surface attributes used in OpenGL

Similarly as in the case of light definition, surface properties (in particular - the surface color) can be defined independently for diffuse, specular and ambient light reflection. The meaning of parameters is analogous as in the surface model described in Assignment 7.

The polygon sides are distinguished using the polygon vertex order. Whether the side displayed on the screen is front or back, depends on the order of vertices in which they were passed to OpenGL. By default: if vertices appear on the screen as counterclockwise ordered then the visible side is assumed to be the front side. This default can be changed by the call of the function:

```
void glFrontFace( GLenum mode )
```

Calling it with GL_CW constant as an argument, causes that polygons, which vertices are clockwise oriented are assumed to be visible from the front side. Calling the function with GL_CCW constant restores default settings.

Scene rendering in internal lighting mode requires the following sequence of actions to be executed in the program:

```
switch internal lighting model on;
set desired shading mode;
activate as many lights as necessary in the scene (glEnable( GL_LIGHTx );
define light parameters (glLight*());
```

```
for each object
{
   define material properties for the object;
   set modeling matrix;
   for all primitives constituting the object
   {
      indicate the begin of the next primitive (glBegin())
      for all vertices of the primitive
      {
         define vertex normal (glNormal*());
         define vertex coordinates (glVertex*());
      }
      indicate end of primitive (glEnd());
   }
}
```

The code below sets the single light in the scene and defines diffuse material for subsequent vertices. Because the material is purely diffuse, only diffuse illumination component needs to be set for the light source. As far as purely diffuse surfaces are displayed, remaining light intensity attributes are unimportant.

```
001:  glEnable(GL_LIGHTING);     // Enable internal lighting model
002:  glEnable(GL_LIGHT0);       // Enable single light no 0
003:  glShadeModel( GL_SMOOTH ); // Enable smooth Gouraud shading
004:
005:  static float light_diff[4] = { 1.0, 1.0, 1.0, 1.0 };
006:  static float light_pos[4] = { -1.0, -1.0, -13.5, 1.0 };
007:  glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diff);
008:  glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
009:
010:  // Set only diffuse reflection factor to nonzero value
011:  static float mat_diff[4] = { 0.0, 0.0, 1.0, 1.0 };
012:  static float mat_amb[4]  = { 0.0, 0.0, 0.0, 1.0 };
013:  static float mat_spec[4] = { 0.0, 0.0, 0.0, 1.0 };
014:
015:  // Define the same attributes for front and back side
016:  glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT,  mat_amb );
017:  glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE,  mat_diff );
018:  glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, mat_spec );
```

Listing 11.1. Fragment of OpenGL program that sets single light in the scene and defines diffuse material

Transforming objects from local coordinates to 2D image plane

OpenGL essentially implements the rendering pipeline described in Assignment 10. In particular, it carries out a series of geometrical transformations that transform vertex coordinates passed to the pipeline to the 2D coordinates in the frame buffer area. Two matrices are used in this process. The first one is called *modelview matrix* and is responsible for transforming 3D points in local coordinate space to common world space and then to observer coordinates (called in OpenGL tutorials *eye coordinates*). The standard observer position in OpenGL is in the origin of observer coordinate system. It looks along negative *z* axis. Projection plane is the near clipping plane of the view frustum. The second transformation matrix defines the projection and is referred to as *projection matrix*. It converts coordinates in observer coordinate system to clip coordinates where the viewing frustum becomes a normalized box. The resulting coordinate system is called *Normalized Device Coordinates* (NDC). The coordinates of visible scene fragments are now within (-1,1) range. Visible surface determination with Z-buffer is carried out in NDC space and visible fragments are parallelly projected onto the front side of NDC box. Finally, the image within the normalized window ((-1,-1),(1,1)) is transformed to pixel coordinate system by *viewport transformation* taking into account requested image

resolution. The sequence of geometrical transformation involved in the described process is shown in Fig. 11.3.

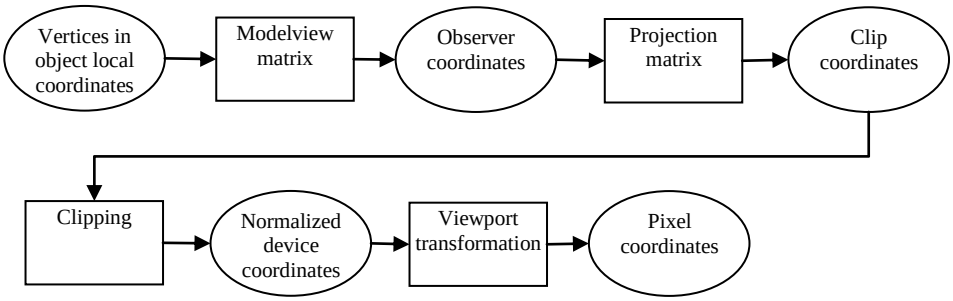


Fig 11.3. Sequence of geometric transformations in OpenGL rendering pipeline.

OpenGL uses right-handed coordinate system. In OpenGL documentation, vertex homogeneous vectors are column vectors. Hence, multiplication of the vector v by the matrix M is denoted by Mv and the sequence of matrix multiplication $M_1 M_2,...,M_n v$ corresponds to execution of transformations in the reversed order, i.e. M_n is the first transformation applied to v and M_1 is the last transformation in the corresponding sequence. OpenGL provides the set of functions that make possible to manipulate the transformation matrices by matrix multiplying or by setting matrix elements explicitly. The operation is executed on the matrix that is currently selected. In order to select the matrix, use `glMatrixMode()` function:

`void glMatrixMode( GLenum mode )`

It selects one of matrices for further operations. The `mode` parameter can be one of `GL_MODELVIEW`, `GL_PROJECTION`, and `GL_TEXTURE`. The following functions can be used to manipulate the selected matrix:

`void glLoadIdentity( void )`

The function loads unit matrix for identity transformation. Identity transformation is the star point to create compound transformations by matrix multiplication.

`void glLoadMatrixd( const GLdouble *m )`
`void glLoadMatrixf( const GLfloat *m )`

These functions make possible to set arbitrary transformation matrix by specifying its elements explicitly. The elements are stored in passed matrix in column order, i.e. the matrix is constructed of `m` array elements as follows:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_0 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

`void glMultMatrixd( const GLdouble *m )`
`void glMultMatrixf( const GLfloat *m )`

The functions multiply the current matrix `C` by the matrix `m` specified explicitly by the function parameter. The result is stored in the current matrix as `Cm`, i.e. the passed matrix is the right operand of the matrix multiplication operator. The order of elements in the matrix `m` is as in the case of `glLoadIdentity*()` function.

```
void glTranslated( GLdouble x, GLdouble y, GLdouble z )
void glTranslatef( GLfloat x, GLfloat y, GLfloat z )
```

These functions multiply the current matrix `C` by the translation matrix `m` corresponding to translation by the vector `(x,y,z)`. The result `Cm` is stored in the current matrix. Pay attention to the order of multiplied matrix - the transformations are combined in the order opposite to the order of executed matrix multiplications.

```
void glRotated( GLdouble angle,
                GLdouble x, GLdouble y, GLdouble z )
void glRotatef( GLfloat angle,
                GLfloat x, GLfloat y, GLfloat z )
```

These functions multiply the current matrix by the rotation matrix corresponding to the rotation by the angle specified by the first parameter about the axis defined by the point `(x,y,z)` and by the origin (0,0,0). The rotation angle is defined in degrees. Positive rotation is counterclockwise. The result is stored in the current matrix.

```
void glScaled( GLdouble x, GLdouble y, GLdouble z )
void glScalef( GLfloat x, GLfloat y, GLfloat z )
```

These functions multiply the current matrix by the transformation matrix `m` corresponding to scaling by factors `x,y,z` along axes. The result is stored in the current matrix.

Modeling transformation can be created by combining elementary transformations with described above functions. The transformation from world to eye coordinates can be also composed of elementary transformations as described in Assignment 8. More convenient way is to use `glLookAt()` function:

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

It multiplies the current matrix by the world-to-observer complete transformation matrix. The observer position in world coordinates, view direction and vertical direction in the image are specified by passed parameters as shown in Fig. 11.3. Due to interpretation of matrix multiplication operations with column vectors, the world-to-observer transformation matrix should be set before modeling transformations are defined with matrix multiplication functions.

