2017-18 Onwards (MR-17)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. VIII Semester		
Code: 70623	MULTIMEDIA APPLICATION DEVELOPMENT [Professional Elective – VI]	L	T	P
Credits: 3		3	-	-

Prerequisites: Computer Graphics

Course Objectives:
This course will enable the students to learn the fundamental concepts of text, image, video and audio concepts and analyze the scripting concepts in object oriented programming frameworks and apply the compression techniques.

MODULE I: Fundamental concepts in Text and Image [09 Periods]
Multimedia and hypermedia, World Wide Web, overview of multimedia software tools. Graphics and image data representation graphics/image data types, file formats, Color in image and video: color science, color models in images, color models in video.

MODULE II: Fundamental concepts in Video and Digital audio [09 Periods]
Types of video signals, analog video, digital video, digitization of sound, MIDI, quantization and transmission of audio.

MODULE III: Action Script I and II [09 Periods]
A: Action Script I
Action Script Features, Object-Oriented Action Script, Data types and Type Checking, Classes, Authoring an Action Script Class.
B: Action Script II
Inheritance, Authoring an Action Script 2.0 Subclass, Interfaces, Packages, Exceptions.

MODULE IV: Application Development and Multimedia data compression [10 Periods]
Application Development- An OOP Application Frame work, Using Components with Action Script, Movie Clip Sub classes.
Multimedia data compression- Lossless compression algorithm: Run-Length Coding, Variable Length Coding, Dictionary Based Coding, Arithmetic Coding, Lossless Image Compression, Lossy compression algorithm: Quantization, Transform Coding, Wavelet-Based Coding, Embedded Zerotree of Wavelet Coefficients Set Partitioning in Hierarchical Trees (SPIHT).

MODULE V: Video Compression Techniques and Multimedia Networks [11 Periods]
Basic Video Compression Techniques- Introduction to video compression, video compression based on motion compensation, search for motion vectors, MPEG, Basic Audio Compression Techniques.
Multimedia Networks- Basics of Multimedia Networks, Multimedia Network Communications and Applications: Quality of Multimedia Data Transmission, Multimedia over IP, Multimedia over ATM Networks, Transport of MPEG-4, Media-on-Demand(MOD).

TEXTBOOKS:
1. Ze-Nian Li and Mark S. Drew, "**Fundamentals of Multimedia**", Pearson Education.
2. Colin Moock,"**Essentials ActionScript** 2.0", SPD O, REILLY.

REFERENCES:

1. Nigel chapman and jenny chapman,"**Digital Multimedia**", Wiley-Dreamtech
2. Steve Heath, "**Multimedia and communications Technology**", Elsevier, Focal Press.
3. Steinmetz, Nahrstedt, "**Multimedia Applications**", Springer.
4. David Hilman, "**Multimedia Technology and Applications**", Galgotia Publications

E –RESOURCES:

1. https://users.dimi.uniud.it/~antonio.dangelo/MMS/materials/Fundamentals_of_Multimedia.pdf
2. https://books.google.co.in/books?isbn=0596526946
3. https://books.google.co.in/books?id=vY25BQAAQBAJandpg=PA10anddq=acm+transactions+on+multimediaandhl=enandsa=Xandved=0ahUKEwiF5brD3tPUAhXLL48KHbvFDJYQ6AEIKzAB#v=onepageandq=acm%20transactions%20on%20multimediaandf=false
4. ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=93
5. nptel.ac.in/courses/117105083/

Course Outcomes:

At the end of the course, students will be able to

1. **Develop** Fundamental concepts in Text and Image
2. **Analyze** the Fundamental concepts in video and digital audio
3. **Apply** the scripting concepts in multimedia environments.
4. **Understand** the Application Development and Multimedia data compression
5. **Outline** Video Compression Techniques and Multimedia Networks

UNIT I
Introduction to Multimedia

What is Multimedia?

- When different people mention the term multimedia, they often have quite different, or even opposing, viewpoints.
- A PC vendor: a PC that has sound capability, a DVD-ROM drive, and perhaps the superiority of multimedia-enabled microprocessors that understand additional multimedia instructions.
- A consumer entertainment vendor: interactive cable TV with hundreds of digital channels available, or a cable TV-like service delivered over a high-speed connection high Internet connection.
- A Computer Science (CS) student: applications that use multiple modalities, including text, images, drawings (graphics), animation, video sound including speech and interactivity video, speech, interactivity.

- **Multimedia and Computer Science:** Graphics, HCI, visualization, computer vision, data compression, graph theory, networking, database systems.

Components of Multimedia

- Multimedia involves multiple modalities of text, audio, images, drawings, animation, and video.
- Examples of how these modalities are put to use:
- Video teleconferencing.
- Distributed lectures for higher education.
- Tele-medicine.
- Co-operative work environments.
- Searching in (very) large video and image databases for target visual objects.
- "Augmented"reality: placing real-appearing computer video scenes graphics and objects into scenes.

- Including audio cues for where video-conference participants are located.
- Building searchable features into new video, and enabling very high-to very low-bit-rate new use of new, scalable multimedia products.
- Making components editable multimedia editable.
- Building "inverse-Hollywood" applications that can recreate the process by which a video was made.
- **Video understanding** has also been called an inverse Hollywood problem.
- Using voice-recognition to build an interactive environment, say a kitchen-wall web browser.

Multimedia Research Topics & Projects

- To the computer science researcher, multimedia consists of a wide variety of topics:
- **Multimedia processing and coding**: multimedia content analysis, content-based multimedia retrieval, multimedia security, audio/image/video processing, compression, etc.
- **Multimedia system support and networking:** network protocols, Internet, operating systems, servers and clients, quality of service (QoS), and databases.
- **Multimedia tools end-systems applications:** Hypermedia systems, user interfaces, authoring systems.
- Multi-modal interaction and integration: "ubiquity"—web-everywhere devices, multimedia education including Computer Supported Collaborative Learning, and design and applications of virtual environments.

Current Multimedia Projects

- Many exciting research projects are currently underway. Here area few of them:

1. **Camera-based object tracking technology:** tracking of the control objects provides user control of the process.
2. **3D motion capture:** used for multiple actor capture so that multiple actors in a virtual studio can be used real to automatically produce realistic animated models with natural movement.
3. **Multiple views:** allowing photo-realistic (video-quality) synthesis of virtual actors from several cameras or from a single camera under differing lighting.
4. **3D capture technology:** allow synthesis of highly realistic speech facial animation from speech.

5. **Specific multimedia applications:** aimed at handicapped persons with low vision capability and the elderly —a rich field of endeavor.
6. **Digital fashion:** aims to develop smart clothing that can communicate with other such enhanced clothing using wireless communication, so as to artificially enhance human interaction in a social setting.
7 **Electronic Housecall system:** an initiative for providing interactive health monitoring services to patients in their homes
8. **Augmented Interaction applications:** used to develop interfaces between real and virtual humans for tasks such as augmented storytelling .

Multimedia and Hypermedia

- **History of Multimedia:**

1. Newspaper: perhaps the first mass communication medium, uses text, graphics, and images.
2. Motion pictures: conceived of in 1830's in order to observe motion too rapid for perception by the human eye.
3. Wireless radio transmission: Guglielmo Marconi, at Pontecchio, Italy, in 1895.
4. Television: the new medium for the 20th century, established video as a commonly available medium and has since changed the world of mass communications .

5. The connection between computers and ideas about multimedia covers what is actually only a short period:

1945 – Vannevar Bush wrote a landmark article describing what amounts to a hypermedia system called Memex.

1960 – Ted Nelson coined the term hypertext.

1967 – Nicholas Negroponte formed the Architecture Machine Group.

1968 – Douglas Engelbart demonstrated the On-Line System (NLS), another very early hypertext program.

1969 – Nelson hypertext and van Dam at Brown University created an early editor called FRESS.

1976 – The MIT Architecture Machine Group proposed a project entitled Multiple Media — resulted in the Aspen Movie Map, hypermedia videodisk, in 1978.

1985 – Negroponte and Wiesner co-founded the MIT Media Lab.

1989 – Tim Berners-Lee proposed the World Wide Web

1990 – Kristina Hooper Woolsey headed the Apple Multimedia Lab.

1991 MPEG 1– MPEG [M(oving) P(ictures) E(xperts) G(roup)]was approved as an international standard for digital video — led to the newer standards, MPEG-2, MPEG-4, and further MPEGs in the 1990s.

1991 – The introduction of PDAs in 1991 began a new period in the use of computers in multimedia.

1992 – JPEG was accepted as the international standard for digital image compression — led to the new JPEG2000 standard.

1992 – The first MBone audio multicast on the Net was made.

1993 The of Illinois National Center for Supercomputing– University Applications produced NCSA Mosaic — the first full-fledged browser.

- 1994 – Jim Clark and Marc Andreessen created the Netscape program .
- 1995 – The JAVA language was created for platform independent application development.
- 1996 – DVD video was introduced; high quality full-length movies were distributed on a single disk.
- 1998 – XML 1.0 was announced as a W3C Recommendation.
- 1998 – Hand-held MP3 devices first made inroads into consumerist tastes in the fall of 1998, with the introduction holding of flash of devices 32MB memory.
- 2000 – WWW size was estimated at over 1 billion pages.

Hypermedia and Multimedia

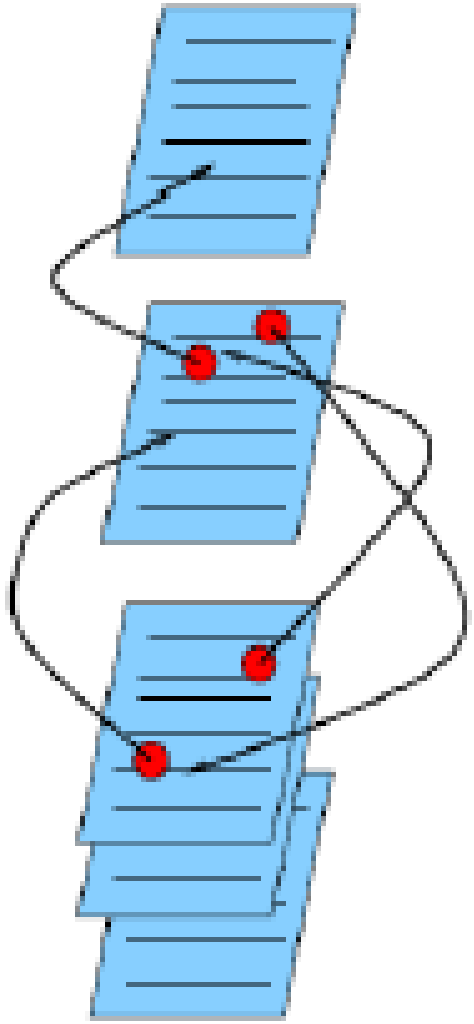
- A hypertext system: meant to be read nonlinearly, by following links that point to other parts of the document, or to other documents.
- HyperMedia: not constrained to be text-based, can include other media, e.g., graphics, images, and especially the continuous media sound and video.
  - The World Wide Web (WWW) — the best example of a hypermedia application.
- Multimedia means that computer information can be represented through audio graphics images audio, graphics, images,
- video, and animation in addition to traditional media.

Normal Text

Linear

"Hot spots"

Hypertext



Nonlinear

Examples of typical present multimedia applications include:

- Digital video editing and production systems.
- Electronic newspapers/magazines.
- World Wide Web.
- On-line reference works: e.g. encyclopedias, games.
- Home shopping.
- Interactive TV.
- Multimedia courseware.
- Video conferencing.
- Video-on-demand.
- Interactive movies.

World Wide Web

- The W3C has listed the following goals for the WWW

1. Universal access of web resources (by everyone everywhere).
2. Effectiveness of navigating available information.
3. Responsible use of posted material.

HTTP (HyperText Transfer Protocol)

- HTTP: a protocol that was originally designed for transmitting hypermedia but can also support transmission of any file type.
- HTTP a stateless request/response protocol: is no information carried over for the next request.
- The basic request format:

  Method URI Version

  Additional-Headers:

  Message-body
- The URI (Uniform Resource Identifier): an identifier for the resource accessed, e.g. the host name, always preceded by the token http://.

- Two popular methods: GET and POST.
- The basic response format:
  - Version Status-Code Status-Phrase
  - Additional-Headers
- Additional – Message-body
- Two commonly seen status codes:
- 200 OK — the request was processed successfully.
- 404 Not Found — the URI does not exist.

HTML (HyperText Markup Language)

- HTML: a language for publishing Hypermedia on the World Wide Web — defined using SGML:

1. HTML uses ASCII, it is portable to all different (possibly binary incompatible) computer hardware.
2. The current version of HTML is version 4.01.
3. The next generation of HTML is XHTML —a reformulation of HTML using XML.

- HTML uses tags to describe document elements:

<token params> —defining a starting point,

</token> — the ending point of the element.

Some elements have no ending tags.

- A very simple HTML page is as follows:

```
<HTML> <HEAD>
 <TITLE>
 A sample web page.
 </TITLE>
 <META NAME = "Author" CONTENT = "Cranky Professor">
</HEAD> <BODY>
 <P>
 We can put any text we like here, since this is
 a paragraph element.
 </P>
</BODY> </HTML>
```

- Naturally, HTML has more complex structures and can be mixed in with other standards.

XML (Extensible Markup Language)

- XML: a markup language for the WWW in which there is modularity of data, structure and view so that user or application can be able to define the tags (structure).
- Example of using XML to retrieve stock information from a database a query according to user

1. First use a global Document Type Definition (DTD) that is already defined.
2. The server side script will abide by the DTD rules to generate document an XML according to the query using data from your database.
3. Finally send user the XML Style Sheet (XSL) depending on the type of device used to display the information.

- The current XML version is XML 1.0, approved by the W3C in Feb. 1998.
- XML syntax looks like HTML syntax, although it is much more strict:
  - All tags are in lower case and a tag has only inline data has to terminate itself, i.e., <token params />.
  - Uses name spaces so that multiple DTDs declaring different elements but with similar tag names can have their elements distinguished.
  - DTDs can be imported from URIs as well.

An example of an XML document structure — the definition for a small XHTML document:

```
<?xml version="1.0" encoding="iso-8859-1"?> <!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
                              transition.dtd">
 <html xmlns="http://www.w3.org/1999/xhtml">
 ... [html that follows
     the above mentioned
     XML rules]
 </html>
```

- The following XML related specifications are also standardized:

  – XML Protocol: used to exchange XML information between processes .

  – XML Schema: a more structured and powerful language defining XML data types (tags).

  – XSL: basically CSS for XML.

  – SMIL : synchronized Integration Multimedia Language, pronounced "smile"— a particular application of XML (globally predefined DTD) that allows for specification of interaction among any media types and user input, in a temporally scripted manner.

SMIL (Synchronized Multimedia Integration Language)

- Purpose of SMIL: it is also desirable to be able to publish presentations using multimedia a markup language.
- A multimedia language needs to scheduling and synchronization of different multimedia elements, and define their interactivity with the user.
- The W3C established a Working Group in 1997 to come up with specifications for a multimedia synchronization language
- SMIL 2 0 was accepted August 2001 2.0 in 2001.

- SMIL 2.0 is specified in XML using a modularization approach similar to the one used in xhtml:

1. All SMIL elements are divided into modules — sets of elements, attributes and values that define one conceptual functionality.
2. In the interest of modularization, not all available modules need to be included for all applications.
3. Language Profiles: specifies a particular grouping of modules, and particular modules may have integration profile must requirements that a follow.

```
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 2.0"
"http://www.w3.org/2001/SMIL20/SMIL20.dtd">
<smil xlmns=
"http://www.w3.org/2001/SMIL20/Language">
<head>
    <meta name="Author" content="Some Professor" />
</head>
<body>
    <par id="MakingOfABook">
        <seq>
        <video src="authorview.mpg" />
        <img src="onagoodday.jpg" />
        </seq>

        <audio src="authorview.wav" />
        <text src="http://www.cs.sfu.ca/mmbook/" />
    </par>
</body>
</smil>
```

Overview of Multimedia Software Tools
1. Music Sequencing and Notation

- **Cakewalk**: now called Pro Audio.

  The term sequencer comes from older devices that stored sequences of notes ("events", in MIDI).

  It is also possible to insert WAV files and Windows MCI commands (for animation and video) into music tracks (MCI is a ubiquitous component of the Windows API.)
- **Cubase:** another sequencing/editing program, with capabilities similar to those of Cakewalk. It includes some tools digital audio editing tools.
- **Macromedia Soundedit**: mature program for creating audio for multimedia projects and the web that integrates well with other Macromedia products such as Flash and Director.

2. Digital Audio

- **Cool Edit**: a very powerful and popular digital audio toolkit; emulates a professional audio studio —multitrack productions and sound file editing including digital signal processing effects.
- **Sound Forge**: a sophisticated PC-based program for WAV audio files.
- **Pro Tools**: a high-end integrated audio production and editing environment MIDI creation and manipulation; powerful audio mixing, recording,and editing software.

3.Graphics and Image Editing

- **Adobe Illustrator**: a powerful publishing tool from Adobe. Uses vector graphics; can be exported to Web graphics Web.
- **Adobe Photoshop:** the standard in a graphics, image processing and manipulation tool.
  - Allows layers of images, graphics, and text that can be separately manipulated for maximum flexibility.
  - Filter factory permits creation of sophisticated lighting-effects filters.
- **Macromedia Fireworks**: software for making graphics specifically for the web.
- **Macromedia Freehand:** a text and web graphics editing tool that supports many bitmap formats such as GIF, PNG, and JPEG.

4.Video Editing

- **Adobe Premiere:** an intuitive, simple video editing tool editing i e clips for nonlinear editing, i.e., putting video into any order:

  Video and audio are arranged in "tracks" tracks .

  Provides a large number of video and audio tracks, superimpositions and virtual clips. => effective multimedia productions with little effort.

- **Adobe After Effects**: a powerful video editing tool that enables users to add and change existing movies. Can add many effects: lighting, shadows, motion blurring; layers.
- **Final Cut Pro**: a video editing tool by Apple; Macintosh only.

5. Animation

Multimedia APIs:

- Java3D: API used by Java to construct and render 3D graphics, similar to the way in which the Java Media Framework is used for handling media files.

1 Provides a basic set of object primitives (cube, splines, etc.) for building scenes.

2. It is an abstraction layer built on top of OpenGL or DirectX (the user can select which).

- DirectX : Windows API that supports video, images, audio and 3-D animation
- OpenGL: the highly portable most popular 3 D API

Rendering Tools:

- **3D Studio Max**: rendering tool that includes a number of very high-end professional tools for character animation, game development, and visual effects production.
- **Softimage XSI:** a powerful modeling, animation, and rendering package used for animation and special effects in films and games.
- **Maya**: competing product to Softimage; as well, it is a complete modeling package.
- **RenderMan:** rendering package created by Pixar.
- **GIF Animation Packages:** a simpler approach to animation, allows very quick development of effective small animations for the web.

6.Multimedia Authoring

- **Macromedia Flash**: allows users to create interactive movies using the score metaphor i.e. a timeline arranged in parallel event sequences.
- **Macromedia Director**: uses a movie metaphor to create interactive presentations — very powerful and includes a built-in scripting language, Lingo, that allows creation of complex interactive movies.
- **Authorware:** a mature, well-supported authoring product based on the Iconic/Flow-control metaphor.
- **Quest**: similar to Authorware in many ways, uses a type of flowcharting metaphor. However, flowchart nodes can encapsulate information in a more abstract way (called frames) than simply subroutine levels.

Graphics and Image Data Representations

Graphics/Image Data Types

- The number of file formats used in multimedia continues to rapidly.

File Import					File Export		Native
Image	Palette	Sound	Video	Anim.	Image	Video	
.BMP, .DIB, .GIF, .JPG, .PICT, .PNG, .PNT, .PSD, .TGA, .TIFF, .WMF	.PAL .ACT	.AIFF .AU .MP3 .WAV	.AVI .MOV	.DIR .FLA .FLC .FLI .GIF .PPT	.BMP	.AVI .MOV	.DIR .DXR .EXE

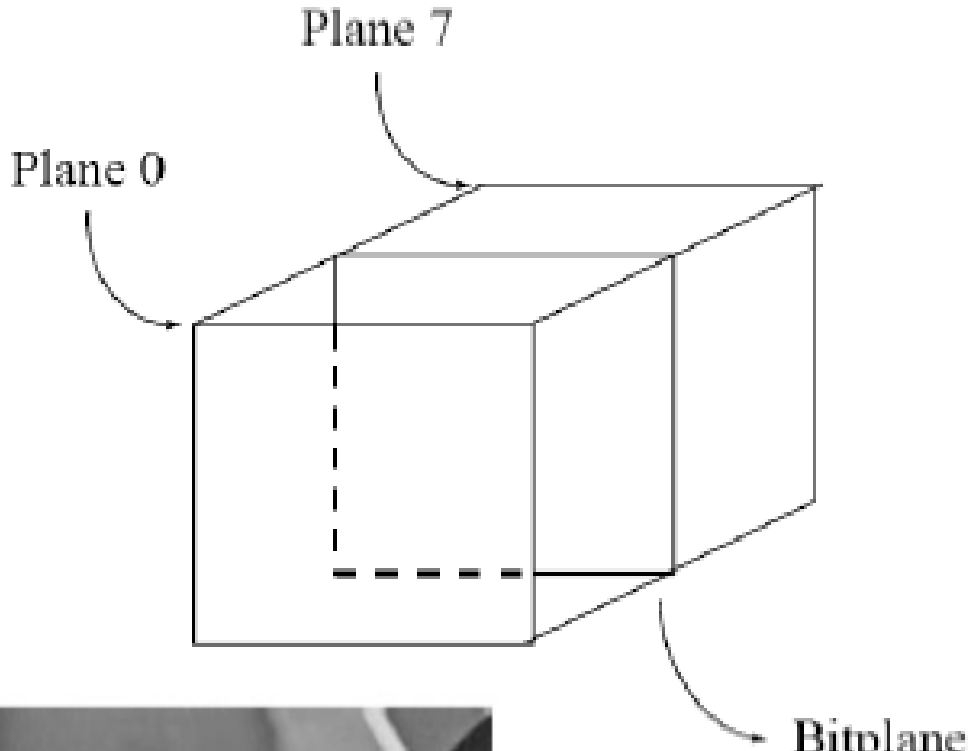
1-bit Images

- Pixels, or pels: picture elements
- Each pixel is stored as a single bit (0 or 1), so also referred to as binary image.
- Such an image is also called a 1-bit monochrome image since it contains no color.