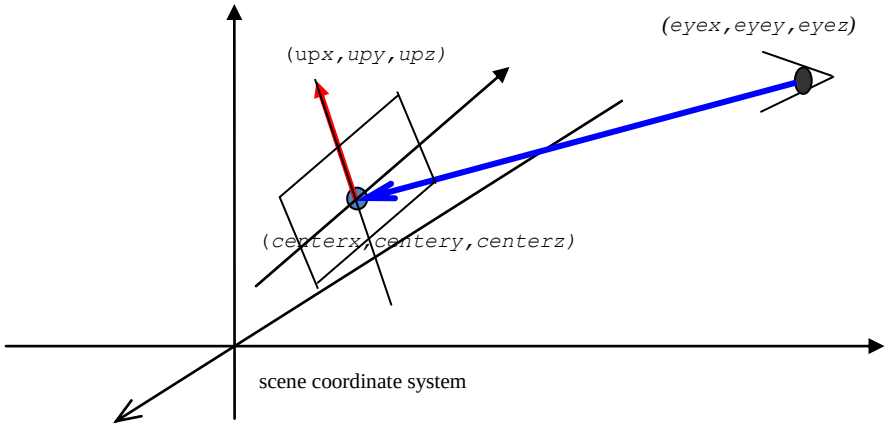


Fig.11.3. `gluLookAt()` function parameters interpretation.

Projection matrix can be also created by using general purpose matrix manipulation functions. This transformation however is expected to fit the intended view frustum into NDC box and it also affects Z-buffer operations by nonlinear scaling of *z*-coordinate. Therefore it is strongly recommended (at least for not experienced OpenGL programmers) to use predefined functions that set projection matrix in more natural and easy to use way. In order to set perspective transformation use one of `gluPerspective()` or `glFrustum()` functions:

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

It sets the projection matrix in a flexible way by explicitly specifying near and far clipping plane and by specifying left, right, bottom and top image window edges on the front clipping plane. Observe that the view frustum does not have to be symmetric. This feature can be used to cover the image of very big resolution by fragments rendered independently. The interpretation of function parameters is explained in Fig. 11.4.

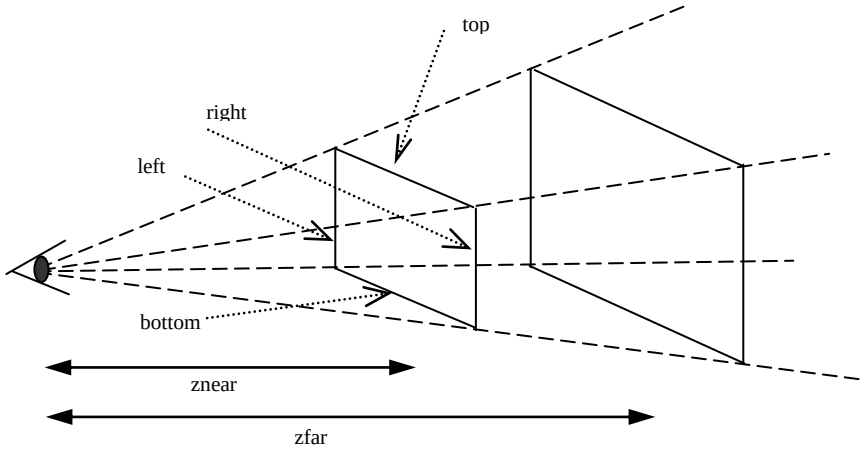


Fig.11.4. `glFrustum()` function parameters interpretation.

```
void gluPerspective( GLdouble fovy,  GLdouble aspect
                     GLdouble znear, GLdouble zfar )
```

This function multiplies the current matrix by the perspective projection matrix corresponding to the projection defined by: a) the field of view angle in horizontal plane, b) the image aspect ratio (width/height) and c) the location of clipping planes. The interpretation of parameters is shown in Fig. 11.5.

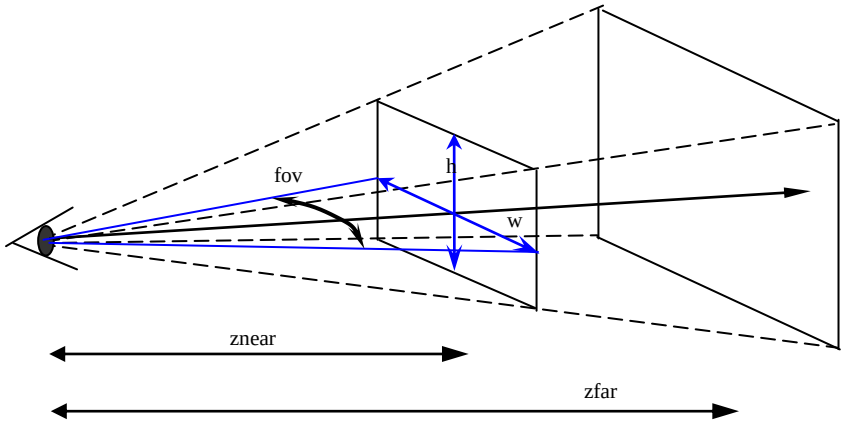


Fig.11.5. `gluPerspective()` function parameters interpretation.

The last step of transformation setting consists in defining viewport transformation. It determines the resolution and position of the image in the output window. Viewport setting can be done with `glViewport()` function:

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

It established the resolution of the output image (`width, height`) and specifies the lower left vertex (`x,y`) of the image area within the window. The image aspect ratio set with `glViewport()` should be the same as when defining projection, otherwise the image will be distorted.

Matrix stacks

Modelview and projection matrices used by OpenGL are located on the top of matrix stacks. There are two independent stacks: one for modelview matrices and one for projection matrices. Minimum size of modelview matrix stack is 32. Projection matrix stack is able to contain at least two matrices. In particular implementations these numbers can be greater. Initially both stacks contain single unit matrix. Matrix stacks can be manipulated with two functions:

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

The operations is always performed on the stack corresponding to the matrix mode selected with `glMatrixMode()` function. `glPushMatrix()` pushes the new matrix onto the stack and fills it by copying the matrix that were located on the stack top before the operation was invoked. `glPopMatrix()` just removes the matrix from the stack. The matrix that was below the top of the stack becomes the current one.

Using aux library to set up OpenGL display window

Tedious operations necessary to initialize the window used as OpenGL output and to bind the window drawing context with OpenGL rendering context can be simplified with `aux` library. The library makes possible to create the window and easily link it to OpenGL rendering context, organize simple interaction based on mouse and keyboard events and to

display simple geometry objects like balls, toruses, boxes and other polyhedrons. Short preview of the most useful `aux` functions is presented below:

`void auxInitDisplayMode( GLenum mask )`
This function initializes a window with required properties necessary for OpenGL operations. `mask` parameter determines required window properties. It is a combination of bit flags defined by the constants:
– AUX_RGBA , AUX_INDEX - determine the required color modes,
– AUX_SINGLE, AUX_DOUBLE - determine if single buffer or double buffer mode is required,
– AUX_DEPTH, AUX_STENCIL, AUX_ACCUM - determine which buffers should be allocated in the window for OpenGL.

`void auxInitInitPosition( GLint x, GLint y,`
`GLint w, GLint h )`
This function sets the size and position of the image output window.

`GLenum auxInitWindow(char *titleString)`
The function creates the window according to the window properties defined earlier by `auxInitDisplayMode()`. The window title is specified by `titleString` parameter.

`void auxMainLoop(void (*displayFunc)(void))`
It sets the main image drawing function where all OpenGL operations should be embedded. The function defined with this call is executed whenever the window manager detects the need to refresh the window image.

`void auxReshapeFunc(void (*function)(GLsizei, GLsizei))`
It sets the function that will be called when the window size or aspect ratio is changed. Typically, the called function should update the perspective projection to fit it to the window sizes.

`void auxIdleFunc(void (*func)(void))` –
It defines a function executed when no other activity is pending. The function defined by this call can be used to organize simple animation by changing some scene elements.

`void auxSwapBuffers(void)`
This function swaps front and back buffers. If rendering results are directed to the back buffer then quick swapping of buffers prevents image flickering.