8-bit Gray-level Images

- Each pixel has a gray-value between 0 and 255.
- Each pixel is represented by a single byte; e.g., a dark pixel might have a value of 10 and bright one might be 230.
- Bitmap: The two-dimensional array of pixel values that represents the graphics/image data.
- Image resolution refers to the number of pixels in a digital image (higher resolution always yields better quality).
- Fairly high resolution for such an image might be 1, 600 × 1, 200, whereas lower resolution might be 640 × 480.

- Frame buffer: Hardware used to store bitmap.
  - Video card (actually a graphics card) is used for this purpose.
- The resolution of the video card does not have to match the desired resolution of the image, but if not enough video card memory is available then the data has to be shifted around in RAM for display.
- 8-bit image can be thought of as a set of 1-bit bitplanes, where each plane consists of a 1-bit representation of the image at higher and higher levels of "elevation": a bit is turned on if the image pixel has a nonzero value that is at or above that bit level.



- Each pixel is usually stored as a byte (a value between 0 to 255), so a 640 × 480 grayscale image requires 300 kB of storage (640 × 480 = 307, 200).
- If we want to print such image, things become more complex.
- When an image is printed, the basic strategy of dithering is used, which trades intensity resolution for spatial resolution to provide ability to print multi-level images on 2-level (1-bit) printers.

Dithering

- Full-color photographs may contain an almost infinite range of color values. Dithering is the most common means of reducing the color range of images down to the 256 (or fewer) colors seen in 8-bit GIF images.
- For printing, Dithering is used to calculate larger patterns of dots such that values from 0 to 255 correspond to pleasing patterns that correctly represent darker and brighter pixel values.

- The main strategy is to replace a pixel value by a larger pattern say 2 × 2 or 4 × 4 such that the number of printed dots approximates the varying sized disks of ink used in analog, in halftone sized printing (e.g., for newspaper photos).
- Half-tone printing is an analog process that uses smaller or larger filled circles of black ink to represent shading, for newspaper printing.

- An algorithm for ordered dither, with $n \times n$ dither matrix, is as follows:

```
BEGIN
    for x = 0 to x_max                    // columns
        for y = 0 to y_max                // rows
            i = x mod n
            j = y mod n
            // I(x, y) is the input, O(x, y) is the output,
            //D is the dither matrix.
            if I(x, y) > D(i, j)
                O(x, y) = 1;
            else
                O(x, y) = 0;
END
```

Fig. (a) shows a grayscale image of "Lena". The ordered-dither version is shown as Fig. (b), with a detail of Lena's right eye in Fig. (c).

Image Data Types

- The most common data types for graphics and image file formats — 24-bit color and 8-bit color.
- Most image formats incorporate some variation of a compression technique due to the large storage size of image files.
- Compression techniques can be classified into either lossless or lossy.

24-bit Color Images

- In a color 24-bit image, each pixel is represented by three bytes, usually representing RGB.
- This format supports 256 × 256 × 256 possible combined colors, or a total of 16,777,216 possible colors.
- However such flexibility does result in a storage penalty: A 640 × 480 24-bit color image would require 921.6 kB of storage without any compression.
- An important point: many 24-bit color images are actually stored as 32-bit images, with the extra byte of data for each pixel used to store an alpha value representing special effect information (e.g., transparency).

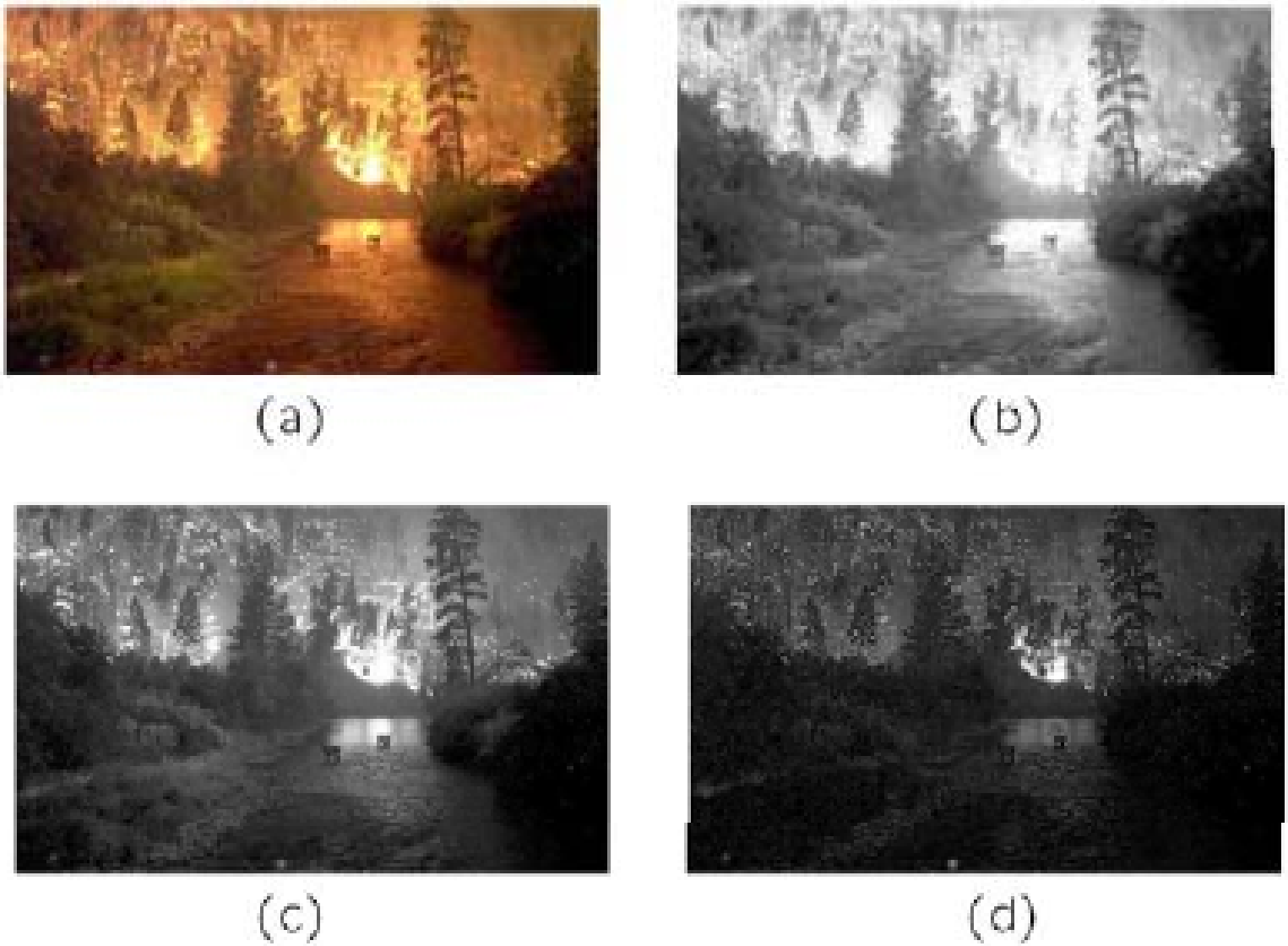


Fig. 3.5 High-resolution color and separate R, G, B color channel images. (a): Example of 24-bit color image "forestfire.bmp". (b, c, d): R, G, and B color channels for this image

8-bit Color Images

- Many systems can make use of 8 bits of color information (the so-called "256 colors") in producing a screen image.
- Such image files use the concept of a lookup table to store color information.
- Basically, the image stores not color, but instead just a set of bytes, each of which is actually an index into a table with 3-byte values that specify the color for a pixel with that lookup table index.

- An image histogram is a type of histogram which acts as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value.

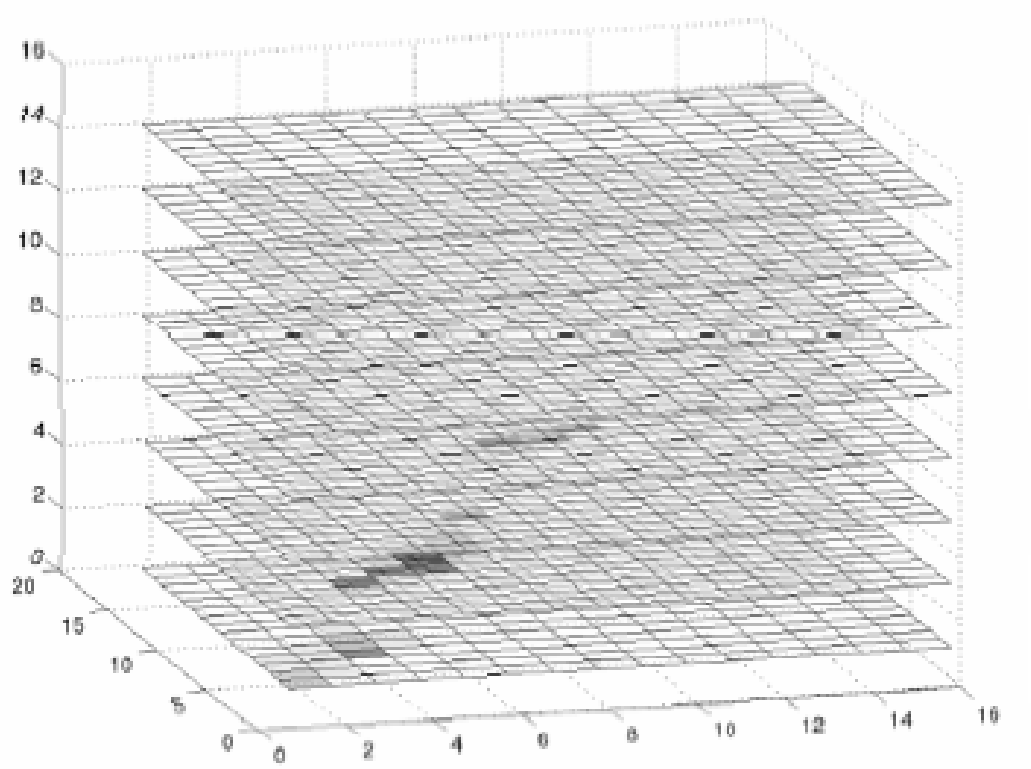
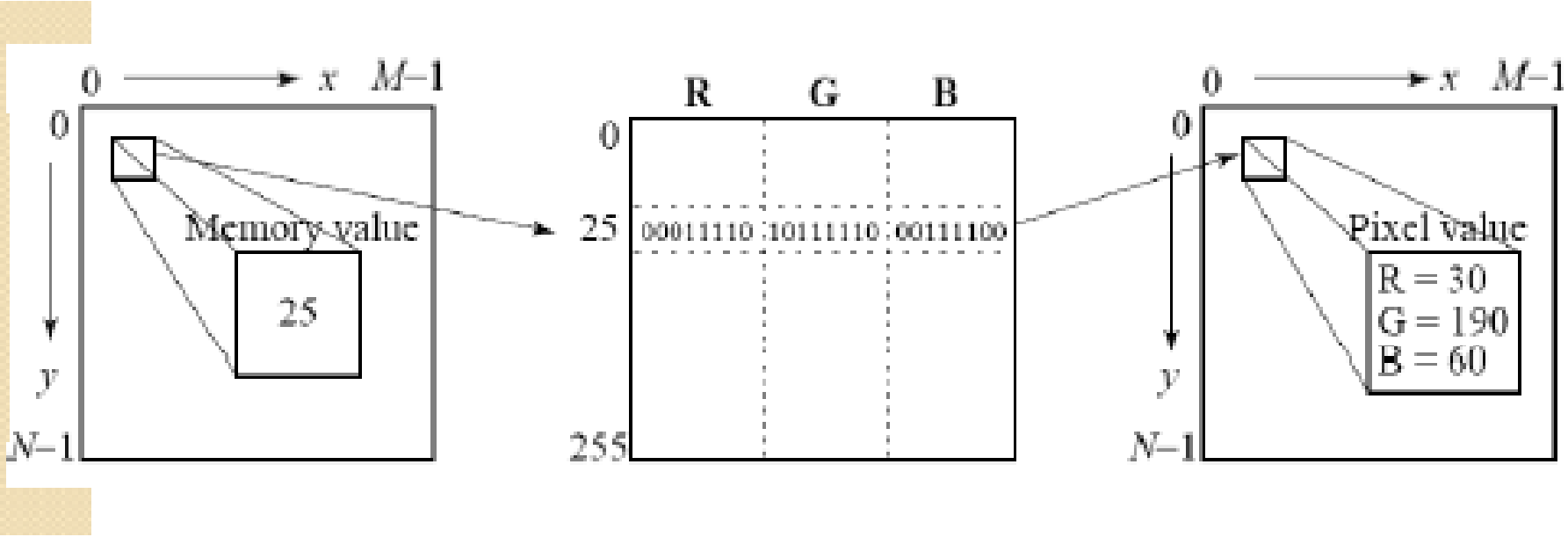


Fig. 3.6: 3-dimensional histogram of RGB colors in "forestre.bmp".

Fig. 3.7 shows the resulting 8-bit image, in GIF format.

Note the great savings in space for 8-bit images, over 24-bit ones: a 640 × 480 8-bit color image only requires 300 kB of storage, compared to 921.6 kB for a color image (again, without any compression applied).



Color Look Look-up Table (LUT)

Also called as Palette

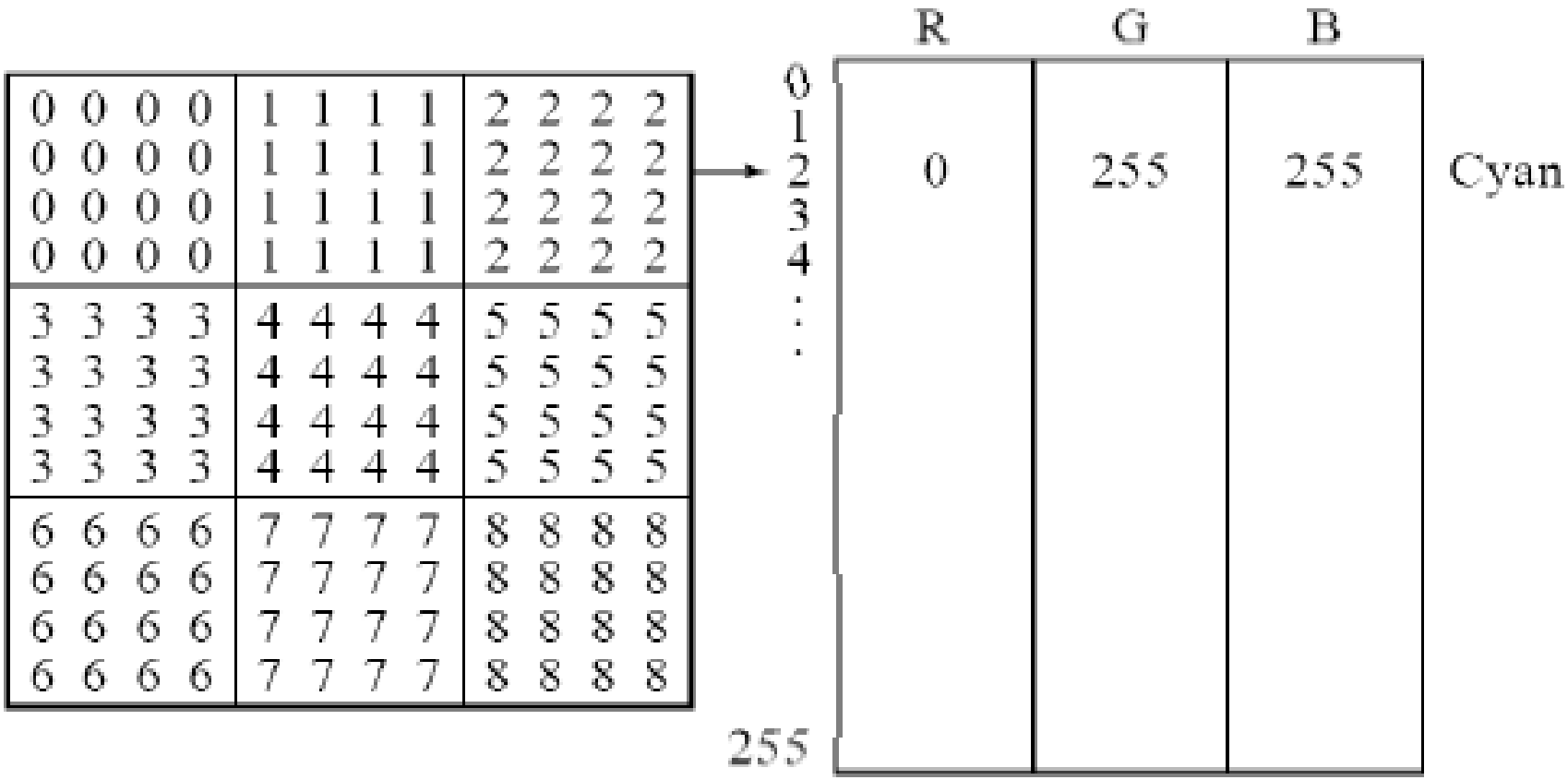


Fig. 3.9: Color-picker for 8-bit color: each block of the color-picker corresponds to one row of the color LUT

Fig. 3.10: (a): 24-bit color image "lena.bmp". (b):Version with color dithering. (c): Detail of dithered version.

How to Devise a Color Lookup Table

- The most straightforward way to make 8-bit look-up color out of 24-bit color would be to divide the RGB cube into equal slices in each dimension.
- The centers of each of the resulting cubes would serve as the entries in the color LUT, while simply scaling the RGB ranges 0..255 into the appropriate ranges would generate the 8-bit codes.
- Since humans are more sensitive to R and G than to B, we could shrink the R range and G range 0..255 into the 3 bit range 0..7 and shrink the B range down to the 2-bit range 0..3, thus making up a total of 8 bits.
- To shrink R and G, we could simply divide the R or G value by (256/8)=32 and then truncate. Then each pixel in the image gets replaced by its 8-bit index and the color LUT serves to generate 24-bit color.

Median -cut algorithm for Color Reduction Problem

- A simple alternate solution that does a better job for this color reduction problem.

a) The idea is to sort the R byte values and find their median; then values smaller than the median are labeled with a "0" bit and values larger than the median are labeled with a "1" bit.

b) This type of scheme will indeed concentrate bits where they most need to differentiate between high populations of close colors.

c) One can most easily visualize finding the median by using a histogram showing counts at position 0..255.

d) Fig. shows a histogram of the R byte values for the forestfire.bmp image along with the median of these values, shown as a vertical line.

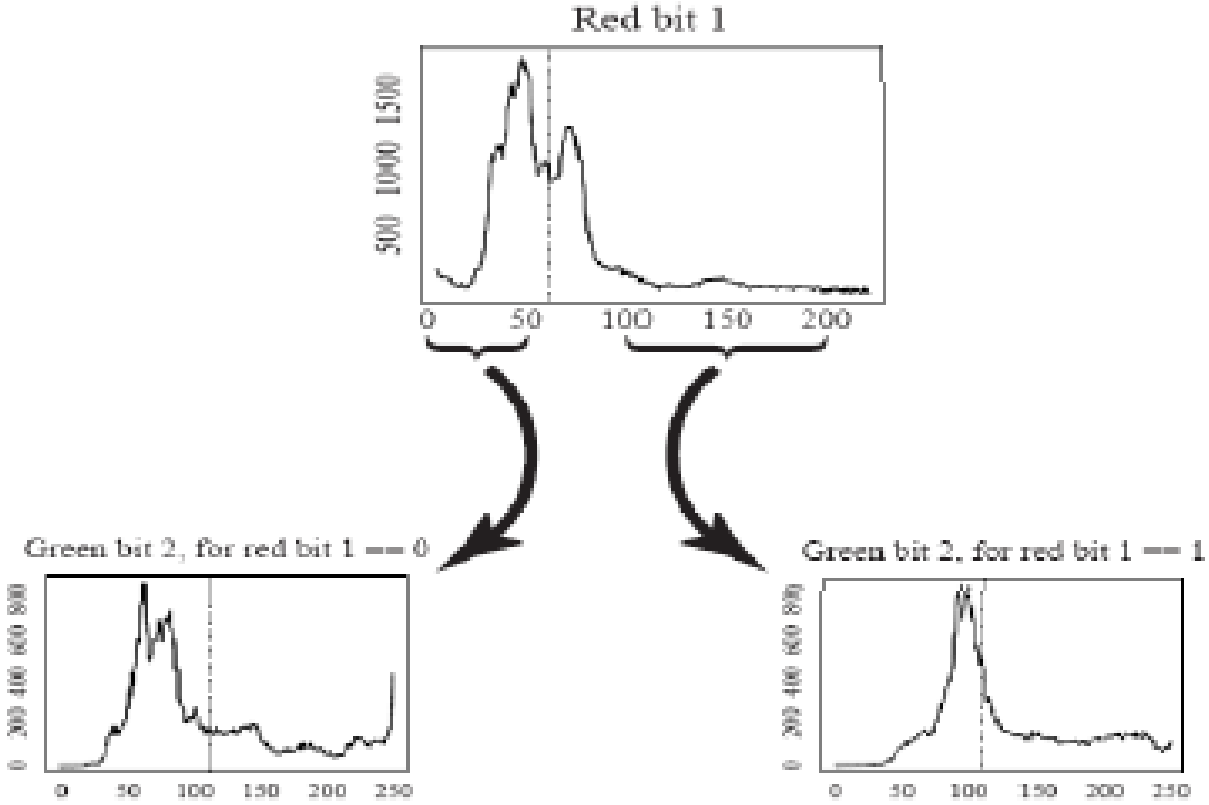


Fig. 3.11 Histogram of R bytes for the 24-bit color image "forestfire.bmp" results in a "0" bit or "1" bit label for every pixel. For the second bit of the color table index being built, we take R values less than the R median and label just those pixels as "0" or "1" according as their G value is less than or greater than the median of the G value, just for the "0" Red bit pixels. Continuing over R, G, B for 8 bits gives a color LUT 8-bit index

Median-Cut Algorithm

1. **Find the smallest box that contains all the colors in the image.**
2. **Sort the enclosed colors along the longest dimension of the box.**
3. **Split the box into two regions at the median of the sorted list.**
4. **Repeat that the above process in steps (2) and (3) until the original color space has been divided into, say, 256 regions.**

5 For every box call the mean of R G and B in that box the representative (the center) color for the box.

6. **Based on the Euclidean distance between a pixel RGB value and the box centers, assign every pixel to one of the representative colors. Replace the pixel by the code in a lookup table that indexes representative colors.**

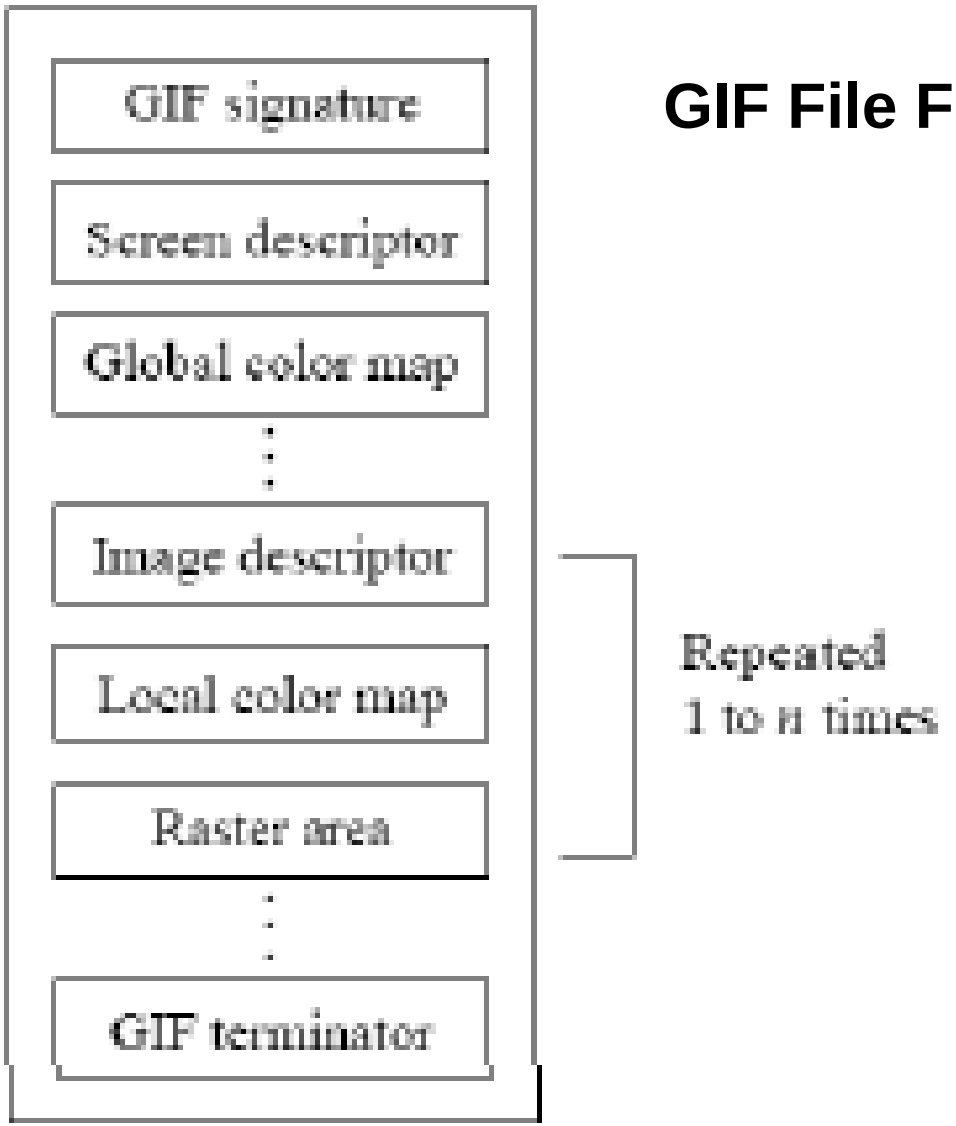
Popular File Formats 3

- GIF (Graphics Interchange Format)
- JPEG (Joint Photographic Experts Group)
- PNG (Portable Network Graphics)
- TIFF(Tagged Image File Format)
- EXIF (Exchange Image File)

GIF (Graphics Interchange Format)

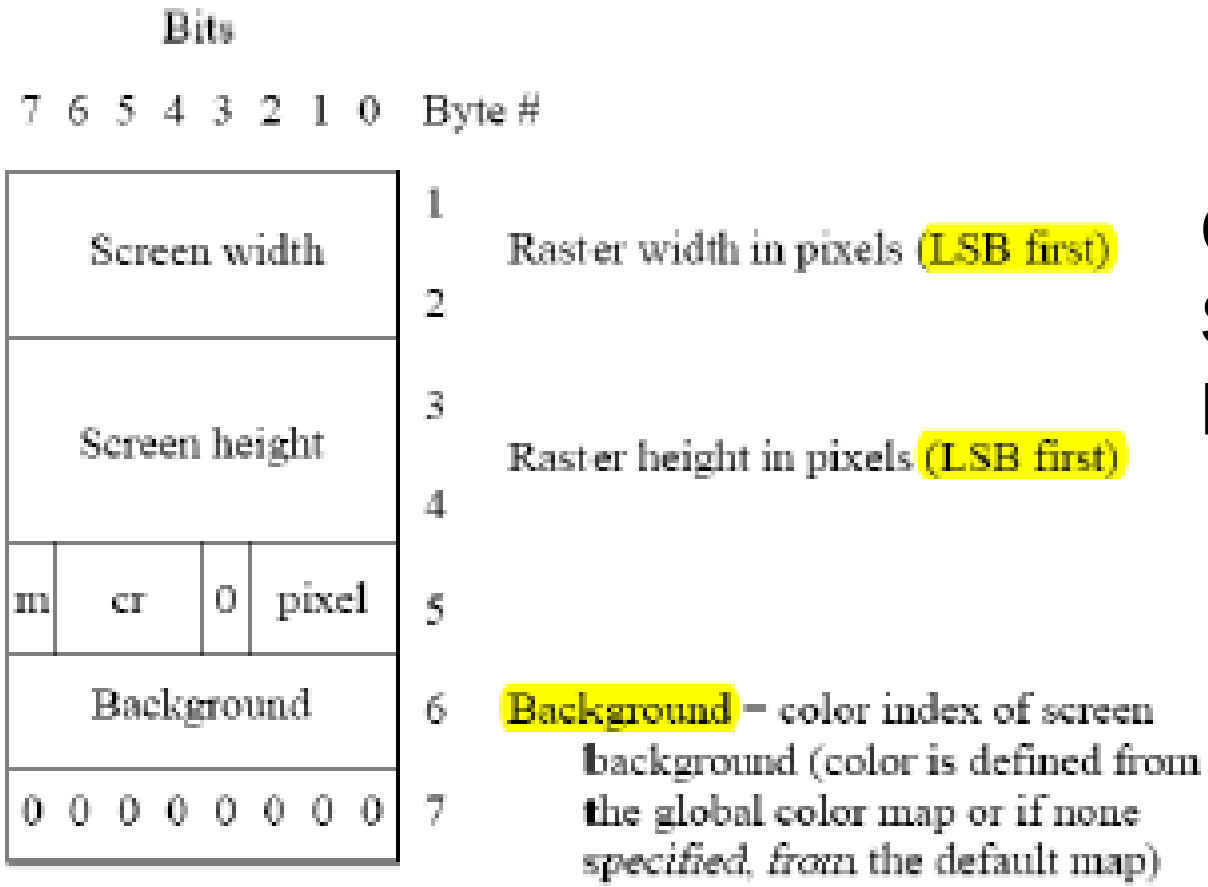
- GIF standard uses Lempel-Ziv-Welch algorithm.
- Limited to 8-bit (256) color images only, which, while producing acceptable color images, is best suited for images with few distinctive colors (e.g., graphics or Drawing).
- GIF standard supports interlacing — successive display of pixels in widely-spaced rows by a 4-pass display process.
- GIF images are of two types
  - GIF87a: The original specification .
  - GIF89a: The later version. Supports simple animation via a Graphics Control Extension block in the data, provides simple control over delay time a transparency index etc.

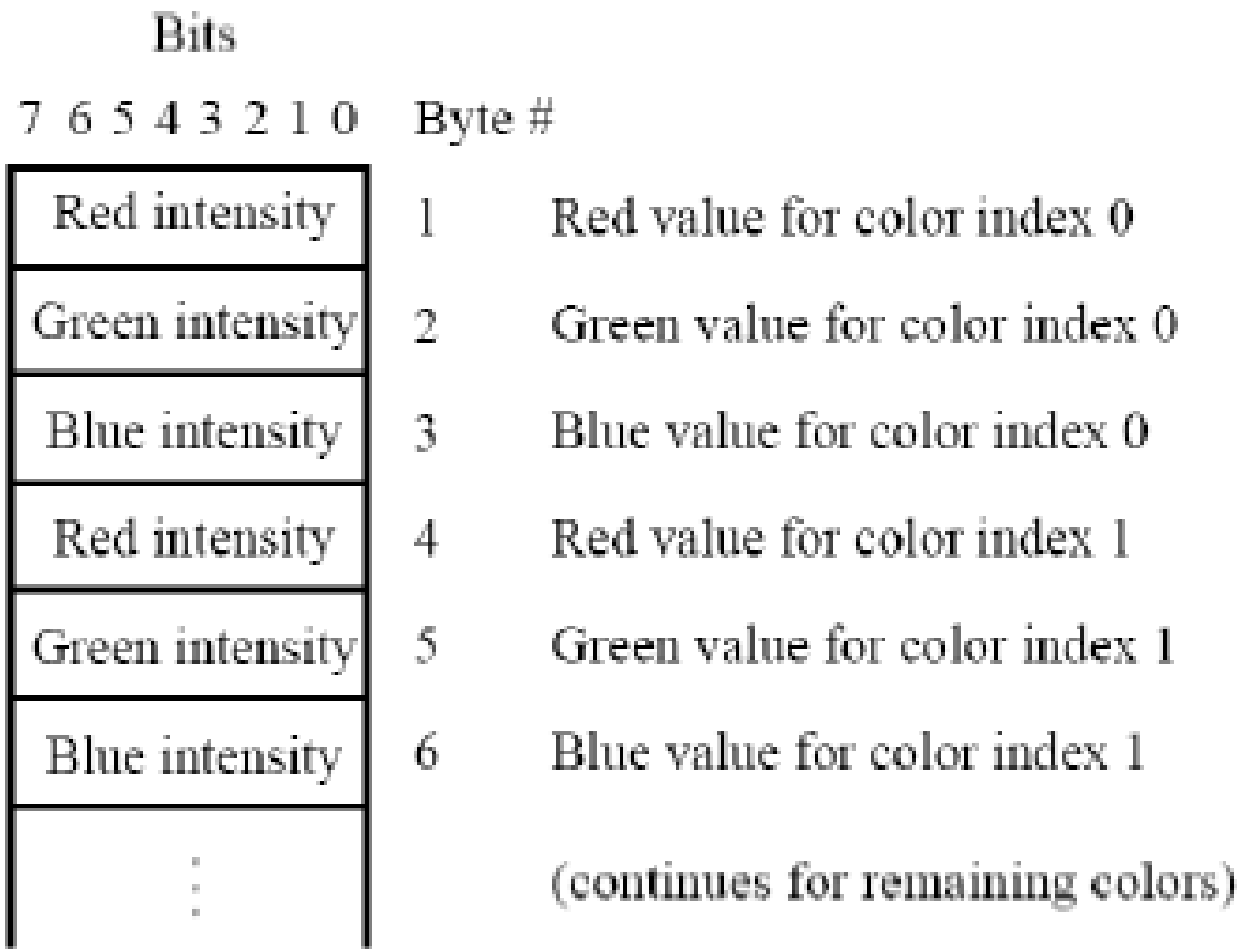
GIF87



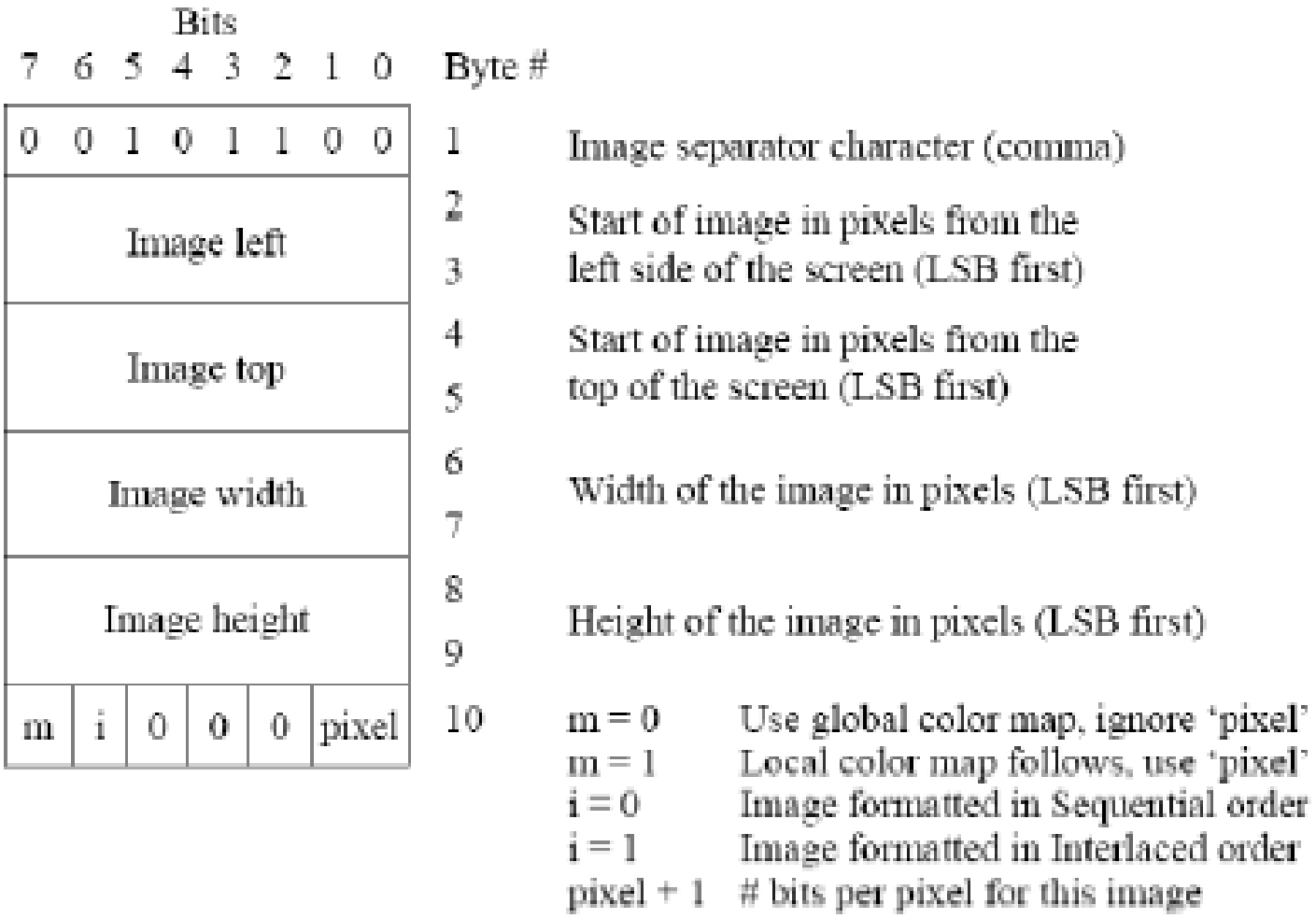
GIF File Format

GIF Screen Descriptor





GIF Color Map



GIF Image Descriptor

JPEG (Joint Photographic Expert Group)

- The most important current standard for image compression.
- The human vision system has some specific limitations and JPEG takes advantage of these to achieve high rates of compression.
- The eye-brain system cannot see extremely fine detail.
- JPEG allows the user to set a desired level of quality, or compression ratio (input divided by output).

JPEG image with low quality specified by user.

- As an example, Fig. shows forestfire image, with a quality factor Q=10%.
- This image is a mere 1.5% of the original size. In comparison, a JPEG image with Q=75% yields an image size 5.6% of the original, whereas a GIF version of this image compresses down to 23.0% of uncompressed image size.

PNG (Portable Network Graphics)

- PNG meant to supersede the GIF standard, and extends it in important ways.
- Special features of PNG files include support for 48 bits of color information.
- Files may contain gamma-correction information for correct display images alpha channel of color images, as well as alpha-information for such uses as control of transparency.
- The display progressively displays pixels in a 2-dimensional fashion by showing a few pixels at a time over seven passes through each 8 × 8 block of an image.

TIFF(Tagged Image File Format)

- TIFF is another popular image file format, developed by the Aldus Corporation in the 1980's and was later supported by Microsoft .
- Its support for attachment of additional information (referred to as "tags") provides a great deal of flexibility.
- The most important tag is a format signifier: what type of compression etc is in use in the stored image etc. image.
- TIFF can store many different types of image: 1-bit, grayscale , 8-bit color, 24-bit RGB, etc.
- TIFF was originally a lossless format but now a new JPEG tag allows one to opt for JPEG compression.

EXIF (Exchange Image File)

- EXIF is an image format for digital cameras:

1. Compressed EXIF files use the baseline JPEG format.
2. A variety of tags (many more than in TIFF) are available to facilitate higher quality printing, since information about the camera and picture-taking picture conditions (flash, exposure, light source, white balance, type of scene, etc.) can be stored and used by printers for possible color correction algorithms.
3. The EXIF standard also includes specification of file format for audio that accompanies digital images. As well, it also supports tags for information needed for conversion to FlashPix (initially developed by Kodak).

Graphics Animation Files

- A few format are aimed at storing graphics animation(series of drawings/graphics illustrations) as opposed to video(series of images).
- FLC is an animation or moving picture file format; it was originally created by Animation Pro. Another format, FLI, is similar to FLC.
- GL produces somewhat better quality moving pictures. GL animations can also usually handle larger file sizes sizes.
- Many older formats: such as DL or Amiga IFF files, Apple Quicktime files, as well as animated GIF89 files.

PS (PostScript) & PDF (Portable Document Format)

- Postscript is an important language for typesetting, and many high-end printers have a Postscript interpreter built into them.
- PS is a vector-based picture language, rather than pixel-based: page element definitions are essentially in terms of vectors.
- PS includes text as well as vector/structured graphics, bit-mapped images can be included in output files.
- Encapsulated PS files add some additional information for inclusion of Postscript files in another document.

- Postscript page description language itself does not provide compression; in fact, Postscript files are just stored as ASCII.
- Another text + figures language has begun to supersede or at least parallel Postscript: Adobe Systems Inc. includes LZW compression in its Portable Document Format (PDF) file format.
- PDF files that do not include images have about the same compression ratio, 2:1 or 3:1, as do files compressed with other LZW-based compression tools.
- For files containing images PDF may achieve higher compression ratio by using JPEG compression for the image content.

Other Formats

- **Windows WMF :-** Windows MetaFile (WMF) is the native vector file format fir MS Windows operating environment.
- **Windows BMP** :- BitMap (BMP) is the major system standard graphics file format for MS Windows used in Paint & other programs.
- **Macintosh PAINT & PICT :-** PAINT was originally used in the MacPaint program, initially only for 1-bit monochrome images. PICT format is used in MacDraw (a vector-based drawing program) for storing structured graphics.
- **X Windows PPM (Portable PixMap) :-** the graphics format for the X Window system. PPM supports 24-bit color
- bitmaps, and can be manipulated using many public domain graphic editors.