`void auxCloseWindow(void)`
It closes a window created by `auxInitWindow()`.

`void auxKeyDownFunc(int key, void (*function))`
It defines a handler function called in response to the specified key press event.

`void auxMouseFunc( int button, int mode, void`
`(*function)(AUX_EVENTREC *))`
It defines a handler function called when the mouse event occur. The event that should be signaled is specified by `button` and `mode` parameters. Possible values of button

parameter are: AUX_NOBUTTON, AUX_LEFTBUTTON, AUX_MIDDLEBUTTON, or AUX_RIGHTBUTTON. Which event is signaled is defined by `mode` parameter. AUX_MOUSELOC, AUX_MOUSEDOWN or AUX_MOUSEUP parameter values are allowed. AUX_MOUSELOC represents mouse movement. Remaining values correspond to mouse button events.

`void auxGetMouseLoc(int *x, int *y)`
It reads the cursor position in relation to the window.

`void auxWireSphere( GLDouble radius ),`
`void auxSolidSphere( GLDouble radius )`
These functions render a sphere with the specified radius. The sphere can be rendered in wireframe mode or in smooth shading with interpolated vertex normals. Similar functions exist also for cubes, toruses, cones, cylinders, Haynes teapots and other simple shapes. See [20] for detailed specification.

The following example shows how to use `aux` library to display simple animated scene. The animation consists in rotating objects about their axes and in moving objects along circular orbits. Pay attention to transformation combining in lines 101-121. Recall that transformation matrices must be combined in the reversed order. Observe how selected material attributes are redefined between object display operations (lines 76-81 and 123-126).

```
 1:   #include <GL/glaux.h>
 2:
 3:   // The rotation angle for shapes animation
 4:   GLfloat rotation = 0.0;
 5:
 6:   // ====================================================================
 7:   // This function adapts projection transformations to current window
 8:   // resolution and aspect size
 9:   // ====================================================================
10:   void CALLBACK reshape_scene( GLsizei width, GLsizei height )
11:   {
12:      // Set OGL viewport to cover whole actual window area
13:      glViewport( 0, 0, width, height );
14:
15:      // Modify projection transformation
16:      glMatrixMode( GL_PROJECTION );
17:
18:      // Initialize projection matrix because gluPerspective multiplies
19:      //   current matrix on the stack by the constructed projection
20:      //   matrix
21:      glLoadIdentity();
22:
23:      // Set field of view equal to 30 degres and adapt
24:      //   window aspect ratio. Set clipping planes so as to enclose
25:      //   the complete geometry to be displayed
26:      gluPerspective( 30, width/(double)height, 5, 500 );
27:
28:      // Switch back to modelview matrix stack
29:      glMatrixMode( GL_MODELVIEW );
30:   }
31:
32:   // ====================================================================
33:   // The complete single frame rendering function
34:   // ====================================================================
35:   void CALLBACK draw_scene(void)
```

```
36:   {
37:      // Vectors for material attributes:
38:      // Diffuse reflectinon factors:
39:      static float mat_diff[4] = { 0.0, 0.0, 0.7, 1.0 };
40:      // Specular reflection factors:
41:      static float mat_spec[4] = { 1.0, 1.0, 1.0, 1.0 };
42:      // Ambient light reflection factors:
43:      static float mat_amb[4]  = { 0.0, 0.0, 0.0, 1.0 };
44:
45:      // Data for light parameters setting:
46:      // Diffuse reflection:
47:      static float light_diff[4] = { 1.0, 1.0, 1.0, 1.0 };
48:      // Specular reflection:
49:      static float light_spec[4] = { 1.0, 1.0, 1.0, 1.0 };
50:      // Light position
51:      static float light_pos[4] = { 10.0, 10.0, 0.0, 1.0 };
52:
53:      // Enable Z-buffer test
54:      glEnable( GL_DEPTH_TEST );
55:
56:      // Enable internal lighting
57:      glEnable( GL_LIGHTING );
58:
59:      // Set Gouraud smooth shading
60:      glShadeModel( GL_SMOOTH );
61:
62:      // Enable a single light
63:      glEnable( GL_LIGHT0 );
64:
65:      // Set buffer fill color,
66:      //   then initialize frame and depth buffers
67:      glClearColor( 1.0, 0.2, 0.2, 0.0 );
68:      glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
69:
70:      // Set prepared light attributes of the tight no 0
71:      glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diff);
72:      glLightfv(GL_LIGHT0, GL_SPECULAR, light_spec);
73:      glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
74:
75:      // Set material attributes to obtain blue glossy material
76:      mat_diff[0] = 0.0; mat_diff[1] = 0.0, mat_diff[2] = 0.7;
77:      glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT,   mat_amb );
78:      glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE,   mat_diff );
79:      glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR,  mat_spec );
80:      glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT,   mat_amb );
81:      glMaterialf(  GL_FRONT_AND_BACK, GL_SHININESS, 40 );
82:
83:      // Switch to modelview matrix manipulation
84:      glMatrixMode(GL_MODELVIEW);
85:      glPushMatrix();
86:
87:      // Create modeling transformation for a sphere
88:      glLoadIdentity();
89:      // Translate in the direction of observation
90:      //   to make the object more distant from observer
91:      glTranslatef(0.0, 0.0, -10.0);
92:      // Rotate about the axis (1,1,0)
93:      glRotatef(30, 1.0, 1.0, 0.0);
94:      // Rotate by dynamically changing angle about Z axis
95:      glRotatef(rotation, 0.0, 0.0, 1.0);
96:
97:      // Draw sphere using aux drawing utility
98:      auxSolidSphere(1.2);
99:
```

```
100:
101:      // Create modeling transformation for a spinning box.
102:      // We want to get the box spinning around its axis (1,1,1)
103:      //   and circulating about the sphere.
104:
105:      // Start with transformation re-initializing
106:      glLoadIdentity();
107:      // Translate in the direction of observation
108:      //   to make the orbit center at the center of the sphere to get
109:      //   the box circulating about the sphere
110:      glTranslatef(0.0, 0.0, -10.0);
111:      // Rotate the orbit plane about OX axis
112:      glRotatef(30, 1.0, 0.0, 0.0);
113:      // Rotate the box to obtain the effect of circulation on the orbit
114:      //   located on the plane XOZ, i.e. rotate about OY axis
115:      glRotatef(rotation, 0.0, 1.0, 0.0);
116:      // The box is to be circulating on the orbit of the radius 2.0
117:     //   about the axis (0,1,0), so translate it by the required radius
118:      glTranslatef(0.0, 0.0, -2.0);
119:      // Rotate by dynamicly changing angle about (1,1,1) diagonal
120:      //   to obtain spinning effect
121:      glRotatef(rotation, 1.0, 1.0, 1.0);
122:
123:      // Change current material to glossy red
124:      //   It will be applied to the cube.
125:      mat_diff[0] = 0.7; mat_diff[1] = 0.0, mat_diff[2] = 0.0;
126:      glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE,   mat_diff );
127:
128:      // Draw the cube using aux drawing utility
129:      auxSolidCube( 1.0 );
130:
131:      // Restore modelview matrices
132:      glPopMatrix();
133:
134:      // The whole rendering was directed to the back buffer.
135:      //   Now swap buffers to make the image visible in the window
136:      auxSwapBuffers();
137:   }
138:
139:
140:   // ========================================================================
141:   // This function is responsible for animation. It periodically
142:   // modifies the rotation angle used in modeling transformation
143:   // ========================================================================
144:   void CALLBACK rotate_objects(void)
145:   {
146:      // Increase the rotation angle by 1 degree
147:      rotation += 1.0;
148:      if (rotation >= 360.0)
149:        rotation -= 360.0;
150:
151:      // Force image to be redrawn
152:      draw_scene();
153:   }
154:
155:   // ========================================================================
156:   // OpenGL window initialization is carred out in main() function
157:   // ========================================================================
158:   void main(void)
159:   {
160:      // Use RGBA color representation on the frame buffer,
161:      //   Request double buffering for flicker-free display,
162:      //   Request dept buffer allocation
163:      auxInitDisplayMode( AUX_RGBA | AUX_DOUBLE | AUX_DEPTH );
```

```
164:
165:        // Display the window

166:        auxInitPosition( 100, 100, 400, 400 );
167:        auxInitWindow( "Simplest OpenGL program" );
168:
169:        // Define callback functions
170:        auxReshapeFunc( reshape_scene );
171:        auxIdleFunc( rotate_objects );
172:        auxMainLoop( draw_scene );
173:    }
```