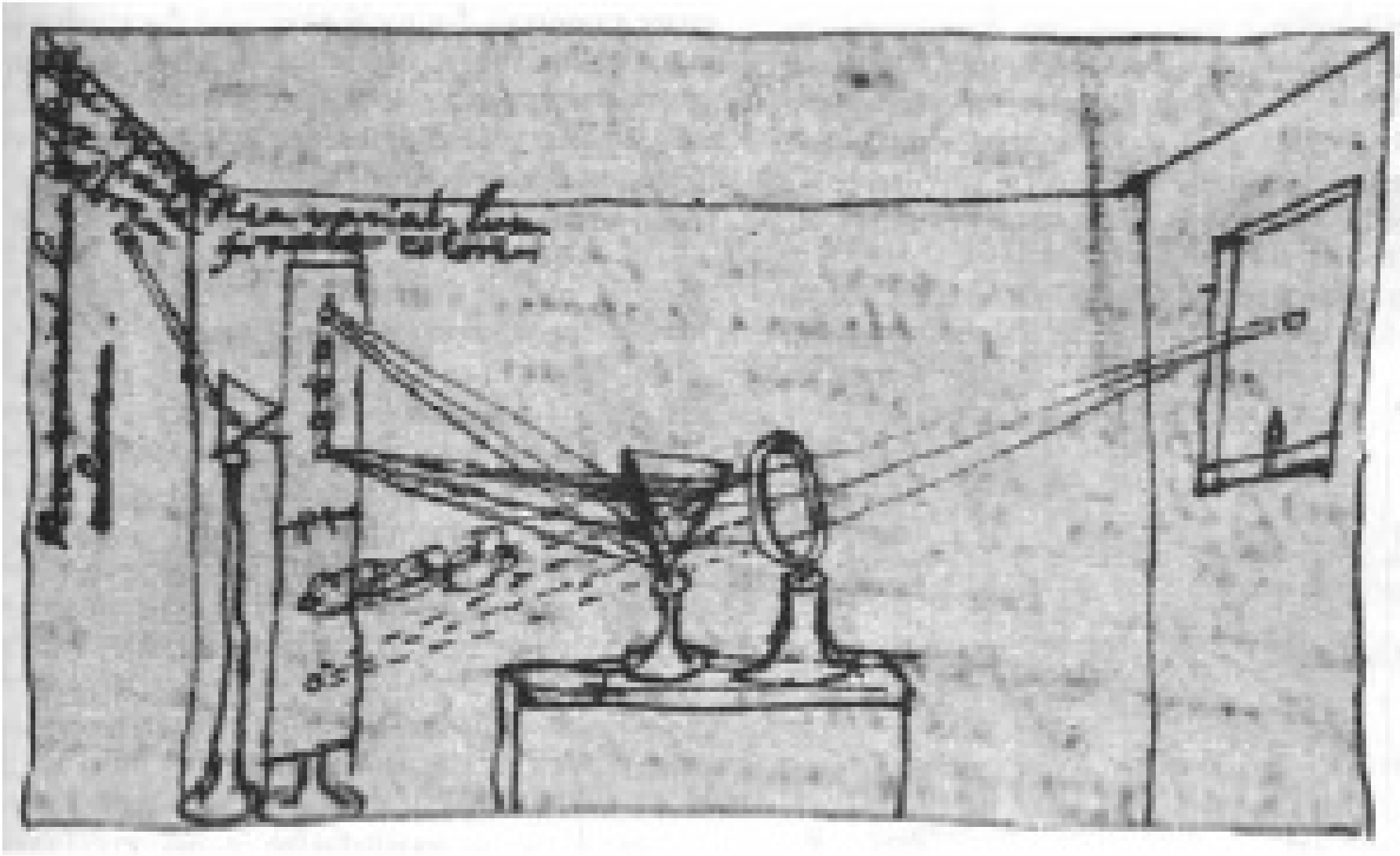
Color in Image and Video

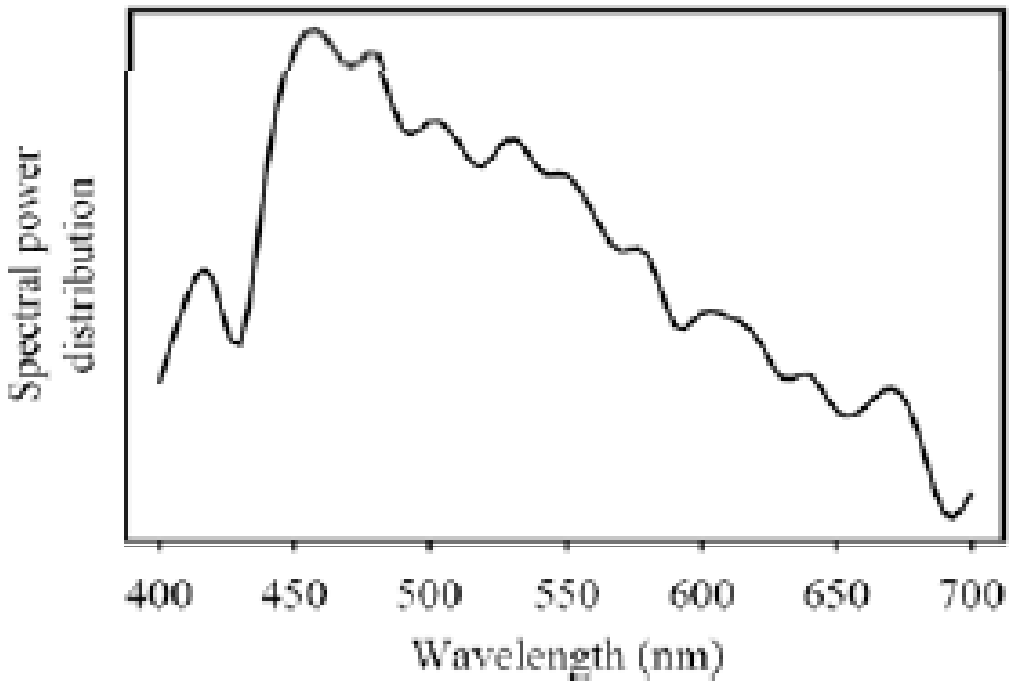
- Color images and videos are everywhere on the web and in multimedia production.
- Also we know that there is discrepancies between the color as seen by the people and displayed on the screens.
- Here we will study following topics.
  - Color Science.
  - Color Models in Images.
  - Color models in Videos.

Color Science

- Light and Spectra :- Light is an electromagnetic wave. Its color is characterized by the wavelength content of the light.
- Laser light consists of a single wavelength: e.g., a ruby laser produces a bright, scarlet-red beam.
- Most light sources produce contributions over many wavelengths.
- However, humans cannot detect all light, just contributions that fall in the "visible wavelengths".
- Short wavelengths produce a blue sensation, long wavelengths produce a red one.

- Spectrophotometer: device used to measure visible light, by reflecting light from a diffraction grating (a ruled surface) that spreads out the different wavelengths.
- Visible light is an electromagnetic wave in the 400 nm to 700 nm.





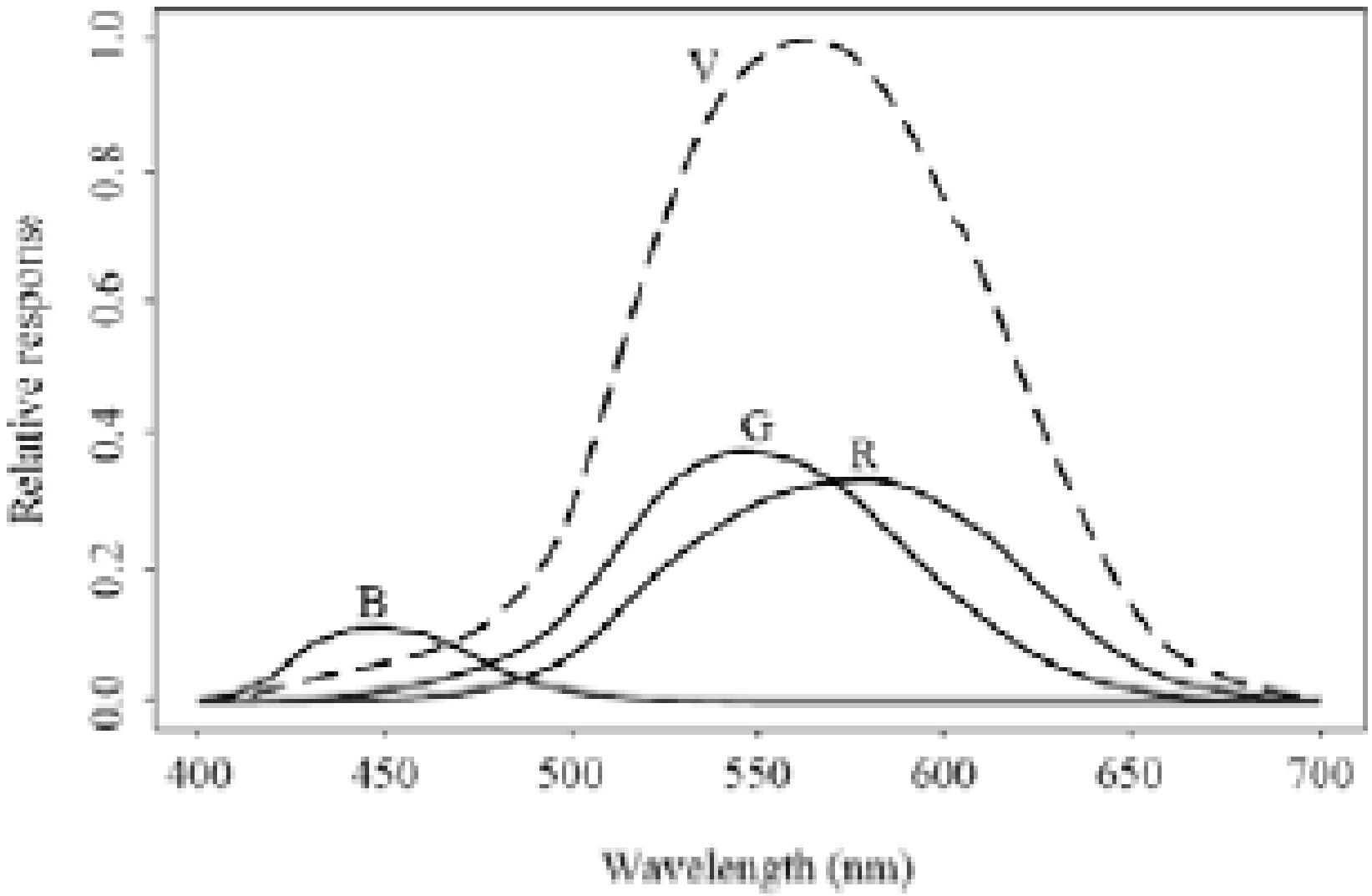
- Fig shows the relative power in each wavelength interval for typical outdoor light on a sunny day.
- This type of curve is called a Spectral Power Distribution (SPD) or a spectrum.
- The symbol for wavelength is λ. This curve is called E(λ).

Human Vision

- Eye works like Camera.
- Retina consists of array of rods (for low light levels and three kinds of cones (for higher light levels).
- The brain makes use of differences R-G, G-B, and B-R, as well as combining all of R, G and B into a high-light-level achromatic channel.

Spectral Sensitivity of the Eye

- The eye is most sensitive to light in the middle of the visible spectrum.
- Fig. shows the overall sensitivity as a dashed line this important curve is called the luminous-efficiency function.
- It is usually denoted V (λ) and is formed as the sum of the response curves for Red, Green, and Blue.



- The eye has about 6 million cones, but the proportions of R, G and B cones are different
- They likely are present in the ratios 40:20:1
- So the achromatic(without color) channel produced by the cones is approximately proportional to

  $2R + G + B/20.$
- These spectral sensitivity functions are usually denoted by letters other than "R, G, B“.
- We use a vector function q(λ), with components

  $q(\lambda)=[q_R(\lambda),q_G(\lambda),q_B(\lambda)]^T$

- The response in each color channel in the eye is proportional to the number of neurons firing.
- A laser light at wavelength λ would result in a certain number of neurons firing. An SPD (spectral power distribution) is a combination of single-frequency lights (like lasers), so we add up the cone responses for all wavelengths, weighted by the eye's relative response at that wavelength.

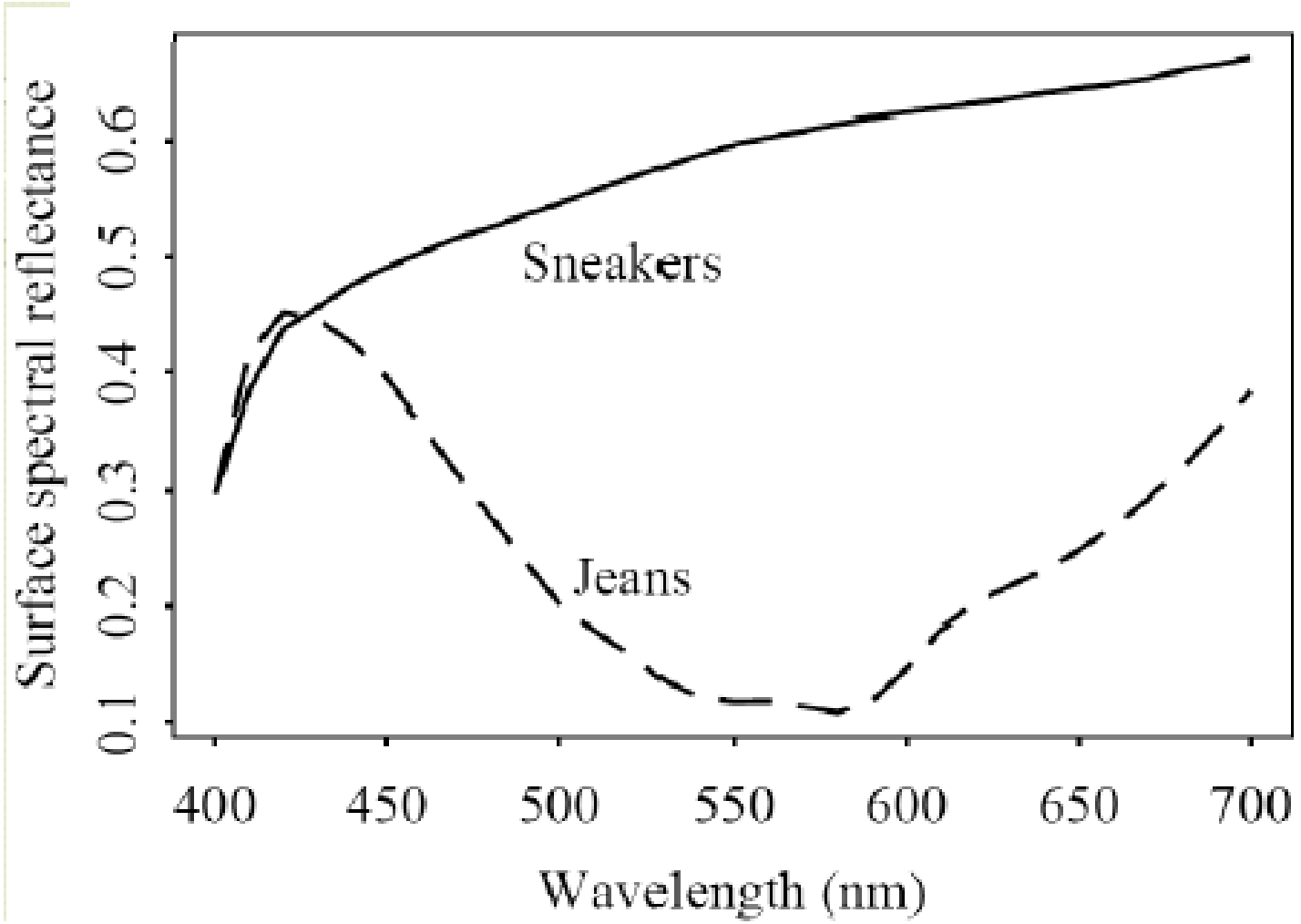
$$R = \int E(\lambda)\, q_R(\lambda)\, d\lambda$$

$$G = \int E(\lambda)\, q_G(\lambda)\, d\lambda$$

$$B = \int E(\lambda)\, q_B(\lambda)\, d\lambda$$

Image Formation

- The above equations applies only when we view a self-luminous object.
- In many situations, we image light reflected from a surface.
- Surfaces reflect different amounts of light at different wavelengths, and dark surfaces reflect less energy than light surfaces.



The surface spectral reflectance from (1) orange sneakers and (2) faded bluejeans .

The reflectance function is denoted S(λ).

Image formation is thus:

- Light from the illuminant with SPD E(λ) impinges on a surface, with surface spectral reflectance function S(λ), is reflected, and then is filtered by the eye's cone functions q (λ).
- Reflection is shown in Fig. below.
- The function C(λ) is called the color signal and consists of the product of E(λ), the p (), illuminant, times S(λ), the reflectance:

$$C(\lambda) = E(\lambda)\, S(\lambda).$$

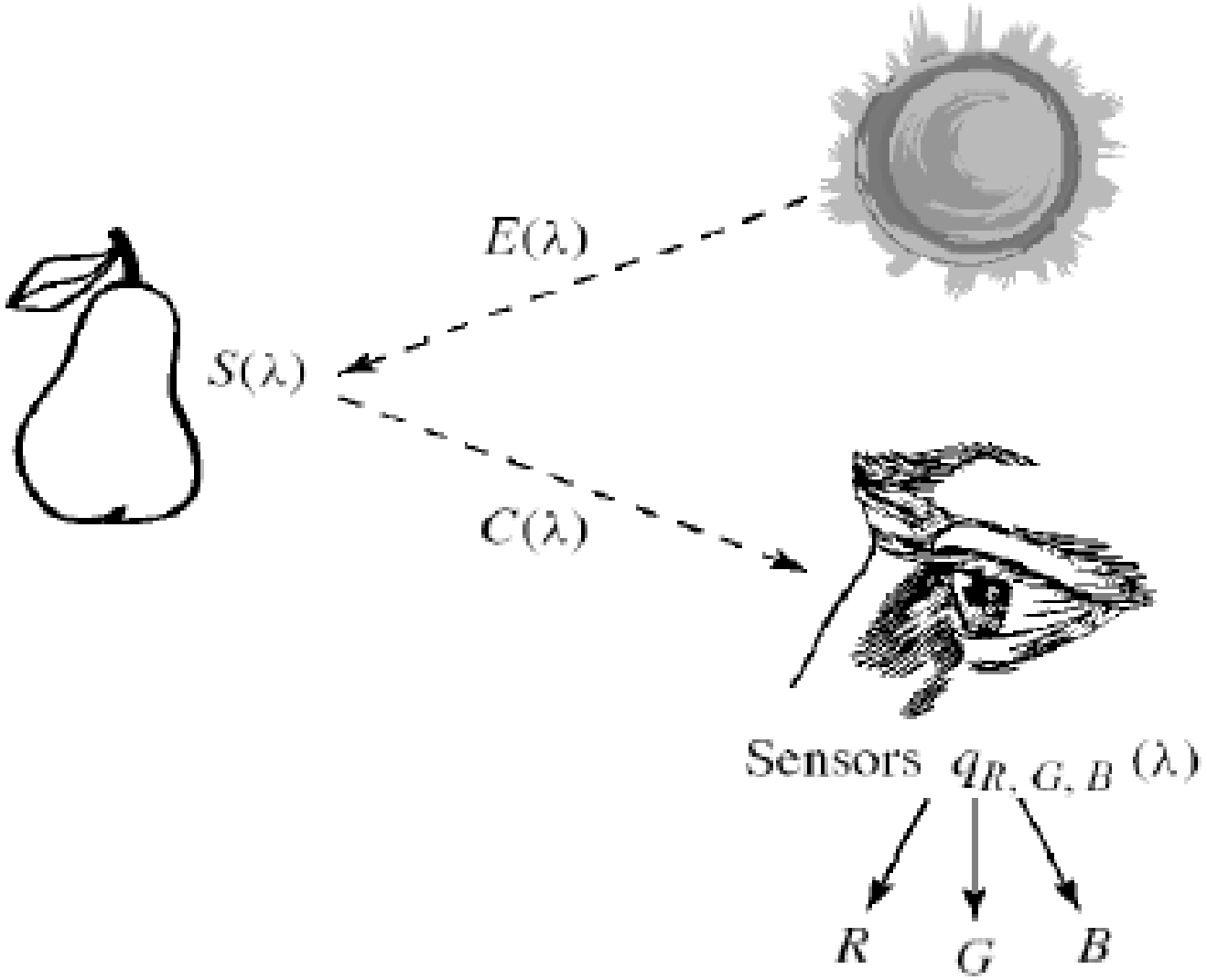


Fig. 4.5: Image formation model

- The equations that take into account the image formation model are:

$$R = \int E(\lambda)\, S(\lambda)\, q_R(\lambda)\, d\lambda$$

$$G = \int E(\lambda)\, S(\lambda)\, q_G(\lambda)\, d\lambda$$

$$B = \int E(\lambda)\, S(\lambda)\, q_B(\lambda)\, d\lambda \qquad (4$$

Camera Systems

Gamma Correction

- The light emitted is in fact roughly proportional to the voltage raised to a power; this power is called gamma, with symbol γ.

a) Thus, if the file value in the red channel is R, the screen emits light proportional to Rγ, with SPD equal to that of the red phosphor paint on the screen that is the target of the red channel electron gun. The value of gamma is around 2.2.

b) It is customary to append a prime to signals that are gamma corrected by raising to the gamma-power (1/γ) before transmission. Thus we arrive at linear signals:

$$R \rightarrow R' = R^{1/\gamma} \Rightarrow (R')^{\gamma} \rightarrow R$$

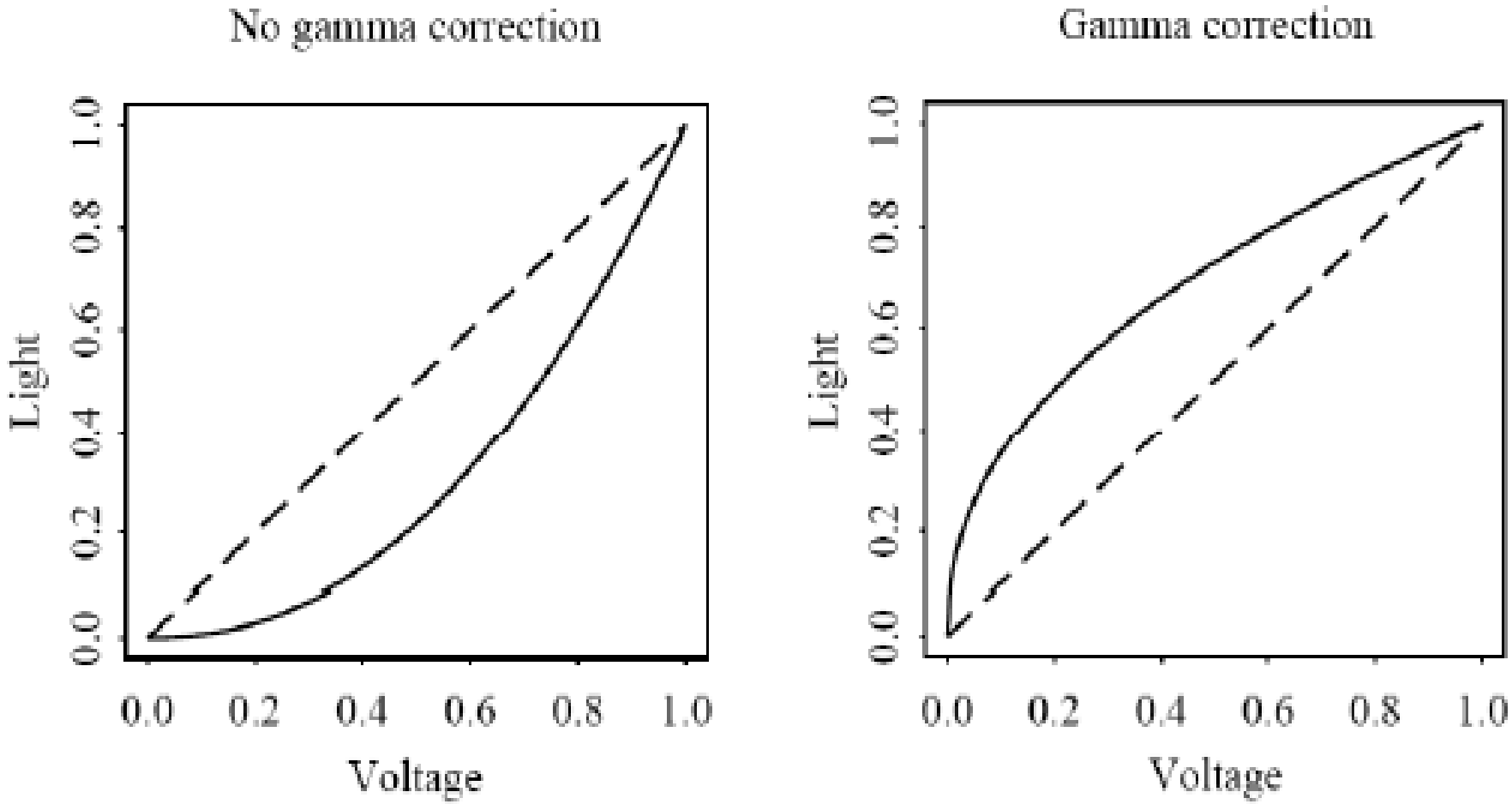


Fig. 4.6: (a): Effect of CRT on light emitted from screen (voltage is normalized to range 0..1). (b): Gamma correction of signal.

- Fig 4 6(a) shows light output with gamma-correction applied. We see that darker values are displayed too dark. This is also shown in Fig. 4.7(a), which displays a linear ramp from left to right.
- Fig 4 6(b) the effect of pre-correcting signals by applying the power law $R^{1/\gamma}$; it is customary to normalize voltage to the range [0,1].

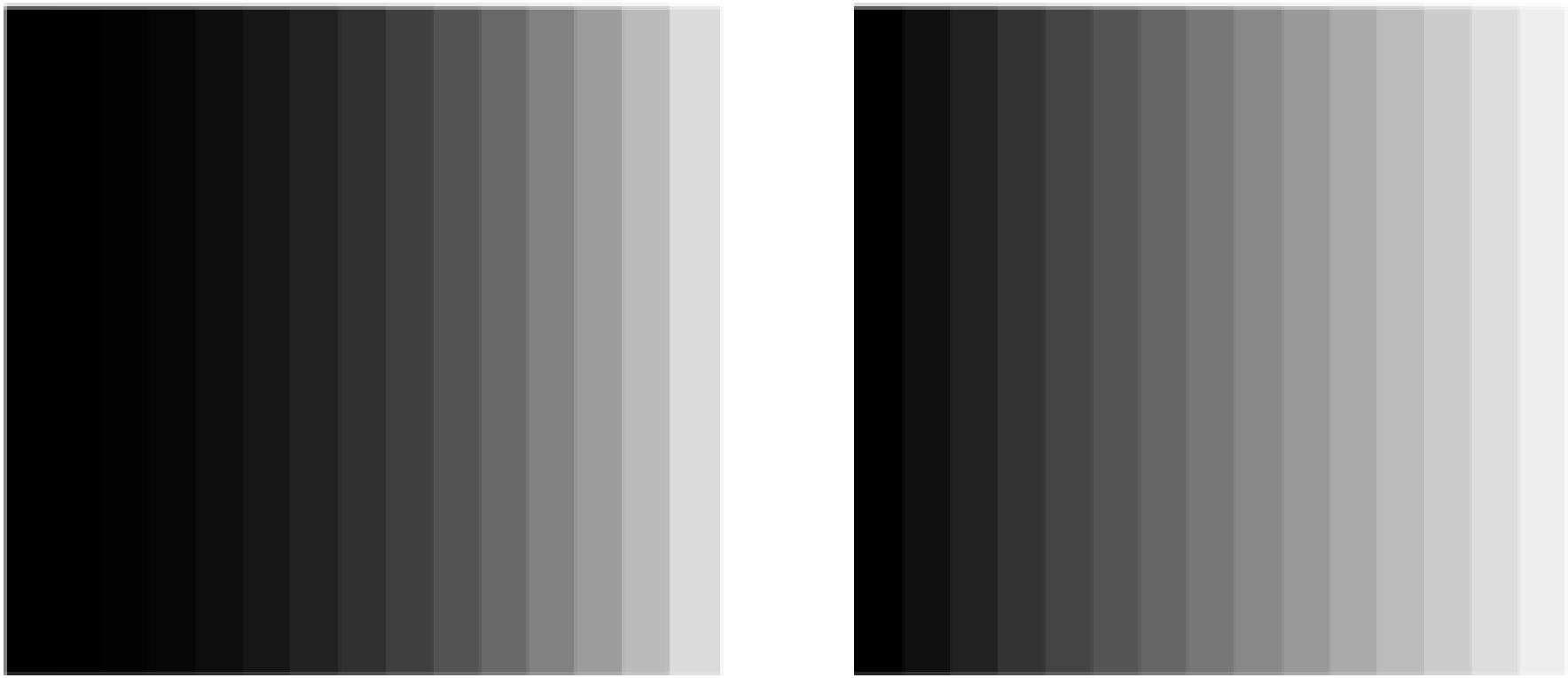


Fig. 4.7: (a): Display of ramp from 0 to 255, with no gamma correction. (b): Image with gamma correction applied

Color-Matching Functions

- Even without knowing the eye-sensitivity curves, a technique evolved in psychology for matching a combination of basic R, G, and B lights to given shade.
- The particular set of three basic lights used in an experiment are called the set of color primaries.
- To match a given color, a subject is asked to separately adjust the brightness of the three primaries using a set of controls until the resulting spot of light most closely matches the desired color.
- The basic situation is shown in Fig.

A device for carrying out such an experiment is called a colorimeter.

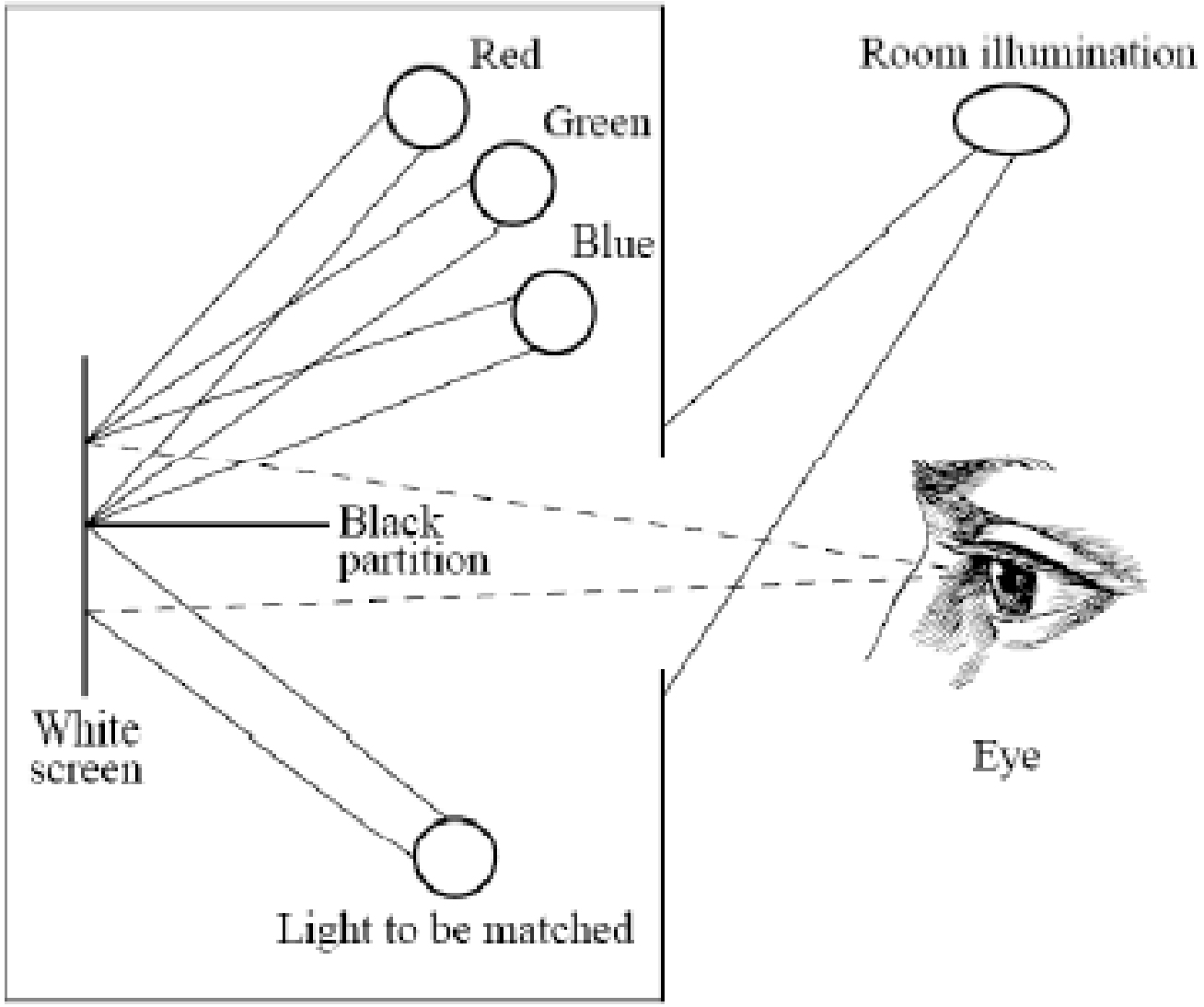


Fig. 4.8: Colorimeter experiment.

Color Monitor Specifications

- Color monitors are specified in part by the white point chromaticity that is desired if the RGB electron guns are all activated at their highest value (1.0, if we normalize to [0,1]).
- We want the monitor to display a specified white when R'=G'=B'=1.
- There are several monitor specifications in current use (Table 4.1).

Table 4.1: Chromaticities and White Points of Monitor Specifications

System	Red		Green		Blue		White Point	
	x_r	y_r	x_g	y_g	x_b	y_b	x_W	y_W
NTSC	0.67	0.33	0.21	0.71	0.14	0.08	0.3101	0.3162
SMPTE	0.630	0.340	0.310	0.595	0.155	0.070	0.3127	0.3291
EBU	0.64	0.33	0.29	0.60	0.15	0.06	0.3127	0.3291

Color Models in Images

- **RGB Color Model for CRT Displays**

1. We expect to be able to use bits per color channel for color that is accurate enough.
2. However, in fact we have to use about 12 bits per channel to avoid an aliasing effect in dark image areas —contour bands that result from gamma correction.
3. For images produced from computer graphics, we store integers proportional to intensity in the frame buffer. So should have a gamma correction LUT between the frame buffer and the CRT.
4. If gamma correction is applied to floats before quantizing to integers, before storage in the frame buffer, then in fact we can use only 8 bits per channel and still avoid contouring artifacts.

Subtractive Color: CMY Color Model

- So far we have effectively been dealing, only with additive color. Namely, when two light beams impinge on a target, their colors add; when two phosphors on a CRT screen are turned on, their colors add.
- But for ink deposited on paper, the opposite situation holds: yellow ink subtracts blue from white illumination, but reflects red and green; it appears yellow.
- Instead of red, green, and blue primaries, we need primaries that amount to -red, -green, -blue. I.e., we need to subtract R, or G, or B.
- These subtractive color primaries are Cyan (C), Magenta (M) and Yellow (Y)inks.

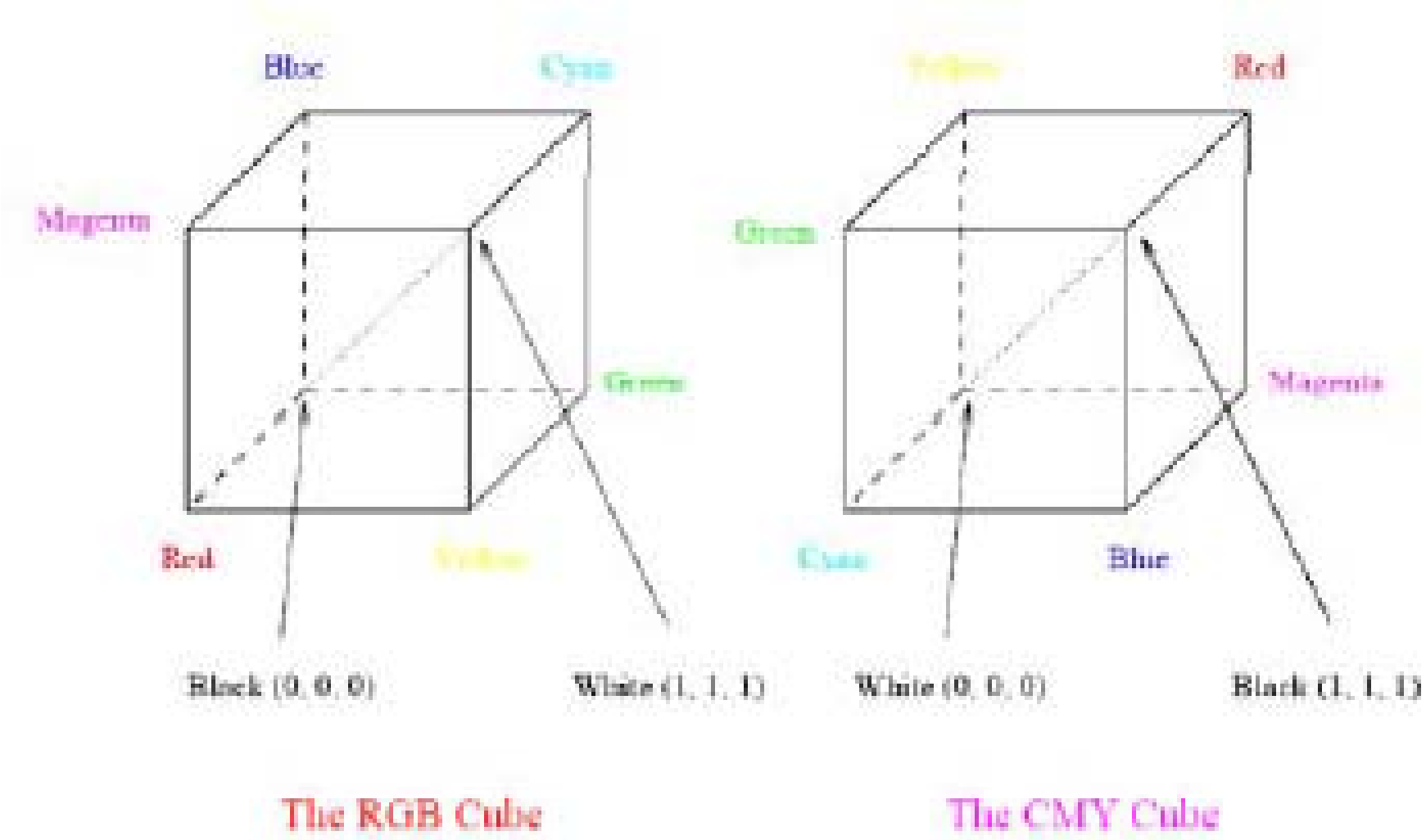


Fig. 4.15: RGB and CMY color cubes.

Transformation from RGB to CMY

- Simplest model we can invent to specify what ink density to lay down on paper, to make a certain desired RGB color.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- Then the inverse transform is:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

color combinations that result from combining primary colors available in the two situations, additive color and subtractive color.

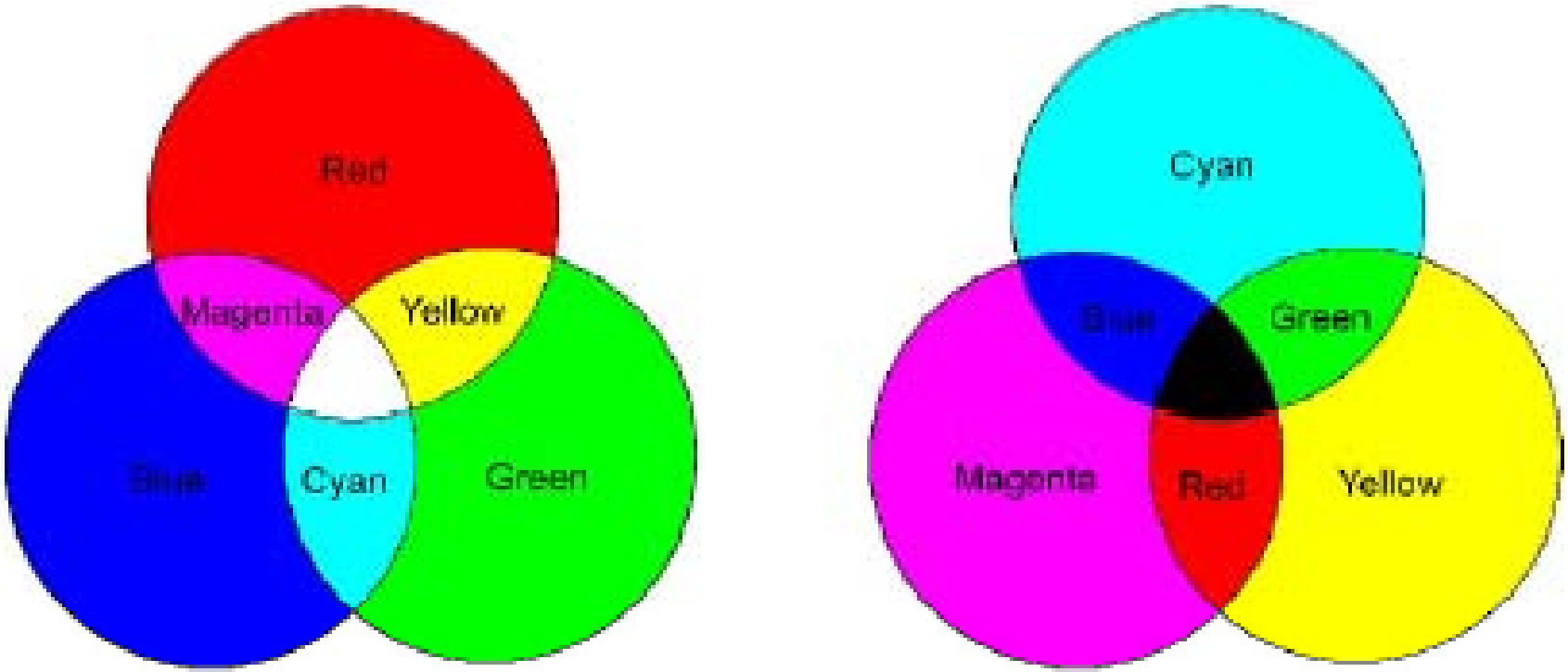


Fig. 4.16: Additive and subtractive color. (a): RGB is used to specify additive color. (b): CMY is used to specify subtractive color

Printer Gamuts

- Actual transmission curves overlap for the C, M, Y inks. This leads to "crosstalk“ between the color channels and difficulties in predicting colors achievable in printing.
- Fig (a) shows typical transmission for real "block dyes", and Fig.(b) shows the resulting color gamut for a color printer.

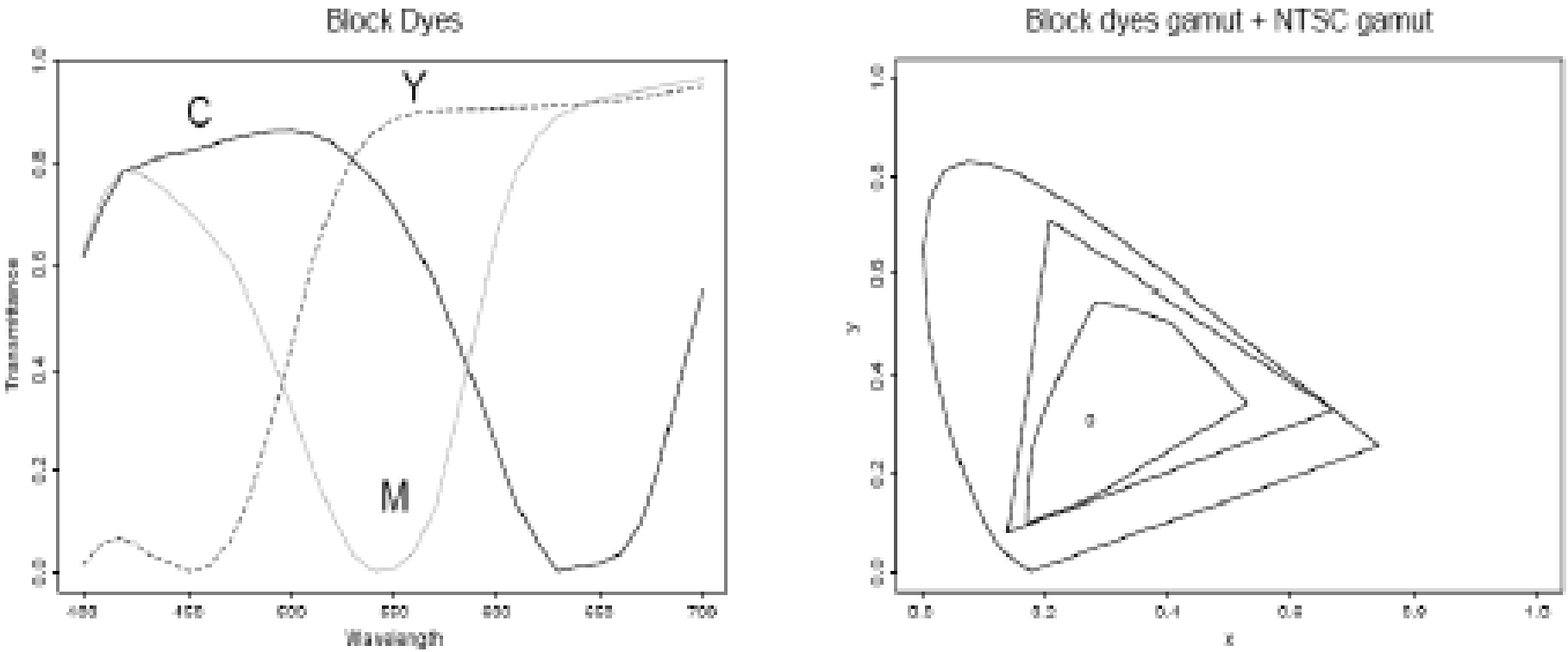


Fig. 4.17: (a): Transmission curves for block dyes. (b): Spectrum locus, triangular NTSC gamut, and 6-vertex printer gamut

Color Models in Video

Video Color Transforms

- Methods of dealing with color in digital video derive largely from older analog methods of coding color for TV. Luminance is separated from color information.
- For example, a matrix transform method called YIQ is used to transmit TV signals in North America and Japan.
- This coding also makes its way into VHS video tape coding in these countries since video tape technologies also use YIQ.
- In Europe, video tape uses the PAL or SECAM codings, which are based on TV that uses a matrix transform called YUV.
- Digital video mostly uses a matrix transform called YCbCr that is closely related to YUV.