Listing 11.1. Simple OpenGL program based on aux library

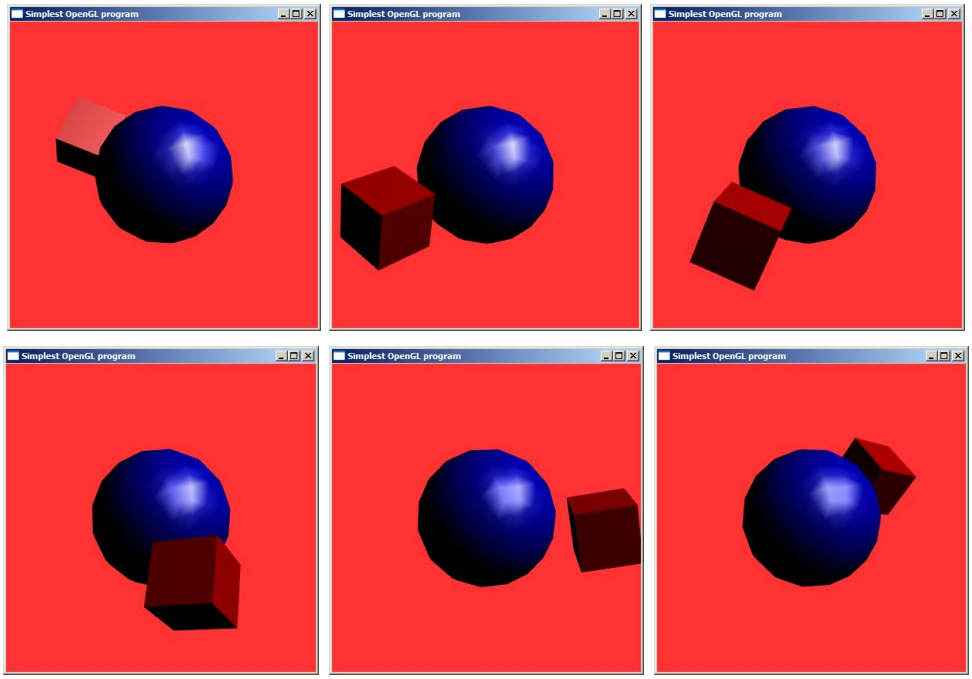


Fig. 11.6. Screenshots of the program from Listing 11.2. at subsequent shape motion stages.

Assignment scope

1. Implement OpenGL application that displays animated bouncing ball. Arrange the ball motion as a superposition of uniformly variable motion in gravity field along OY axis and uniform motion along the circle in XOZ plane. Use `aux` library to organize the animation and to display the ball shape.

2. Implement a simple application that displays a set of balls arranged into the uniform grid in XOZ plane. Apply various material attributes to balls. Using `aux` support for mouse events handling make possible to change the observer location on the hemisphere that surrounds the grid of balls. The position on the hemisphere can be specified with two angles (φ,θ). Let mouse movements along horizontal axis change φ angle. Change θ angle by vertical movements of the mouse. Fix observer target point in the origin.

3. Implement a test program for OpenGL performance experiments similar to the program that draws random triangles from Assignment 6. Now use OpenGL to display random triangles. Use orthographic projection. In order to achieve it, either find OpenGL function that sets orthographic projection or arrange appropriate projection matrix and set if using matrix operations in your code. Compare the performance of the program implemented in Assignment 6 with the performance of its improved version based on OpenGL.

4. Rebuild the program from Assignment 8 so as to use OpenGL for orthographic and perspective views. Display the perspective view using GL_POLYGON display mode. Draw orthographic views in the wireframe mode.

Assignment 12

Simple ray tracing program

Aim

The aim of this assignment is get to know the parading of backward ray tracing, experience its capabilities and limitations, master the process of scene data tuning so as to obtain required visual effects, to learn techniques and methods used in backward ray tracing implementation and to test ray tracing efficiency with various sort of scenes.

Theoretical fundamentals

Backward ray tracing (popularly known as *ray tracing* or RT for short) was primarily presented in its limited form by Appel in 1968 ([1]). Originally, it was proposed as a method of visible surface determination and sharp shadow rendering by *ray casting*. Ray casting consists in finding the scene object visible from the specified location in the specified direction. Computationally, it consists in finding an intersection of the half-line with one of scene object surfaces, which is closest to the half-line origin. The concept of Appel was then extended by Whited in 1982, by ray casting application to directions of light rays that are specularly reflected on the mirror-like surfaces and transmitted by refracting surfaces. The lighting model was extended with components related to the contribution of specularly reflected and refracted ray to the visible point lightness. The observed color is calculated by the recursive procedure that computes the color of directly visible fragment by applying recursively the same lighting model formula to objects visible in the direction of reflection and refraction. The method reproduces visual effects of sharp shadows and light reflection and refraction, what essentially contributes to the realism of the rendered image. Because the method simulates light transport in the scene in physically accurate way, the effects are reproduced very precisely. Therefore, the method can be considered as the first photorealistic 3D rendering technique. The examples of typical RT effects are shown in Figures 12.1-12.6.

Additional advantage of the method is that it can be easily extended with simulation of other lighting phenomena, like e.g. volumetric effects (lighting effects caused by light interaction with ambient media like fog, smog, clouds), soft shadows, depth-of-field and many others. It can be also combined with other lighting simulation methods that analyze other kinds of light transport paths not taken into account in backward RT like: multiple light scattering on diffuse (Lambertian) surfaces simulated by radiosity method ([7]) and light focusing and indirect illumination simulated with photon tracing approach ([15]).

Unfortunately the method is computationally extensive due to the necessity of testing great number of scene objects for intersection with cast rays. For this reason, the method did not gained much popularity just after presenting its concepts by Appel and Whited. The situation changed with introduction of acceleration techniques that reduced the number of intersection tests by a few orders of magnitude. Some acceleration methods are described in [5], [8] and [14]. Progress in acceleration techniques and dramatic growth of computer computational capabilities made possible to carry out RT rendering on PC-class computers,

even for complex scenes consisting of millions of geometric primitives. Nowadays, backward RT is used as the basic method in the majority of photorealistic 3D rendering systems.

Fig. 12.1. RT effects - hard shadows.[3]

Fig. 12.2. RT effects - mirror reflections on the flat surface. Mirror-like attributes are assigned to the floor.

[3] Geometry model used to render this image: courtesy of Integra Inc. Japan.

Fig. 12.3. RT effects - mirror reflections on the surface that is both diffuse and specular.

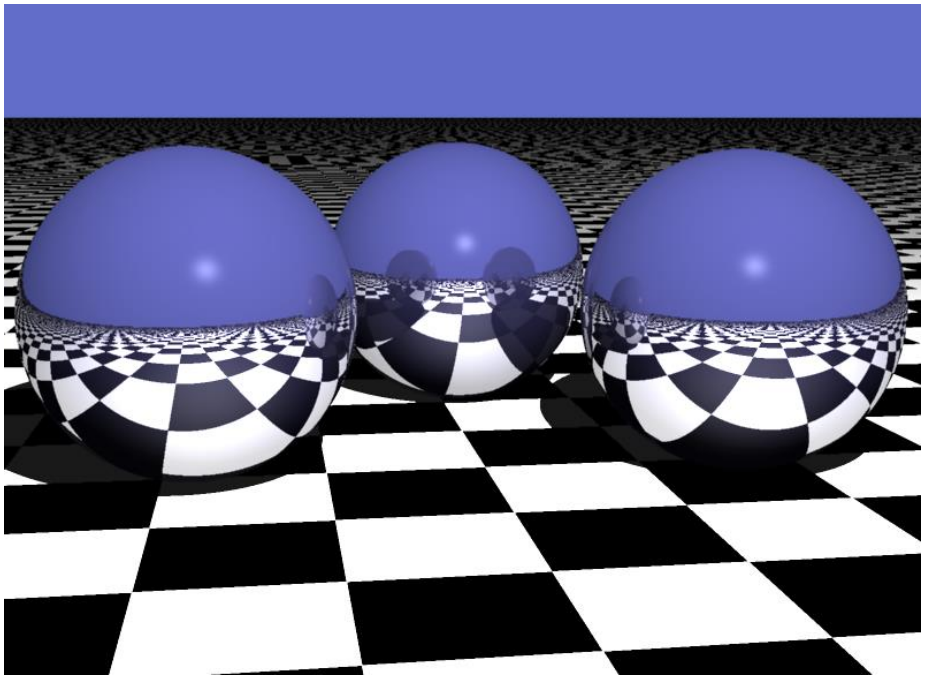


Fig.12.4. RT effects - mirror reflections on smooth curved surfaces.