YUV Color Model

- YUV codes a luminance signal (for gamma-corrected signals) equal to Y’ in Eq. (4.20). the "luma".
- Chrominance refers to the difference between a color and a reference white at the same luminance → use color differences U, V :
- U = B’–Y’ V = R’–Y’ (4.27)
- From Eq. (4.20) [Y’ =0.299·R’+0.587·G’+0.114·B’] & eq (4.27)

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \quad (4.28)$$

- For gray, R’ = G’ = B’ , the luminance Y’ equals to that gray, since 0.299 + 0.587 + 0.114 = 1.0. And for a gray ("black & white") image the chrominance (U, V)is zero.
- In the actual implementation U and V are rescaled to have a more convenient maximum and minimum.
- For dealing with composite video, it turns out to be convenient to contain U, V within the range − 1/3to +4/3. So U and V are rescaled:
- U =0.492111 (B’− Y’)
- V =0.877283 (R’− Y’) (4.29)
- The chrominance signal = the composite signal C:
- $C = U \cdot \cos(\omega t) + V \cdot \sin(\omega t)$ (4.30)

- Zero is not the minimum value for U, V.
- U is approximately from blue (U> 0) to yellow (U< 0) in the RGB cube; V is approximately from red (V> 0) to cyan (V< 0).
- Fig. shows the decomposition of a color image into its Y', U, V components. Since both U and V go negative, in fact the images displayed are shifted and rescaled.

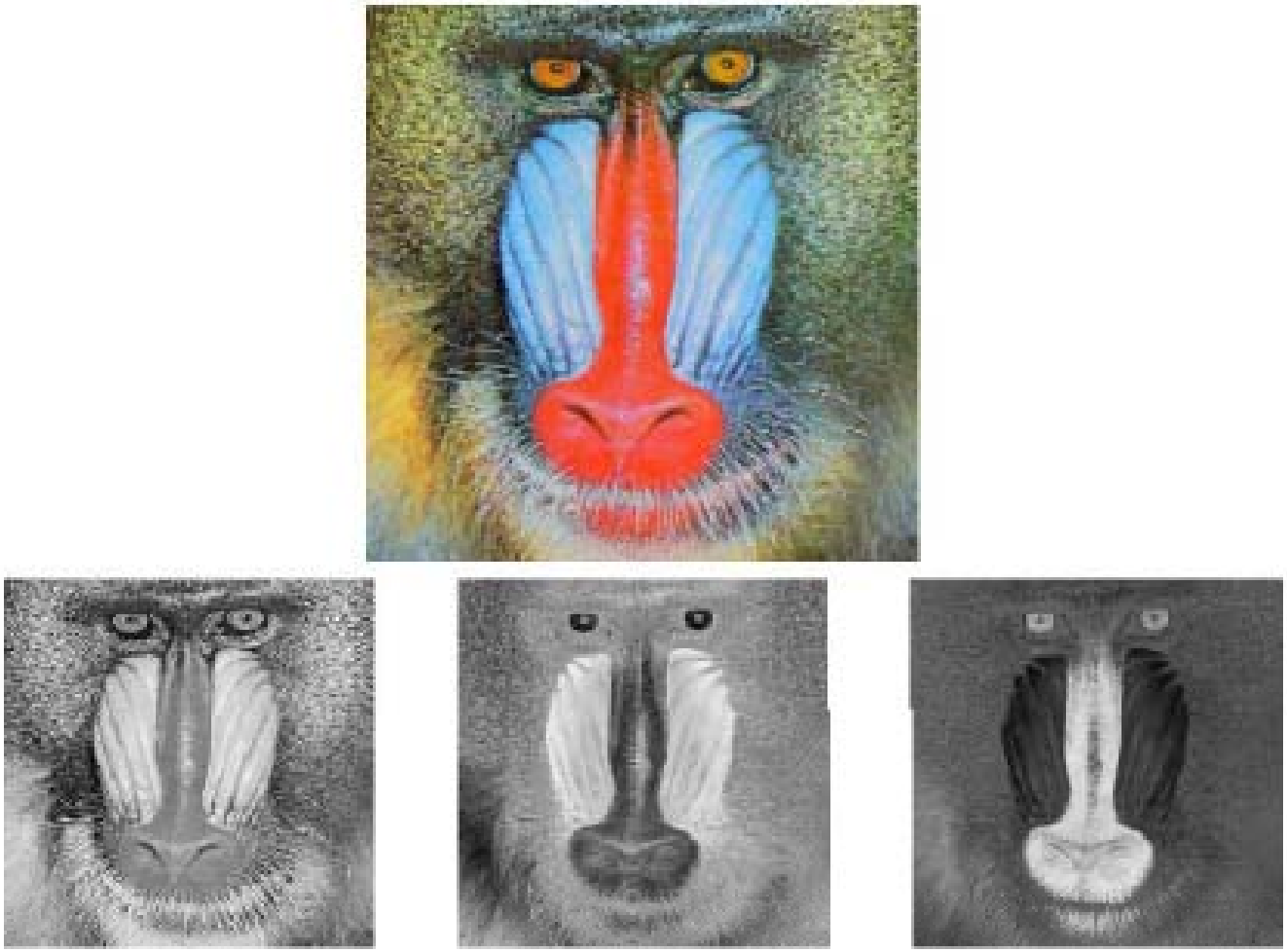


Fig. 4.18: Y'UV decomposition of color image. Top image (a) is original color image; (b) is Y'; (c,d) are (U,V)

UNIT I
Introduction to Multimedia

By :-
Vijaykumar Mantri,
Asso. Prof. in IT Dept.,
BVRIT, Narsapur

What is Multimedia?

- When different people mention the term multimedia, they often have quite different, or even opposing, viewpoints.
- A PC vendor: a PC that has sound capability, a DVD-ROM drive, and perhaps the superiority of multimedia-enabled microprocessors that understand additional multimedia instructions.
- A consumer entertainment vendor: interactive cable TV with hundreds of digital channels available, or a cable TV-like service delivered over a high-speed connection high Internet connection.
- A Computer Science (CS) student: applications that use multiple modalities, including text, images, drawings (graphics), animation, video sound including speech and interactivity video, speech, interactivity.

- **Multimedia and Computer Science:** Graphics, HCI, visualization, computer vision, data compression, graph theory, networking, database systems.

Components of Multimedia

- Multimedia involves multiple modalities of text, audio, images, drawings, animation, and video.
- Examples of how these modalities are put to use:
- Video teleconferencing.
- Distributed lectures for higher education.
- Tele-medicine.
- Co-operative work environments.
- Searching in (very) large video and image databases for target visual objects.
- "Augmented"reality: placing real-appearing computer video scenes graphics and objects into scenes.

- Including audio cues for where video-conference participants are located.
- Building searchable features into new video, and enabling very high-to very low-bit-rate new use of new, scalable multimedia products.
- Making components editable multimedia editable.
- Building "inverse-Hollywood" applications that can recreate the process by which a video was made.
- **Video understanding** has also been called an inverse Hollywood problem.
- Using voice-recognition to build an interactive environment, say a kitchen-wall web browser.

Multimedia Research Topics & Projects

- To the computer science researcher, multimedia consists of a wide variety of topics:
- **Multimedia processing and coding**: multimedia content analysis, content-based multimedia retrieval, multimedia security, audio/image/video processing, compression, etc.
- **Multimedia system support and networking:** network protocols, Internet, operating systems, servers and clients, quality of service (QoS), and databases.
- **Multimedia tools end-systems applications:** Hypermedia systems, user interfaces, authoring systems.
- Multi-modal interaction and integration: "ubiquity"—web-everywhere devices, multimedia education including Computer Supported Collaborative Learning, and design and applications of virtual environments.

Current Multimedia Projects

- Many exciting research projects are currently underway. Here area few of them:

1. **Camera-based object tracking technology:** tracking of the control objects provides user control of the process.
2. **3D motion capture:** used for multiple actor capture so that multiple actors in a virtual studio can be used real to automatically produce realistic animated models with natural movement.
3. **Multiple views:** allowing photo-realistic (video-quality) synthesis of virtual actors from several cameras or from a single camera under differing lighting.
4. **3D capture technology:** allow synthesis of highly realistic speech facial animation from speech.

5. **Specific multimedia applications:** aimed at handicapped persons with low vision capability and the elderly —a rich field of endeavor.
6. **Digital fashion:** aims to develop smart clothing that can communicate with other such enhanced clothing using wireless communication, so as to artificially enhance human interaction in a social setting.
7 **Electronic Housecall system:** an initiative for providing interactive health monitoring services to patients in their homes
8. **Augmented Interaction applications:** used to develop interfaces between real and virtual humans for tasks such as augmented storytelling .

Multimedia and Hypermedia

- **History of Multimedia:**

1. Newspaper: perhaps the first mass communication medium, uses text, graphics, and images.
2. Motion pictures: conceived of in 1830's in order to observe motion too rapid for perception by the human eye.
3. Wireless radio transmission: Guglielmo Marconi, at Pontecchio, Italy, in 1895.
4. Television: the new medium for the 20th century, established video as a commonly available medium and has since changed the world of mass communications .

5. The connection between computers and ideas about multimedia covers what is actually only a short period:

1945 – Vannevar Bush wrote a landmark article describing what amounts to a hypermedia system called Memex.

1960 – Ted Nelson coined the term hypertext.

1967 – Nicholas Negroponte formed the Architecture Machine Group.

1968 – Douglas Engelbart demonstrated the On-Line System (NLS), another very early hypertext program.

1969 – Nelson hypertext and van Dam at Brown University created an early editor called FRESS.

1976 – The MIT Architecture Machine Group proposed a project entitled Multiple Media — resulted in the Aspen Movie Map, hypermedia videodisk, in 1978.

1985 – Negroponte and Wiesner co-founded the MIT Media Lab.

1989 – Tim Berners-Lee proposed the World Wide Web

1990 – Kristina Hooper Woolsey headed the Apple Multimedia Lab.

1991 MPEG 1– MPEG [M(oving) P(ictures) E(xperts) G(roup)]was approved as an international standard for digital video — led to the newer standards, MPEG-2, MPEG-4, and further MPEGs in the 1990s.

1991 – The introduction of PDAs in 1991 began a new period in the use of computers in multimedia.

1992 – JPEG was accepted as the international standard for digital image compression — led to the new JPEG2000 standard.

1992 – The first MBone audio multicast on the Net was made.

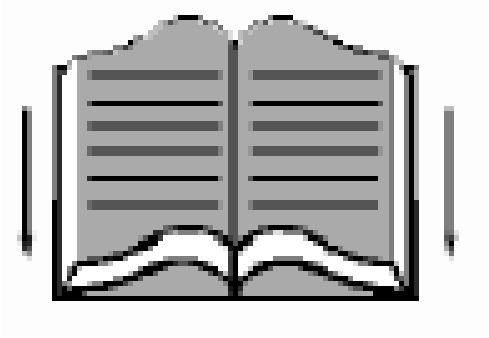
1993 The of Illinois National Center for Supercomputing– University Applications produced NCSA Mosaic — the first full-fledged browser.

- 1994 – Jim Clark and Marc Andreessen created the Netscape program .
- 1995 – The JAVA language was created for platform independent application development.
- 1996 – DVD video was introduced; high quality full-length movies were distributed on a single disk.
- 1998 – XML 1.0 was announced as a W3C Recommendation.
- 1998 – Hand-held MP3 devices first made inroads into consumerist tastes in the fall of 1998, with the introduction holding of flash of devices 32MB memory.
- 2000 – WWW size was estimated at over 1 billion pages.

Hypermedia and Multimedia

- A hypertext system: meant to be read nonlinearly, by following links that point to other parts of the document, or to other documents.
- HyperMedia: not constrained to be text-based, can include other media, e.g., graphics, images, and especially the continuous media sound and video.
  - The World Wide Web (WWW) — the best example of a hypermedia application.
- Multimedia means that computer information can be represented through audio graphics images audio, graphics, images,
- video, and animation in addition to traditional media.

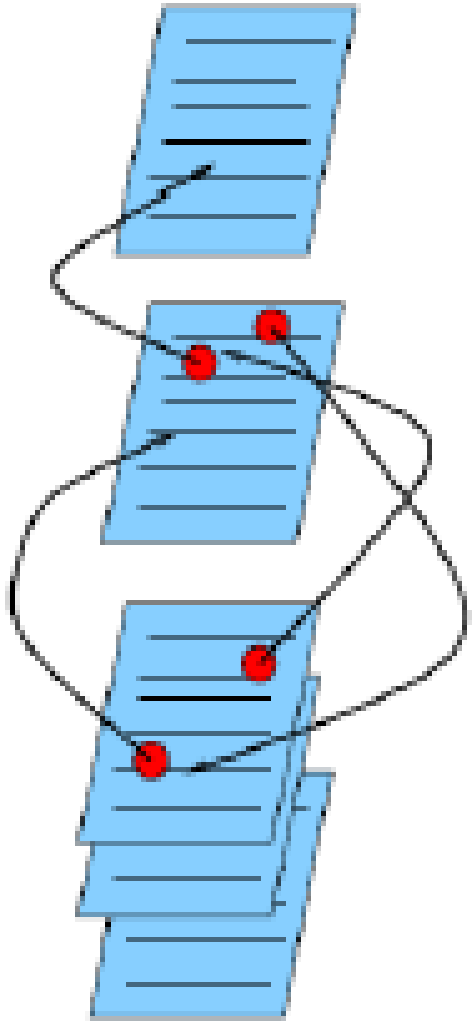
Normal Text



Linear

"Hot spots"

Hypertext



Nonlinear

Examples of typical present multimedia applications include:

- Digital video editing and production systems.
- Electronic newspapers/magazines.
- World Wide Web.
- On-line reference works: e.g. encyclopedias, games.
- Home shopping.
- Interactive TV.
- Multimedia courseware.
- Video conferencing.
- Video-on-demand.
- Interactive movies.

World Wide Web

- The W3C has listed the following goals for the WWW

1. Universal access of web resources (by everyone everywhere).
2. Effectiveness of navigating available information.
3. Responsible use of posted material.

HTTP (HyperText Transfer Protocol)

- HTTP: a protocol that was originally designed for transmitting hypermedia but can also support transmission of any file type.
- HTTP a stateless request/response protocol: is no information carried over for the next request.
- The basic request format:

  Method URI Version

  Additional-Headers:

  Message-body
- The URI (Uniform Resource Identifier): an identifier for the resource accessed, e.g. the host name, always preceded by the token http://.

- Two popular methods: GET and POST.
- The basic response format:
  – Version Status-Code Status-Phrase
  – Additional-Headers
- Additional – Message-body
- Two commonly seen status codes:
- 200 OK — the request was processed successfully.
- 404 Not Found — the URI does not exist.

HTML (HyperText Markup Language)

- HTML: a language for publishing Hypermedia on the World Wide Web — defined using SGML:

1. HTML uses ASCII, it is portable to all different (possibly binary incompatible) computer hardware.
2. The current version of HTML is version 4.01.
3. The next generation of HTML is XHTML —a reformulation of HTML using XML.

- HTML uses tags to describe document elements:

<token params> —defining a starting point,

</token> — the ending point of the element.

Some elements have no ending tags.

- A very simple HTML page is as follows:

```
<HTML> <HEAD>
 <TITLE>
 A sample web page.
 </TITLE>
 <META NAME = "Author" CONTENT = "Cranky Professor">
</HEAD> <BODY>
 <P>
 We can put any text we like here, since this is
 a paragraph element.
 </P>
</BODY> </HTML>
```

- Naturally, HTML has more complex structures and can be mixed in with other standards.

XML (Extensible Markup Language)

- XML: a markup language for the WWW in which there is modularity of data, structure and view so that user or application can be able to define the tags (structure).
- Example of using XML to retrieve stock information from a database a query according to user

1. First use a global Document Type Definition (DTD) that is already defined.
2. The server side script will abide by the DTD rules to generate document an XML according to the query using data from your database.
3. Finally send user the XML Style Sheet (XSL) depending on the type of device used to display the information.

- The current XML version is XML 1.0, approved by the W3C in Feb. 1998.
- XML syntax looks like HTML syntax, although it is much more strict:
  - All tags are in lower case and a tag has only inline data has to terminate itself, i.e., <token params />.
  - Uses name spaces so that multiple DTDs declaring different elements but with similar tag names can have their elements distinguished.
  - DTDs can be imported from URIs as well.

An example of an XML document structure — the definition for a small XHTML document:

```
<?xml version="1.0" encoding="iso-8859-1"?> <!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
                              transition.dtd">
 <html xmlns="http://www.w3.org/1999/xhtml">
 ... [html that follows
     the above mentioned
     XML rules]
 </html>
```

- The following XML related specifications are also standardized:

  – XML Protocol: used to exchange XML information between processes .

  – XML Schema: a more structured and powerful language defining XML data types (tags).

  – XSL: basically CSS for XML.

  – SMIL : synchronized Integration Multimedia Language, pronounced "smile"— a particular application of XML (globally predefined DTD) that allows for specification of interaction among any media types and user input, in a temporally scripted manner.

SMIL (Synchronized Multimedia Integration Language)

- Purpose of SMIL: it is also desirable to be able to publish presentations using multimedia a markup language.
- A multimedia language needs to scheduling and synchronization of different multimedia elements, and define their interactivity with the user.
- The W3C established a Working Group in 1997 to come up with specifications for a multimedia synchronization language
- SMIL 2 0 was accepted August 2001 2.0 in 2001.

- SMIL 2.0 is specified in XML using a modularization approach similar to the one used in xhtml:

1. All SMIL elements are divided into modules — sets of elements, attributes and values that define one conceptual functionality.
2. In the interest of modularization, not all available modules need to be included for all applications.
3. Language Profiles: specifies a particular grouping of modules, and particular modules may have integration profile must requirements that a follow.

```
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 2.0"
"http://www.w3.org/2001/SMIL20/SMIL20.dtd">
<smil xlmns=
"http://www.w3.org/2001/SMIL20/Language">
<head>
    <meta name="Author" content="Some Professor" />
</head>
<body>
    <par id="MakingOfABook">
        <seq>
        <video src="authorview.mpg" />
        <img src="onagoodday.jpg" />
        </seq>

        <audio src="authorview.wav" />
        <text src="http://www.cs.sfu.ca/mmbook/" />
    </par>
</body>
</smil>
```

Overview of Multimedia Software Tools

1. Music Sequencing and Notation

- **Cakewalk**: now called Pro Audio.

  The term sequencer comes from older devices that stored sequences of notes ("events", in MIDI).

  It is also possible to insert WAV files and Windows MCI commands (for animation and video) into music tracks (MCI is a ubiquitous component of the Windows API.)
- **Cubase:** another sequencing/editing program, with capabilities similar to those of Cakewalk. It includes some tools digital audio editing tools.
- **Macromedia Soundedit**: mature program for creating audio for multimedia projects and the web that integrates well with other Macromedia products such as Flash and Director.

2. Digital Audio

- **Cool Edit**: a very powerful and popular digital audio toolkit; emulates a professional audio studio —multitrack productions and sound file editing including digital signal processing effects.
- **Sound Forge**: a sophisticated PC-based program for WAV audio files.
- **Pro Tools**: a high-end integrated audio production and editing environment MIDI creation and manipulation; powerful audio mixing, recording,and editing software.

3.Graphics and Image Editing

- **Adobe Illustrator**: a powerful publishing tool from Adobe. Uses vector graphics; can be exported to Web graphics Web.
- **Adobe Photoshop:** the standard in a graphics, image processing and manipulation tool.
  - Allows layers of images, graphics, and text that can be separately manipulated for maximum flexibility.
  - Filter factory permits creation of sophisticated lighting-effects filters.
- **Macromedia Fireworks**: software for making graphics specifically for the web.
- **Macromedia Freehand:** a text and web graphics editing tool that supports many bitmap formats such as GIF, PNG, and JPEG.

4.Video Editing

- **Adobe Premiere:** an intuitive, simple video editing tool editing i e clips for nonlinear editing, i.e., putting video into any order:

  Video and audio are arranged in "tracks" tracks .

  Provides a large number of video and audio tracks, superimpositions and virtual clips. => effective multimedia productions with little effort.

- **Adobe After Effects**: a powerful video editing tool that enables users to add and change existing movies. Can add many effects: lighting, shadows, motion blurring; layers.
- **Final Cut Pro**: a video editing tool by Apple; Macintosh only.

5. Animation

Multimedia APIs:

- Java3D: API used by Java to construct and render 3D graphics, similar to the way in which the Java Media Framework is used for handling media files.

1 Provides a basic set of object primitives (cube, splines, etc.) for building scenes.

2. It is an abstraction layer built on top of OpenGL or DirectX (the user can select which).

- DirectX : Windows API that supports video, images, audio and 3-D animation
- OpenGL: the highly portable most popular 3 D API

Rendering Tools:

- **3D Studio Max**: rendering tool that includes a number of very high-end professional tools for character animation, game development, and visual effects production.
- **Softimage XSI:** a powerful modeling, animation, and rendering package used for animation and special effects in films and games.
- **Maya**: competing product to Softimage; as well, it is a complete modeling package.
- **RenderMan:** rendering package created by Pixar.
- **GIF Animation Packages:** a simpler approach to animation, allows very quick development of effective small animations for the web.