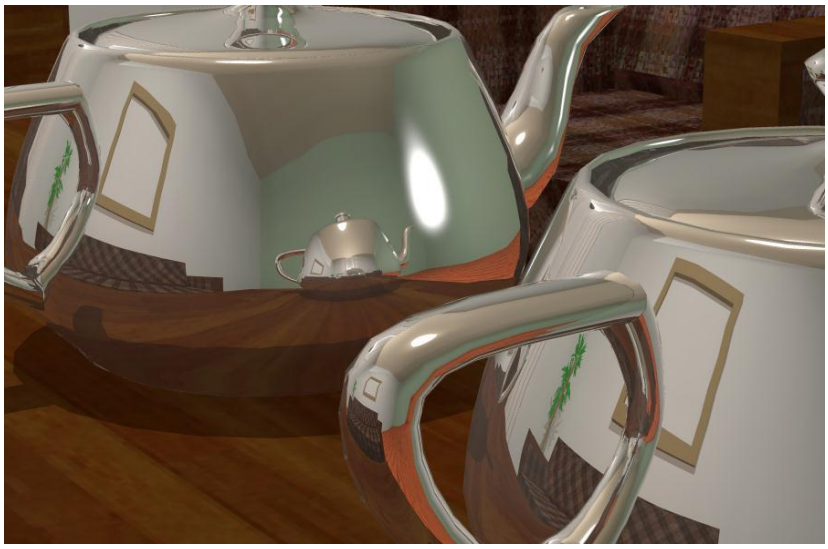


Fig.12.5. RT effects - multiple mirror reflections.

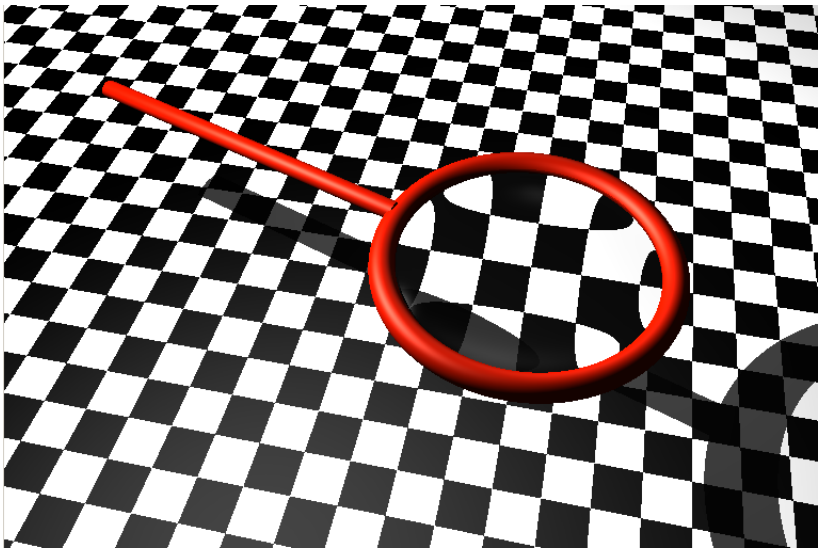


Fig. 12.6. RT effects - light refraction. The textured surface is observed through the magnifying glass.

Principles of backward ray tracing

The fundamental principle of RT consists in finding possible paths of lights from primary point light sources to the observer. Only the paths corresponding to: a) direct illumination of diffuse and specular surfaces by primary lights, b) specular reflection and c) transmission of light through transparent refracting surfaces are taken into account. Light paths are however traced in the reversed order: from the observer to the light source. Hence, the method is named *backward ray tracing*. Each ray carries the light energy that finally comes up to the observer and determines the brightness and color of the perceived image fragment. In order to render a raster image, the colors of individual pixels are computed by analyzing the light paths that come to the observer and go through pixel centers. The light

path segments: a) between surface fragments, b) from the observer to the directly visible surface fragment and c) from the surface fragment to the light source are called *rays*. The ray from the observer to the visible surface fragment is a *primary ray*. Reflected and refracted rays are *secondary rays*. Rays from the diffuse surface fragments to the primary light sources are *shadow rays*. Light diffusion, specular reflection and refraction can concurrently appear on the same surface. In result, rays traced for the pixel in the raster image constitute tree-like structure where the root represents the observer and leaves represent primary lights or events consisting in tracing rays out of domain. The exemplary ray tree is presented in Fig. 12.7.

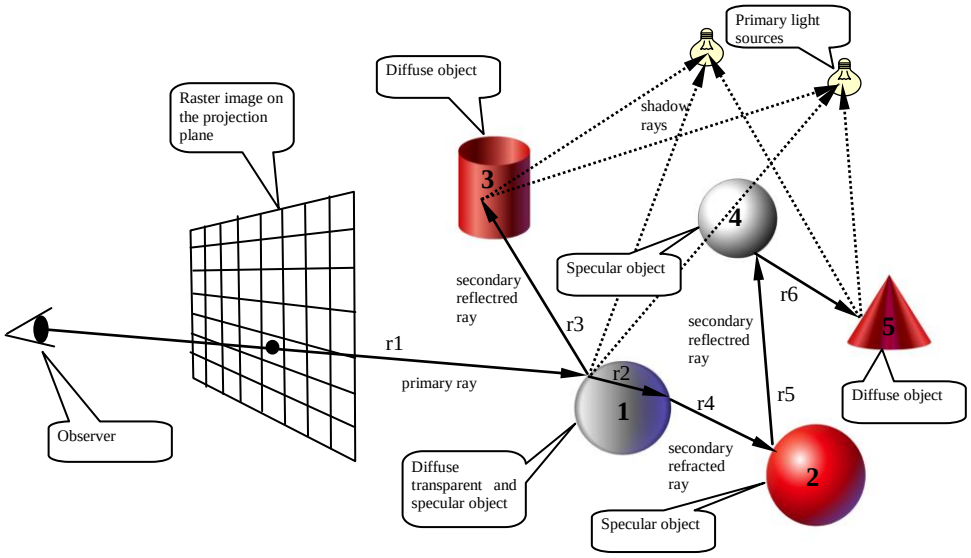


Fig. 12.7. Recursive ray tracing and ray-tree construction.

The ray tracing procedure shown in Fig. 12.7 starts with determining the primary ray direction r_1 for the pixel currently being processed. The primary ray origin is always in the observer point. The direction is determined by the position of the pixel center on the projection plane located in the scene space. In the case of RT rendering, there is no need to transform to the observer coordinate space. The primary ray intersects the surface of the object 1 that appears to be both specular and transparent (e.g. it can be a glass ball). Therefore at the point of intersection two secondary rays are started. The first secondary ray traces the path of specularly reflected light. The second one corresponds to light refraction. For both of these rays we need to know the perceived colors of objects seen along ray directions from their origin. This is exactly the same problem as we initially stated for the primary ray. Therefore the procedure can be used recursively to find out the colors corresponding to observation along the direction of rays r_2 and r_3. The secondary ray colors will be then combined with the color calculated at the intersection of the primary ray r_1 with the object 1 surface using the lighting model of the surface 1.

The recursive procedure traces the ray r_3 to the intersection point with the object 3. The surface in the intersection point is purely diffuse. No further secondary rays are raised but two shadow tests are traced to primary lights. The refracted ray r_2 hits again the transparent surface and is refracted once again. At the intersection point the next refracted ray r_4 is raised. It reaches the purely specular surface of the object 2. Next reflected ray r_5 is created

and traced to its intersection with the specular object 4. The reflected ray r_6 finally hits purely diffuse surface of the object 5, where only shadow tests are traced to primary lights.

Lighting model for RT

Any lighting model can be used in RT rendering, provided that the components related to reflected and refracted rays are taken into consideration. We will adapt the Phong model explained in Assignment 7. The Phong formula can be extended in the following way:

$$L_C = S_C + k_{dC}\sum_{i=1}^{n} f_{acc}^{RT}(r_i)E_{iC}N \bullet I_i + k_{sC}\sum_{i=1}^{n} f_{acc}^{RT}(r_i)E_{iC}(I_i \bullet O_S)^g + k_{aC}A_C + k_{sC}L_C(O_S) + k_{tC}L_C(O_R) \quad (12.1)$$

The component representing the light that comes from the specular reflection direction is $k_{sC}L_C(O_S)$, where $L_C(O_S)$ denotes the result of the recursive application of the same formula to the reflected ray O_S. The term $k_{tC}L_C(O_R)$ models the contribution of the refracted light coming from the direction of the refracted ray O_R. $L_C(O_R)$ is the result of recursive application of Phong formula to the surface fragment visible in the direction of the refracted ray O_R. k_{tC} is the surface transmittance factor for the color component *C*.

The light attenuation formula f_{acc}^{RT} now includes the factor representing light attenuation on the surfaces intersected by the shadow ray. We neglect here the volumetric effect consisting in light attenuation on the light path inside solids, where the light interacts with the material. Such media are called *participating media*. In the simplified approach let as assume that the light in attenuated only when the object surface is crossed. We also do not take into account the light attenuation by the medium. The formulas that precisely model the physics of light attenuation in participating media can be found in [14]. The adapted simplified attenuation formula for light attenuation is:

$$f_{acc}^{RT}(r) = \prod_j k_{tC}^j \min\left(\frac{1}{c_2 r^2 + c_1 r + c_0}, 1\right), \quad (12.2)$$

where *j* is the index of he surface intersected by the shadow ray and k_{tC}^j is the transmission factor for the *j*-th intersected surface.

The recursive procedure that implements the formula (12.1) can be implemented as the subroutine that computes the color corresponding to the ray, i.e. the color that would be seen when looking at the ray direction from the point of the ray origin. The procedure can be implemented according to the following pseudocode:

```
Color CalculateColor( Point3D P0, Vector3D u )
{
   Color  color;

   find the object d intersected by a ray (P0,u);
   if no such object
      return background_color;

   find the point P' being an intersection of the object d with the ray;
   find the normal N vector at the intersection point;

   // Initialize the accumulated color with ambient comonent
```

```
   color = k_aC(d) * A;

   // Trace refracted ray for surfaces with nonzero transmittance factor
   if (k_tC(d) > 0)
   {
      determine the direction of the refracted ray u';
      color += k_tC(d) * CalculateColor( P',u')
   }

   // Trace reflected ray if the surface exhibits specular properties
   if (k_sC(d) > 0)
   {
      determine the direction of reflected ray u';
      Color += k_sC(d) *  CalculateColor( P',u')
   }

   // accumulate direct illumination from primary lights
   for each point light i
   {
      test if there is any other opaque object between P
         and the point light i and calculate the attenuation
         factor f_acc;
      if ( point P' not in the shadow )
         color += f_acc *( E_i*(k_dC(d)*(NO_i) + k_sC(d)*(u'O_i)^g(d)));
   }

   return color;
}
```

Finding intersections of rays with scene objects

The most extensively used computation element in RT is finding the object intersected by the ray. In the case of primary, reflected and refracted rays we are interested in the intersected object which intersection point is closest to the observer. In the case of shadow rays we are interested in any intersection with completely opaque surface or, if there is no such surface, in all semi-transparent surfaces intersected by the line segment of the shadow ray. In order to obtain efficient implementation of the ray tracer, it is extremely important to take care of the efficiency of the intersection procedure.

Ray intersection with quadrics

Quadric is the surface described by the second order polynomial of $P = (x, y, z)$:

$$P^T Q P + P^T S + h = 0 \tag{12.3}$$

where Q is *3x3* matrix and P is a vector. Geometrical primitives used in CSG geometry representation are usually solids bounded by quadrics. Spheres, cones, cylinders and planes belong to surfaces that can be represented by the equations from the class defined by (12.3). Quadrics are the particular case of implicit surfaces, i.e. the surfaces defined by the equation $F(x, y, z) = 0$.

Finding the intersection of the ray (P_O, u) with a quadric surface consists in putting the parametric representation of points on the ray $P = P_O + ut$ to the surface equation (12.3):

$$(P_O + ut)^T Q(P_O + ut) + (P_O + ut)^T S + h = 0\,. \tag{12.4}$$

By simple rearrangements we obtain:

$$At^2 + Bt + C = 0, \tag{12.5}$$

where:

$$A = u^T Qu, \quad \mathrm{B} = \mathrm{P}_O^{\mathrm{T}} Qu + u^T QP_O + u^T S, \quad C = P_O^T QP_O + P_O^T S + h. \tag{12.6}$$

The quadratic equation (12.5) can be solved with respect to *t*, giving 0, 1 or 2 solutions. If there are no solutions then it means that the ray misses the surface. If there is only single solution then it means that the ray is tangent to the surface. We will also assume in this case that there is no ray-surface intersection. In the case of two solutions t_1 and t_2, we are interested only in these values that are greater than 0, because intersections on the back side of the ray are not being considered. In the case of primary or secondary rays the finally found intersection point corresponds to this value t^* of $\{t_1, t_2\}$ which is smaller and is greater than 0. The coordinates of the intersection point are obtained by putting t^* into the ray equation.

Ray intersection with triangles

Finding the intersection of the ray with a triangle consists of two stages. At the first stage the intersection with the plane of the triangle is found. The parameter value t^* corresponding to the intersection point in the ray equation can be calculated using the formula (8.4) from Assignment 8. Then the coordinates of the intersection point can be calculated by putting into the ray equation $P = P_O + ut^*$. At the second stage, it has to be tested if the ray-plane intersection point is located inside the triangle and if it belongs to the ray fragment we are interested in. If the found t^* parameter value is negative then the intersection is on the wrong side of the ray origin and the test is immediately terminated. The test can be also terminated immediately with the negative result if we are interested only in intersection points within certain interval of *t* parameter and the found value is not within this interval. It is the case when we use the test in order to find the closest object along the primary or secondary ray and we have already found the intersection with another object for which $t < t^*$.

Point-inside-triangle test for the triangle located on the plane in 3D space can be reduced to the equivalent test on 2D plane. The triangle and the point being tested can be orthographically projected onto one of XOY, XOZ or YOZ planes. The selected projection plane is called *driving plane*. Any of XOY, XOZ, YOZ planes can be used, which is not perpendicular to the triangle plane. If it is satisfied then the triangle projection on the driving plane is also a triangle. Many methods to solve point-inside-triangle test in 2D have been proposed. They differ in number of arithmetic operations necessary to complete the test. One possible approach is based on checking if the point is on the correct sided of lines which contain triangle edges. The triangle is the common part of three hemiplanes defined by triangle edges. The point is inside the triangle if for all three triangle edges it is on the same side of the hemiplane edge that contains the triangle. Therefore to test if the point is inside the triangle it is sufficient to test on which side of edge lines the point is located. The elements used in the test on 2D driving plane are shown in Fig. 12.8.

In order to efficiently test the point location in relation to the edge line, the edge line equation should be prepared at the preprocessing stage. For vertical and horizontal edges the equations are correspondingly: $x = d$ and $y = d$. For all other edges the equation can

be written in the form: $ax + y = c$. a and c factors can be computed based on the two edge vertices (x_1, y_1) and (x_2, y_2):

$$a = \frac{y_2 - y_1}{x_1 - x_2}; \quad c = \frac{y_2 x_1 - y_1 x_2}{x_1 - x_2}. \tag{12.5}$$

To test if the point is on the correct side of the edge line it is sufficient to test if $ax + y \geq c$ if the triangle is above the edge or $ax + y \leq c$ if the triangle is below the edge. To obtain the uniform test for both cases, the inequality for the latter case can be multiplied by *-1* and then for both cases the same test $a'x + by \geq c'$ can be used where:

– $a' = a,\ b = 1,\ c' = c$ if the triangle is located above the edge and

– $a' = -a,\ b = -1,\ c' = -c$ if the triangle is located below the edge.