6.Multimedia Authoring

- **Macromedia Flash**: allows users to create interactive movies using the score metaphor i.e. a timeline arranged in parallel event sequences.
- **Macromedia Director**: uses a movie metaphor to create interactive presentations — very powerful and includes a built-in scripting language, Lingo, that allows creation of complex interactive movies.
- **Authorware:** a mature, well-supported authoring product based on the Iconic/Flow-control metaphor.
- **Quest**: similar to Authorware in many ways, uses a type of flowcharting metaphor. However, flowchart nodes can encapsulate information in a more abstract way (called frames) than simply subroutine levels.

Graphics and Image Data Representations

Graphics/Image Data Types

- The number of file formats used in multimedia continues to rapidly.

File Import					File Export		Native
Image	Palette	Sound	Video	Anim.	Image	Video	
.BMP, .DIB, .GIF, .JPG, .PICT, .PNG, .PNT, .PSD, .TGA, .TIFF, .WMF	.PAL .ACT	.AIFF .AU .MP3 .WAV	.AVI .MOV	.DIR .FLA .FLC .FLI .GIF .PPT	.BMP	.AVI .MOV	.DIR .DXR .EXE

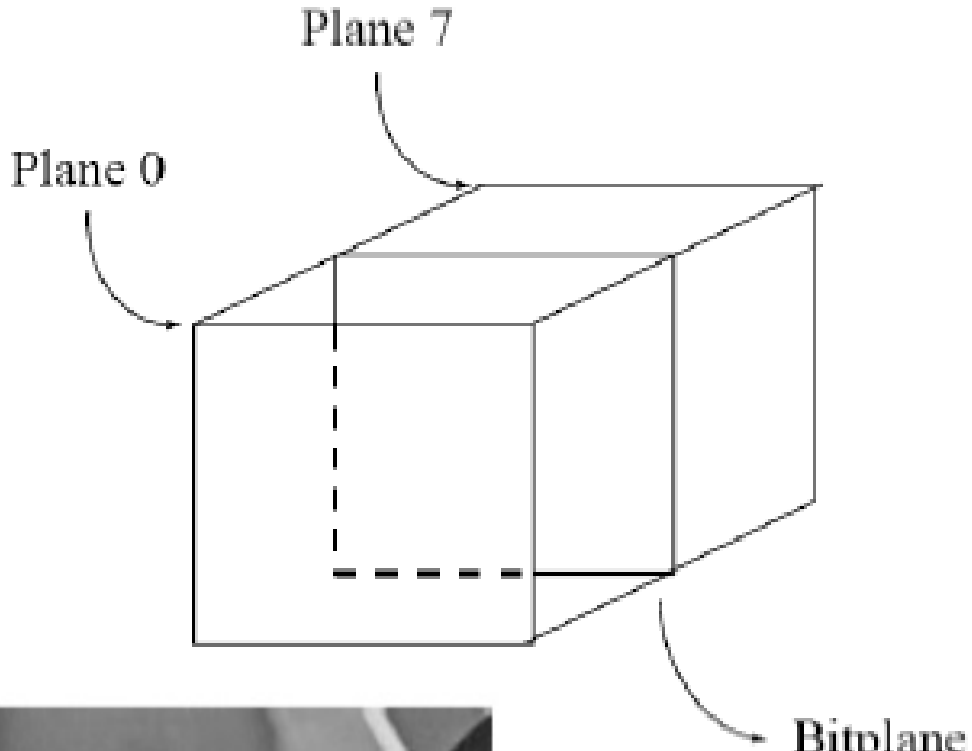
1-bit Images

- Pixels, or pels: picture elements
- Each pixel is stored as a single bit (0 or 1), so also referred to as binary image.
- Such an image is also called a 1-bit monochrome image since it contains no color.

8-bit Gray-level Images

- Each pixel has a gray-value between 0 and 255.
- Each pixel is represented by a single byte; e.g., a dark pixel might have a value of 10 and bright one might be 230.
- Bitmap: The two-dimensional array of pixel values that represents the graphics/image data.
- Image resolution refers to the number of pixels in a digital image (higher resolution always yields better quality).
- Fairly high resolution for such an image might be 1, 600 × 1, 200, whereas lower resolution might be 640 × 480.

- Frame buffer: Hardware used to store bitmap.
  - Video card (actually a graphics card) is used for this purpose.
- The resolution of the video card does not have to match the desired resolution of the image, but if not enough video card memory is available then the data has to be shifted around in RAM for display.
- 8-bit image can be thought of as a set of 1-bit bitplanes, where each plane consists of a 1-bit representation of the image at higher and higher levels of "elevation": a bit is turned on if the image pixel has a nonzero value that is at or above that bit level.



- Each pixel is usually stored as a byte (a value between 0 to 255), so a 640 × 480 grayscale image requires 300 kB of storage (640 × 480 = 307, 200).
- If we want to print such image, things become more complex.
- When an image is printed, the basic strategy of dithering is used, which trades intensity resolution for spatial resolution to provide ability to print multi-level images on 2-level (1-bit) printers.

Dithering

- Full-color photographs may contain an almost infinite range of color values. Dithering is the most common means of reducing the color range of images down to the 256 (or fewer) colors seen in 8-bit GIF images.
- For printing, Dithering is used to calculate larger patterns of dots such that values from 0 to 255 correspond to pleasing patterns that correctly represent darker and brighter pixel values.

- The main strategy is to replace a pixel value by a larger pattern say 2 × 2 or 4 × 4 such that the number of printed dots approximates the varying sized disks of ink used in analog, in halftone sized printing (e.g., for newspaper photos).
- Half-tone printing is an analog process that uses smaller or larger filled circles of black ink to represent shading, for newspaper printing.

- An algorithm for ordered dither, with $n \times n$ dither matrix, is as follows:

```
BEGIN
    for x = 0 to x_max                    // columns
        for y = 0 to y_max                // rows
            i = x mod n
            j = y mod n
            // I(x, y) is the input, O(x, y) is the output,
            //D is the dither matrix.
            if I(x, y) > D(i, j)
                O(x, y) = 1;
            else
                O(x, y) = 0;
END
```

Fig. (a) shows a grayscale image of "Lena". The ordered-dither version is shown as Fig. (b), with a detail of Lena's right eye in Fig. (c).

Image Data Types

- The most common data types for graphics and image file formats — 24-bit color and 8-bit color.
- Most image formats incorporate some variation of a compression technique due to the large storage size of image files.
- Compression techniques can be classified into either lossless or lossy.

24-bit Color Images

- In a color 24-bit image, each pixel is represented by three bytes, usually representing RGB.
- This format supports 256 × 256 × 256 possible combined colors, or a total of 16,777,216 possible colors.
- However such flexibility does result in a storage penalty: A 640 × 480 24-bit color image would require 921.6 kB of storage without any compression.
- An important point: many 24-bit color images are actually stored as 32-bit images, with the extra byte of data for each pixel used to store an alpha value representing special effect information (e.g., transparency).

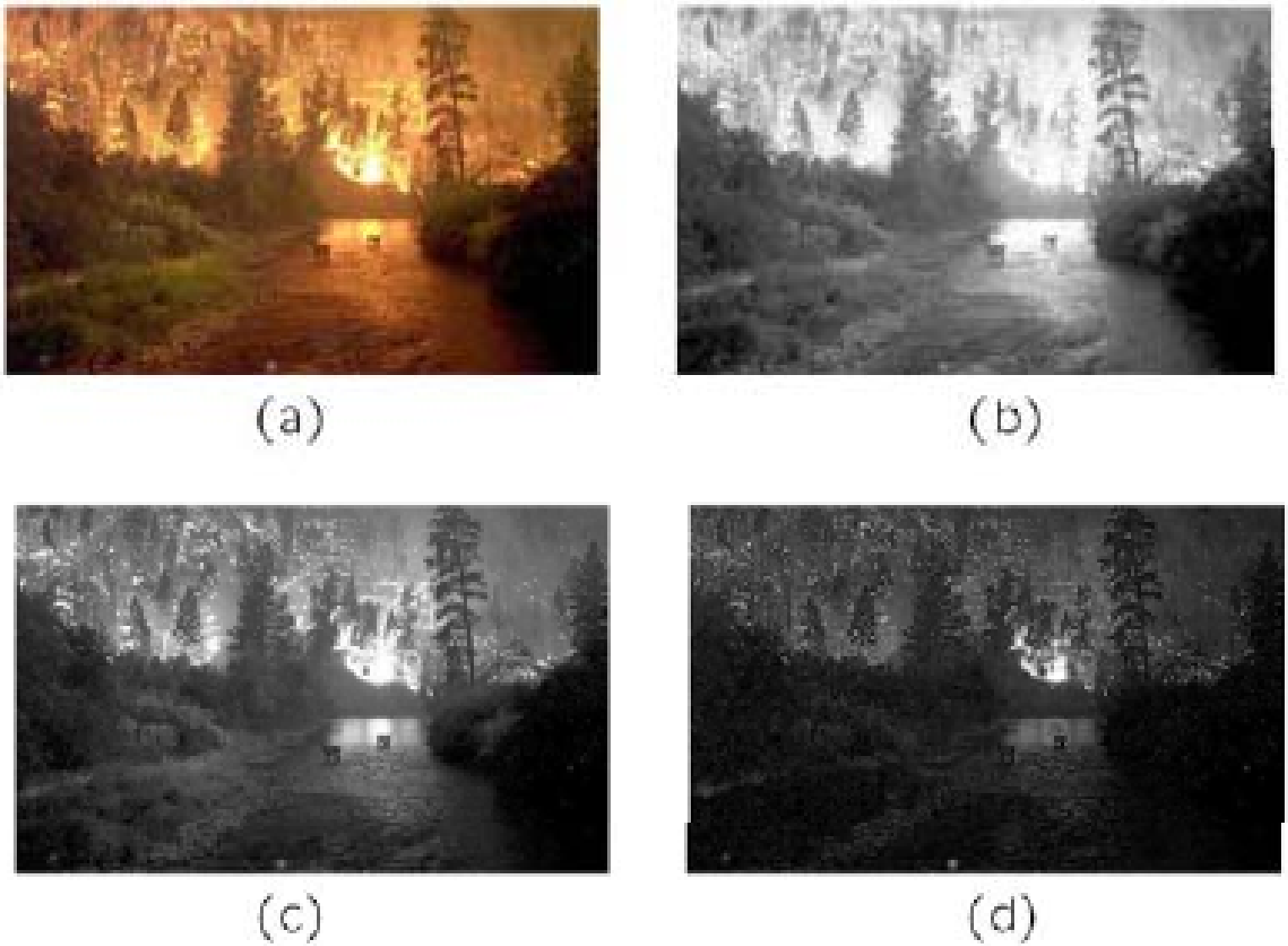


Fig. 3.5 High-resolution color and separate R, G, B color channel images. (a): Example of 24-bit color image "forestfire.bmp". (b, c, d): R, G, and B color channels for this image

8-bit Color Images

- Many systems can make use of 8 bits of color information (the so-called "256 colors") in producing a screen image.
- Such image files use the concept of a lookup table to store color information.
- Basically, the image stores not color, but instead just a set of bytes, each of which is actually an index into a table with 3-byte values that specify the color for a pixel with that lookup table index.

- An image histogram is a type of histogram which acts as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value.

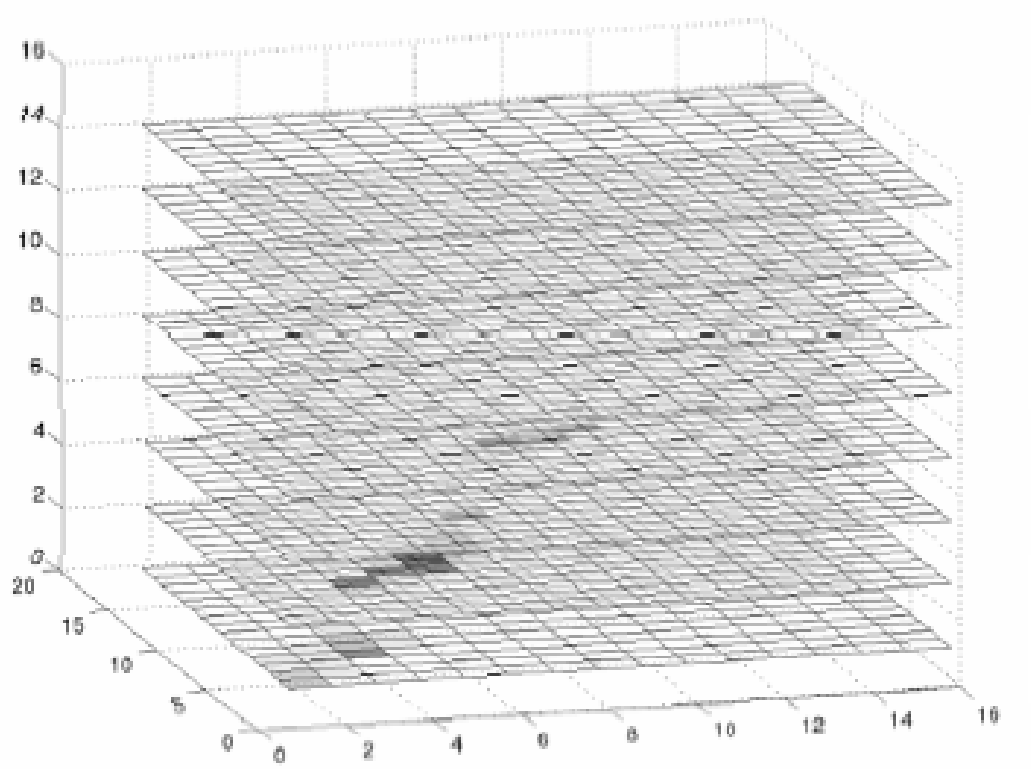
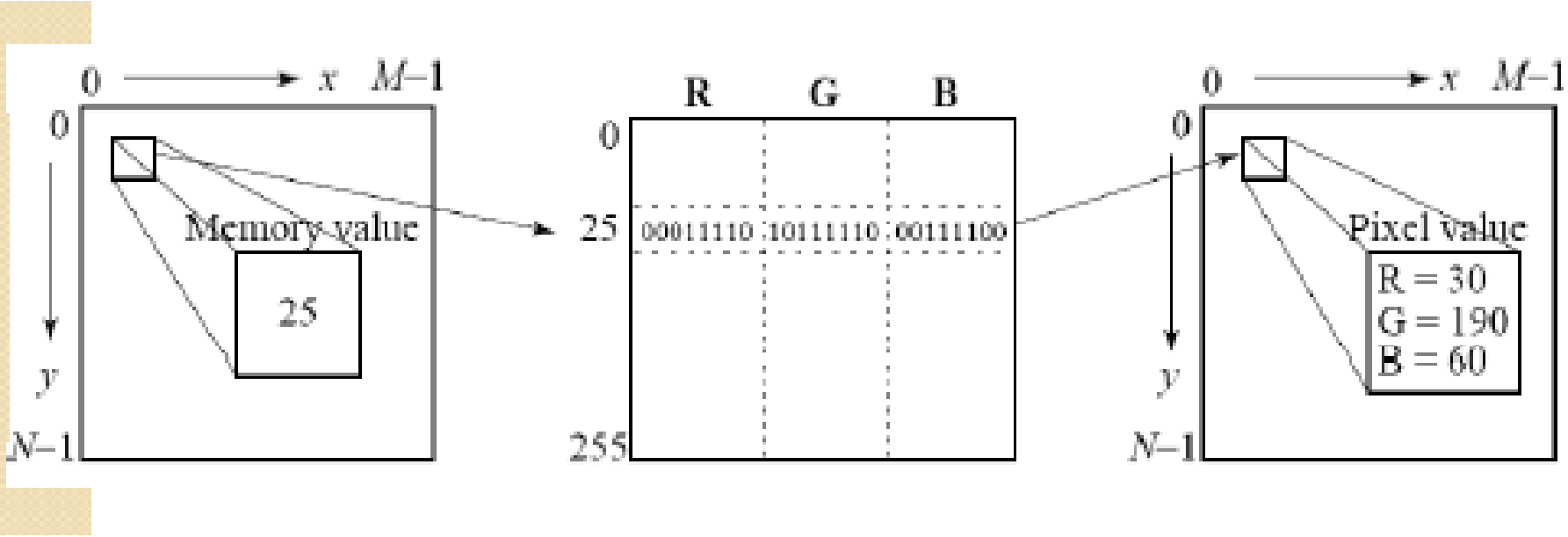


Fig. 3.6: 3-dimensional histogram of RGB colors in "forestre.bmp".

Fig. 3.7 shows the resulting 8-bit image, in GIF format.

Note the great savings in space for 8-bit images, over 24-bit ones: a 640 × 480 8-bit color image only requires 300 kB of storage, compared to 921.6 kB for a color image (again, without any compression applied).



Color Look Look-up Table (LUT)

Also called as Palette

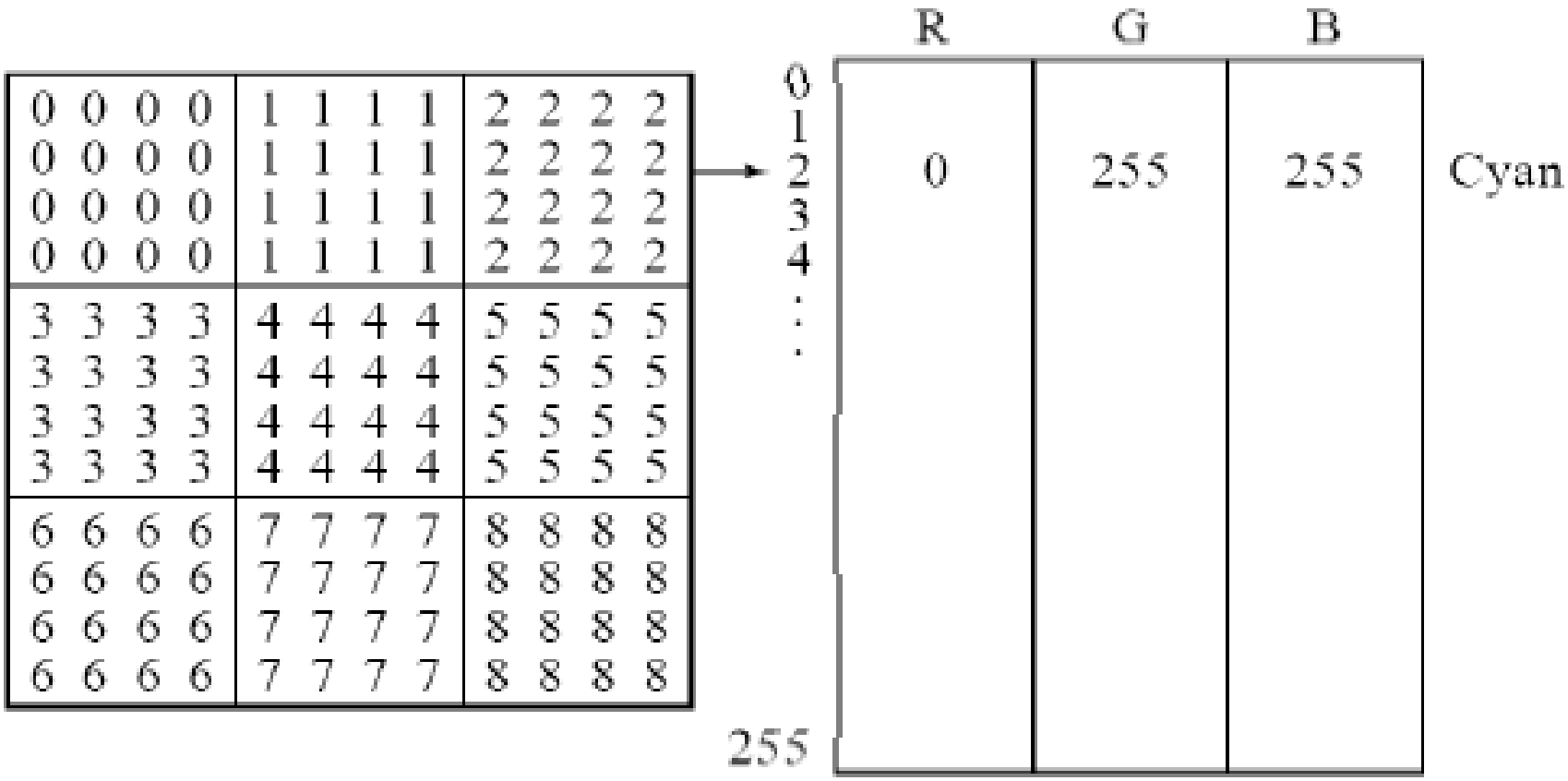


Fig. 3.9: Color-picker for 8-bit color: each block of the color-picker corresponds to one row of the color LUT

Fig. 3.10: (a): 24-bit color image "lena.bmp". (b):Version with color dithering. (c): Detail of dithered version.

How to Devise a Color Lookup Table

- The most straightforward way to make 8-bit look-up color out of 24-bit color would be to divide the RGB cube into equal slices in each dimension.
- The centers of each of the resulting cubes would serve as the entries in the color LUT, while simply scaling the RGB ranges 0..255 into the appropriate ranges would generate the 8-bit codes.
- Since humans are more sensitive to R and G than to B, we could shrink the R range and G range 0..255 into the 3 bit range 0..7 and shrink the B range down to the 2-bit range 0..3, thus making up a total of 8 bits.
- To shrink R and G, we could simply divide the R or G value by (256/8)=32 and then truncate. Then each pixel in the image gets replaced by its 8-bit index and the color LUT serves to generate 24-bit color.

Median -cut algorithm for Color Reduction Problem

- A simple alternate solution that does a better job for this color reduction problem.

a) The idea is to sort the R byte values and find their median; then values smaller than the median are labeled with a "0" bit and values larger than the median are labeled with a "1" bit.

b) This type of scheme will indeed concentrate bits where they most need to differentiate between high populations of close colors.

c) One can most easily visualize finding the median by using a histogram showing counts at position 0..255.

d) Fig. shows a histogram of the R byte values for the forestfire.bmp image along with the median of these values, shown as a vertical line.