In order to quickly complete the test for some points, *the triangle bounding rectangle* can be used. The triangle bounding rectangle can be constructed by finding smallest intervals (x_{min},x_{may}) and (y_{min},y_{max}) containing all vertex x and y coordinate values correspondingly. Then the quick test that makes possible to eliminate points from further test is:

$$(x < x_{\min}) \vee (x > x_{\max}) \vee (y < y_{\min}) \vee (y > y_{\max}). \tag{12.9}$$

If this predicate is satisfied then the point is obviously out of the triangle. The test procedure is therefore as follows (for the sake of simplicity we use *(x,y)* coordinate symbols, but depending on the triangle driving plane, it can *be (x,y), (y,z)* or *(x,z)*).

```
project the triangle onto the driving plane;
if (x <xmin) or (x >xmax) or (y<ymin) or (y> ymax)
   the point is out of the triangle
else
   if ( a1'x+b1y < c1')
      the point is out of the rectangle;
   else   if ( a2'x+b2y < c2')
      the point is out of the rectangle;
   else   if ( a3'x+b3y < c3')
      the point is out of the rectangle;
   else the point is inside the rectangle;.
```

In the most optimistic case the test consists just of one floating point comparison. In the most pessimistic case (the point is inside the triangle) the test requires 7 comparisons, 3 multiplications and 3 additions. All necessary data for triangle intersection test should be prepared in the preprocessing stage, and stored in appropriate data structures for further usage in the rendering process. In this way the amount of calculations that have to be executed "per intersection test" in the rendering time is minimized, thus the rendering time is shortened. The following data could be precalculated:

– triangle plane normal *N* and the plane distance *d* to the coordinate system origin,

– triangle driving plane index,

– 2D triangle edge line equation factors,

– triangle bounding rectangle *(x_{min},x_{max},y_{min},y_{max})*.

The procedure of finding the intersection of the ray with the objects closest to the observer that avoids unnecessary computations can be organized as follows:

```
tmin=∞;
for all triangles in the scene
{
   find t* for the intersection with the triangle plane;
```

```
   if t* >= tmin
      continue;
   find two coordinates of the intersection point corresponding to
      the triangle driving plane;
   if the intersection point inside the triangle on the driving plane
   {
      tmin = t*;
      store the index of the current triangle;
   }
}
if tmin = ∞
    no triangle intersected by the ray;
else
{
   calculate remaining coordinate of the intersection point
      using tmin and the ray equation;
   the recently stored triangle index is the index
      of the triangle intersected by the ray;
}
```

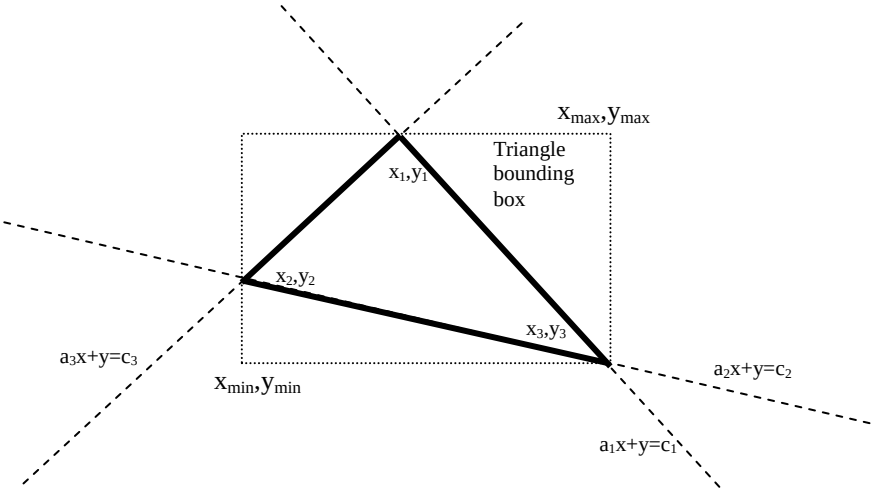


Fig. 12.8. Geometric elements used in point-inside-triangle test on 2D plane

Shooting primary rays

In order to obtain the perspective projection in RT it is convenient to specify the observer (camera) parameters by giving four points: the location of the observer P_O and position of three screen parallelogram vertices in the scene space: P_{UL} - upper left vertex, P_{LL} - lower left vertex and P_{UR} - upper right vertex. Moreover, using such representation of the camera parameters is convenient for interactive camera presentation and manipulation. There is no need to introduce the observer coordinate space in RT.

Primary rays are traced in RT from the observer point through the centers of raster image pixels. The raster image covers the parallelogram defined by P_{UL}, P_{LL} and P_{UR}. Let x_{res} and y_{res} denote the resolution of the image to be rendered. If *U* and *V* represent vectors parallel to horizontal and vertical edges of the image correspondingly then the center P_{ij} of the pixel in *j*-th column and *i*-th row of the image can be computed as:

$$P_{ij} = P_{UL} + \frac{j}{x_{res} - 1} U + \frac{i}{y_{res} - 1} V; \quad U = P_{UR} - P_{UL} \ , \ V = P_{LL} - P_{UL} \, . \tag{12.10}$$

The ray equation for the pixel (i,j) is:

$$P = P_O + \frac{P_{ij} - P_O}{| P_{ij} - P_O |} t \, , \tag{12.11}$$

where $| P_{ij} - P_O |$ is the length of the vector $P_{ij} - P_O$ The method of finding primary rays is illustrated in Fig. 12.9.

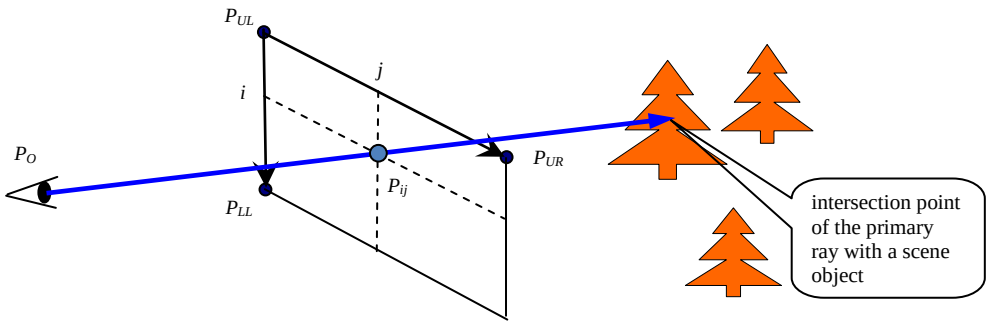


Fig.12.9. Primary rays shooting.

The complete RT rendering procedure consists now in finding primary rays for consecutive pixels and in calculating their colors using `CalculateColor()` function described in previous sections in accordance with the following pseudocode:

```
for ( i =0; i < yres; i++ )
   for ( j =0; j < xres; j++ )
   {
      find equation of the primary ray(Po,u) for the pixel i,j;
      find pixel color using CalculateColor(Po,u);
      put calculated color to the raster image pixel p[i,j];
   }
```

Finding directions of secondary rays

Specularly reflected rays

The incident ray, the surface normal vector and the reflected ray are coplanar. The direction of the reflected ray follows from the reflection principle. It states that the angles:
– between the incident ray *I* and the surface normal *N* and
– between the normal vector *N* and the reflected ray *S*
are equal. The incident ray can be decomposed into two perpendicular components; one of them is parallel to the normal vector. Then the reflected ray can be constructed by combining the component parallel to the normal vector with the reversed component perpendicular to it as shown in Fig. 12.10. The formula for the reflected ray is then:

$$S = 2N(N \bullet I) - I \, . \tag{12.12}$$

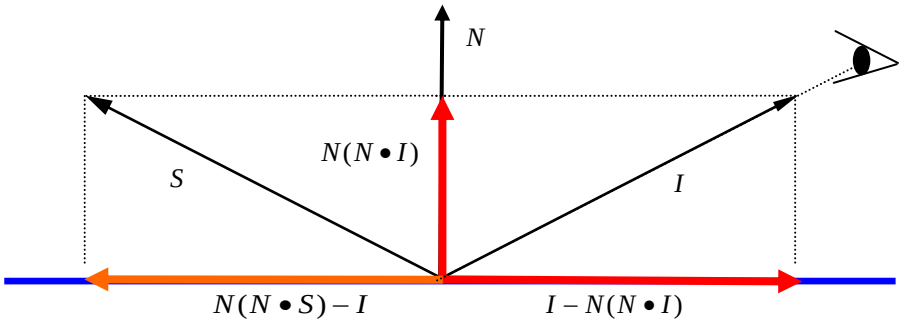


Fig.12.10. Finding direction of the reflected ray.

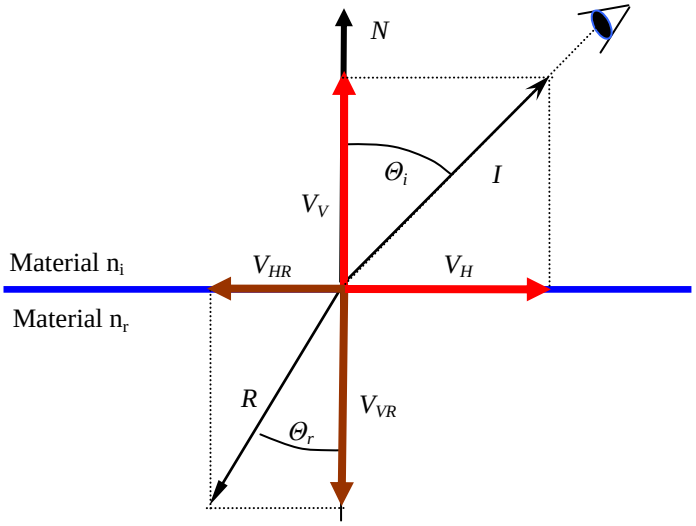


Fig.12.11. Finding direction of the refracted ray.

Refracted rays

The direction of the refracted ray can be found using similar reasoning. First, the incident ray I is decomposed into two parallel components: $V_H = I - N(N \bullet I)$ and $V_V = I - V_H$ as shown in Fig. 12.11. Then, taking into account the index of refraction:

$$\eta = \frac{\sin(\theta_r)}{\sin(\theta_i)}. \tag{12.13}$$

The component of the refracted ray V_{HR} parallel to the surface can be computed as:

$$V_{HR} = -\eta V_H = -\eta(I - N(N \bullet I)). \tag{12.14}$$

The vertical component V_{VR} is parallel to -N and its length is determined by the length of V_V and the ratio of θ_i and θ_r cosines. The following dependencies hold true:

$$\frac{|V_{VR}|}{|V_V|} = \frac{\cos(\theta_r)}{\cos(\theta_i)}$$

$$|V_{VR}| = \frac{\cos(\theta_r)}{\cos(\theta_i)}|V_V| = \frac{\sqrt{1-\sin^2(\theta_r)}}{N \bullet I}|V_V| = \frac{\sqrt{1-\eta^2(1-(N \bullet I)^2)}}{N \bullet I}|V_V| \quad (12.15)$$

$$V_{VR} = -\frac{\sqrt{1-\eta^2(1-(N \bullet I)^2)}}{N \bullet I} N(N \bullet O) = -\sqrt{1-\eta^2(1-(N \bullet I)^2)}N.$$

So finally, the refracted ray R can be composed of V_{HR} and V_{VR}:

$$R = V_{HR} + V_{VR} = -\eta(I - N(N \bullet I)) - \sqrt{1-\eta^2(1-(N \bullet I)^2)}N\,. \quad (12.16)$$

Assignment scope

1. Implement the program that renders images with RT. Use boundary representation extended with spheres defined as implicit surfaces. Extend the boundary representation format used in Assignment 8 so as to include spheres. Define the sphere by its center and radius. Use separate files with surface attribute specification and camera specification. Assume that the camera is specified by four points: P_O, P_{UL}, P_{LL}, P_{UR} and by horizontal and vertical resolution of the image. Apply Phong lighting model adapted to RT needs. Divide the implementation into the following stages:
   - scene data reading and primary rays casting - at that stage display objects hit by primary rays using flat shading with colors specified as k_{dC} coefficients,
   - shadow analysis by shadow ray tracing,
   - implementation of the complete recursive ray tracing.
2. Carry out tests using geometry models created in Assignment 9 and other scenes provided by the lecturer. Set surface attributes and light positions so as to obtain visual effects of hard shadows, light specular reflection and refraction. Make sure there are no visible artifacts on the rendered images.
3. Experiments with provided scenes. Set lighting and surface attributes so as to obtain the most realistic and pleasant look of scenes. Render series of images for constant camera settings and for changing lighting conditions.
4. Using the program implemented in Assignment 9 prepare series of shapes differing in the density of the created triangle mesh. Then render images of shape variants preserving the same camera and lighting. Observe how the rendering time depends on the number of triangles. Perform similar tests with constant camera and geometry but with changing number of point lights.
5. Integrate your ray tracer with the program implemented in Assignment 8. Add the pushbutton that invokes the raytracer for current camera settings. Elaborate the method that converts camera parameters used in Assignment 8 to the representation used in primary ray casting.

Bibliography:

1. Appel A. - Some techniques for shading machine renderings of solids, Proc. of AFIPS, Spring Joint Conf. 1968, pp. 37-45
2. Gouraud H. - Continuous shading of curved surfaces, IEEE Trans on Computers, June 1971, pp. 623-629
3. Phong B.T. - Illumination of computer generated pictures, Communications of ACM, vol. 18, no 6, 1975, pp. 311-317
4. Whitted T. - An improved illumination model for shaded display, Communications of the ACM, vol. 23, no 6, 1980, pp. 343-349
5. Fujimoto A., Tanaka T., Iwata K. - ARTS: Accelerated ray-tracing system, IEEE Trans. on Computer Graphics, vol. 6, no 4, 1986, pp. 16-26
6. Heckbert S., Moreton H. - Interpolation for polygon texture mapping and shading, State of the Art in Computer Graphics: Visualization and Modeling, Springer-Verlag, pp. 101-11, 1991
7. Cohen M., Wallace J. - Radiosity and realistic image synthesis, Academic Press Professional, 1994
8. Wilt N. - Object-oriented ray tracing in C++, Wiley & Sons Inc., 1994,
9. Foley J.D., van Dam A., Feiner S. K., Hughes J.F. - Computer Graphics, Principles and Practice, 2nd edition, Addison-Wesley, 1995
10. Worley, S. - A cellular texture basis function. In *Proc. of SIGGRAPH 1996*, ACM Press, pp. 291–294, 1996
11. Haines E., Akenine-Moller T., Hoffman N. - Real-Time Rendering (2nd Edition), A.K. Peters Ltd., 2002
12. Low K.L. - Perspective-correct interpolation, Technical report, University of North Carolina at Chapel Hill, 2002
13. Ebert D., Musgrave F., Peachey D., Perlin K., Worley S. - Texturing and Modeling. A procedural approach, Elsevier Science, 2003
14. Shirley P., Morley K. - Realistic Ray Tracing, Second Edition, AK Peters, Ltd., 2003
15. Wann Jensen H. - Monte Carlo Ray tracing, Proc. of SIGGRAPH 2003, Course 44
16. Shirley P. at al. - Fundamentals of Computer Graphics, 2nd edition, A.K. Peters, Ltd., 2005
17. McReynolds T., Blythe D. - Advanced Graphics Programming Using OpenGL, Elsevier, 2005
18. Eckel B. - Thinking in Java, Fourth Edition. Prentice Hall, 2006
19. Zhang H., Liang Y.D. - Computer Graphics Using Java 2D and 3D, Prentice Hall, 2006
20. Neider J., Davis T., Woo M. – OpenGL Programming Guide. Version 2.1. (6th Edition), Addison-Wesley, 2007
21. Klawonn F. - Introduction to Computer Graphics using Java 2D and 3D, Springer Verlag, 2008