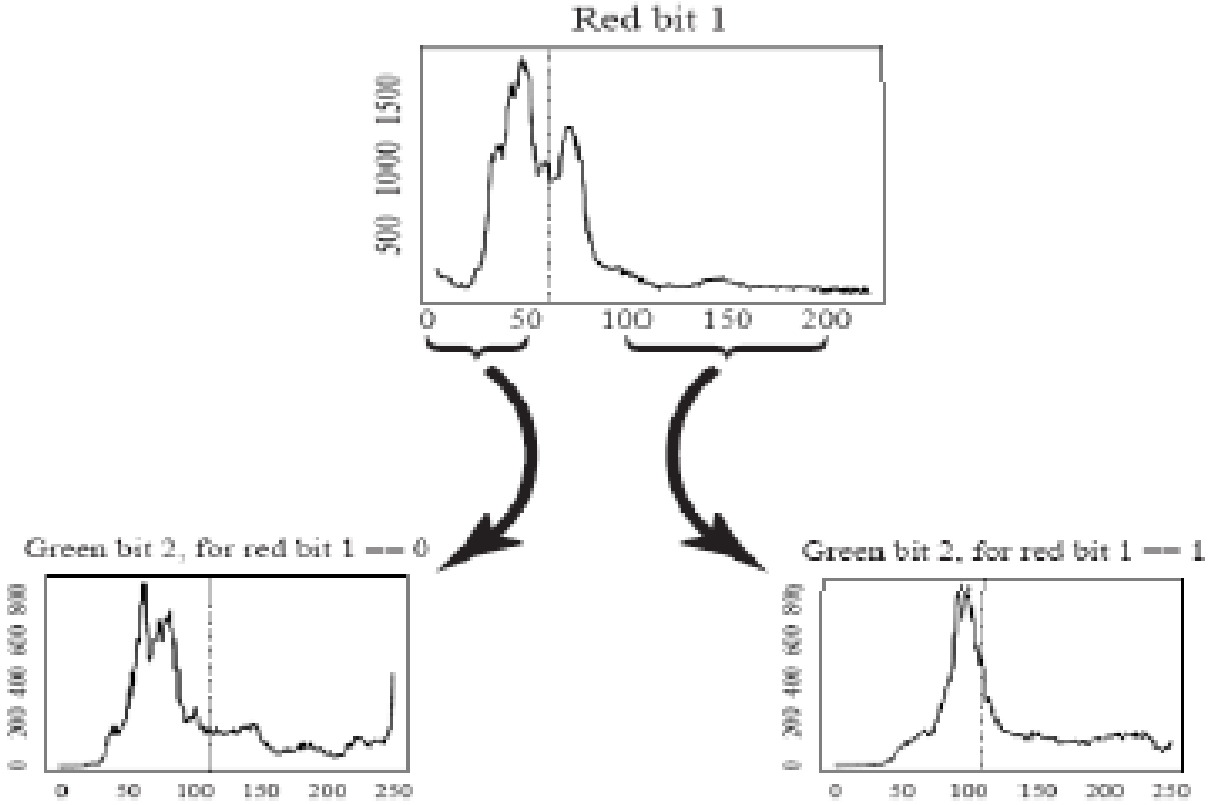


Fig. 3.11 Histogram of R bytes for the 24-bit color image "forestfire.bmp" results in a "0" bit or "1" bit label for every pixel. For the second bit of the color table index being built, we take R values less than the R median and label just those pixels as "0" or "1" according as their G value is less than or greater than the median of the G value, just for the "0" Red bit pixels. Continuing over R, G, B for 8 bits gives a color LUT 8-bit index

Median-Cut Algorithm

1. **Find the smallest box that contains all the colors in the image.**
2. **Sort the enclosed colors along the longest dimension of the box.**
3. **Split the box into two regions at the median of the sorted list.**
4. **Repeat that the above process in steps (2) and (3) until the original color space has been divided into, say, 256 regions.**

5 For every box call the mean of R G and B in that box the representative (the center) color for the box.

6. **Based on the Euclidean distance between a pixel RGB value and the box centers, assign every pixel to one of the representative colors. Replace the pixel by the code in a lookup table that indexes representative colors.**

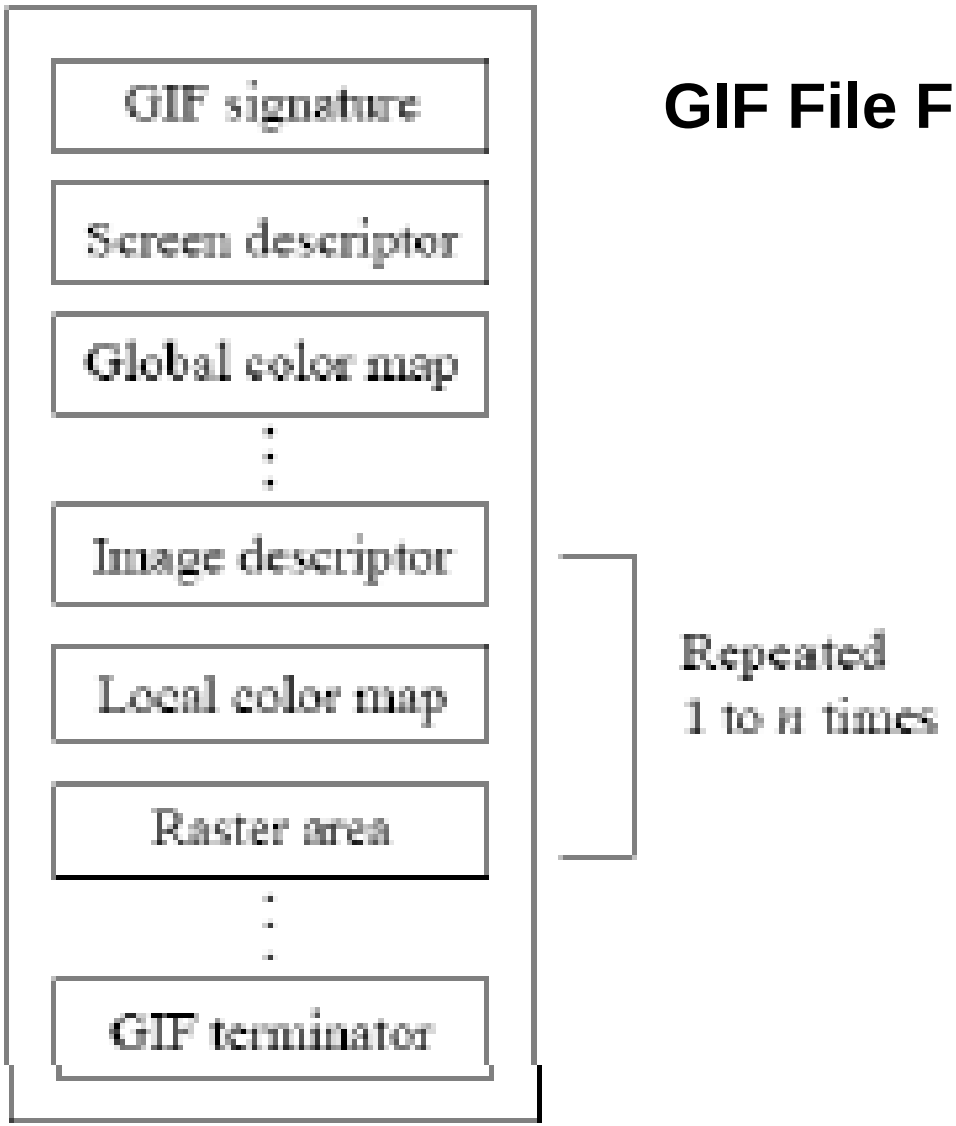
Popular File Formats 3

- GIF (Graphics Interchange Format)
- JPEG (Joint Photographic Experts Group)
- PNG (Portable Network Graphics)
- TIFF(Tagged Image File Format)
- EXIF (Exchange Image File)

GIF (Graphics Interchange Format)

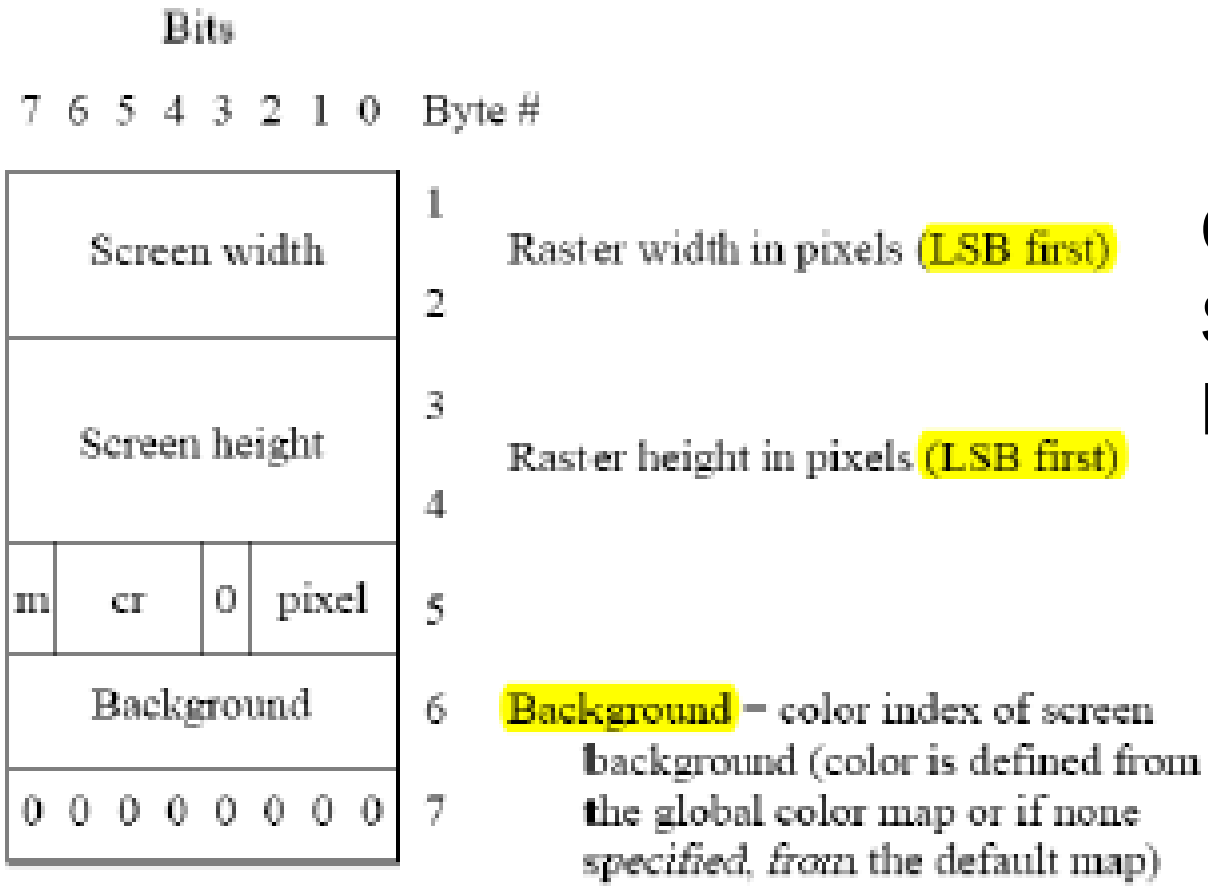
- GIF standard uses Lempel-Ziv-Welch algorithm.
- Limited to 8-bit (256) color images only, which, while producing acceptable color images, is best suited for images with few distinctive colors (e.g., graphics or Drawing).
- GIF standard supports interlacing — successive display of pixels in widely-spaced rows by a 4-pass display process.
- GIF images are of two types
  - GIF87a: The original specification .
  - GIF89a: The later version. Supports simple animation via a Graphics Control Extension block in the data, provides simple control over delay time a transparency index etc.

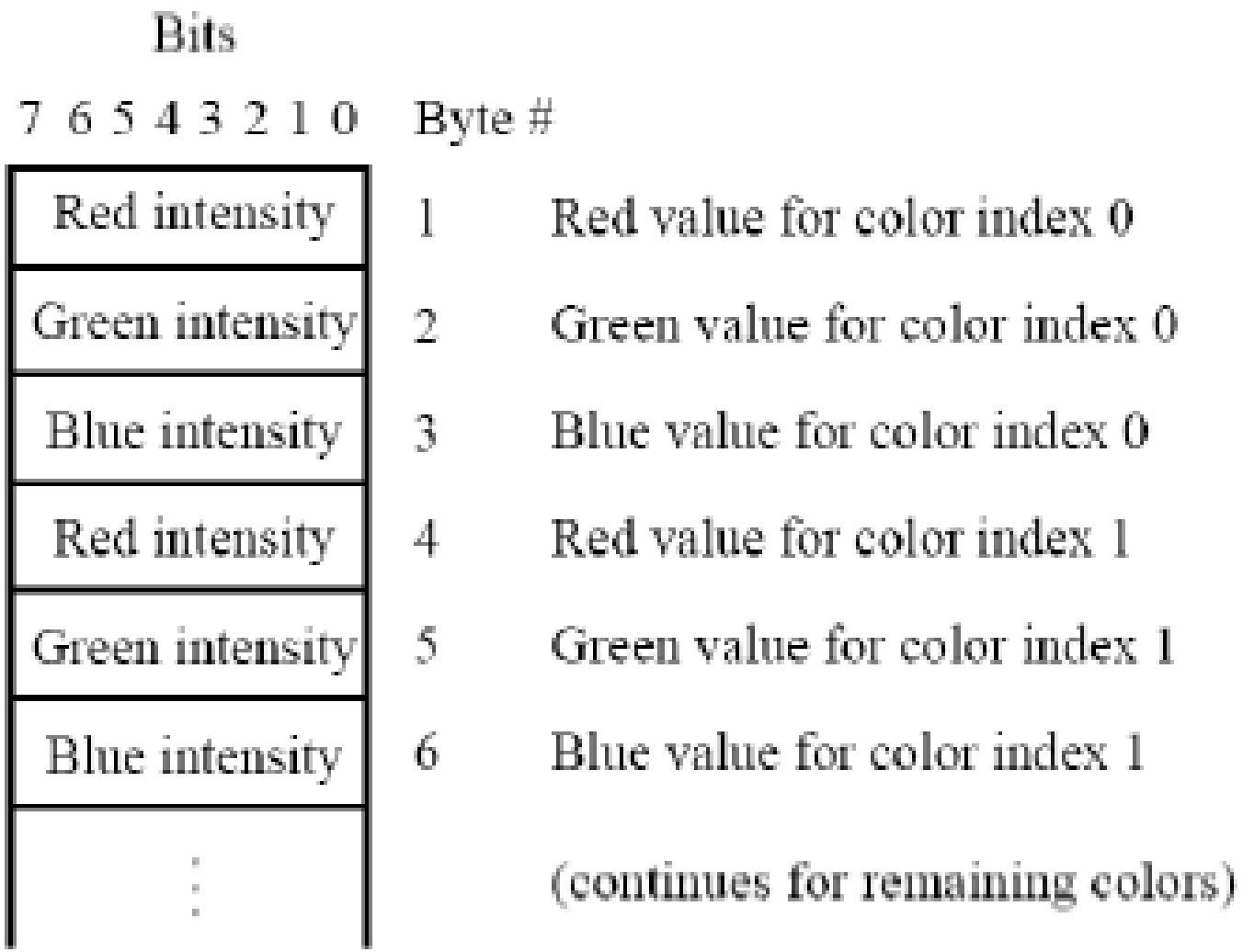
GIF87



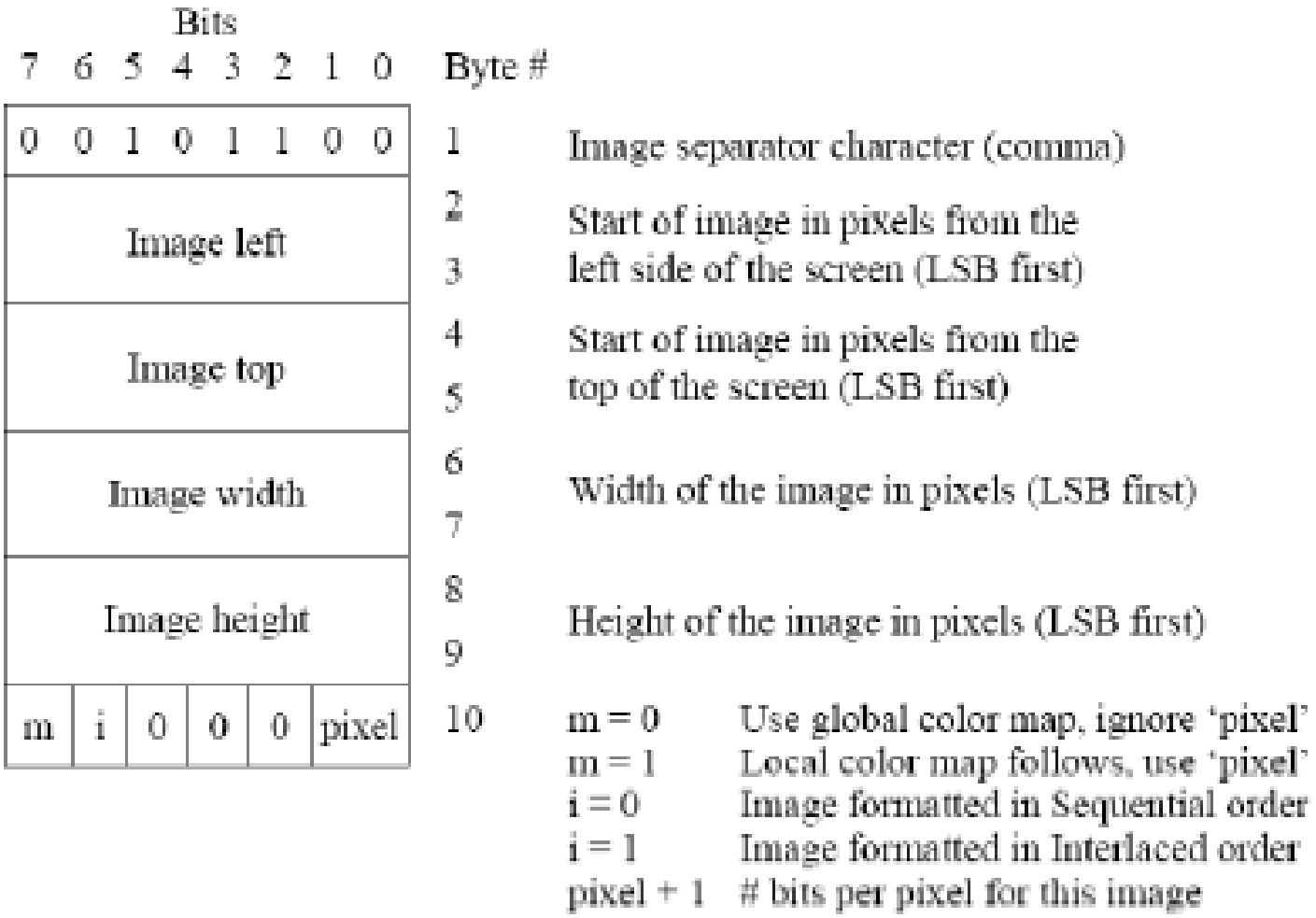
GIF File Format

GIF Screen Descriptor





GIF Color Map



GIF Image Descriptor

JPEG (Joint Photographic Expert Group)

- The most important current standard for image compression.
- The human vision system has some specific limitations and JPEG takes advantage of these to achieve high rates of compression.
- The eye-brain system cannot see extremely fine detail.
- JPEG allows the user to set a desired level of quality, or compression ratio (input divided by output).

JPEG image with low quality specified by user.

- As an example, Fig. shows forestfire image, with a quality factor Q=10%.
- This image is a mere 1.5% of the original size. In comparison, a JPEG image with Q=75% yields an image size 5.6% of the original, whereas a GIF version of this image compresses down to 23.0% of uncompressed image size.

PNG (Portable Network Graphics)

- PNG meant to supersede the GIF standard, and extends it in important ways.
- Special features of PNG files include support for 48 bits of color information.
- Files may contain gamma-correction information for correct display images alpha channel of color images, as well as alpha-information for such uses as control of transparency.
- The display progressively displays pixels in a 2-dimensional fashion by showing a few pixels at a time over seven passes through each 8 × 8 block of an image.

TIFF(Tagged Image File Format)

- TIFF is another popular image file format, developed by the Aldus Corporation in the 1980's and was later supported by Microsoft .
- Its support for attachment of additional information (referred to as "tags") provides a great deal of flexibility.
- The most important tag is a format signifier: what type of compression etc is in use in the stored image etc. image.
- TIFF can store many different types of image: 1-bit, grayscale , 8-bit color, 24-bit RGB, etc.
- TIFF was originally a lossless format but now a new JPEG tag allows one to opt for JPEG compression.

EXIF (Exchange Image File)

- EXIF is an image format for digital cameras:

1. Compressed EXIF files use the baseline JPEG format.
2. A variety of tags (many more than in TIFF) are available to facilitate higher quality printing, since information about the camera and picture-taking picture conditions (flash, exposure, light source, white balance, type of scene, etc.) can be stored and used by printers for possible color correction algorithms.
3. The EXIF standard also includes specification of file format for audio that accompanies digital images. As well, it also supports tags for information needed for conversion to FlashPix (initially developed by Kodak).

Graphics Animation Files

- A few format are aimed at storing graphics animation(series of drawings/graphics illustrations) as opposed to video(series of images).
- FLC is an animation or moving picture file format; it was originally created by Animation Pro. Another format, FLI, is similar to FLC.
- GL produces somewhat better quality moving pictures. GL animations can also usually handle larger file sizes sizes.
- Many older formats: such as DL or Amiga IFF files, Apple Quicktime files, as well as animated GIF89 files.

PS (PostScript) & PDF (Portable Document Format)

- Postscript is an important language for typesetting, and many high-end printers have a Postscript interpreter built into them.
- PS is a vector-based picture language, rather than pixel-based: page element definitions are essentially in terms of vectors.
- PS includes text as well as vector/structured graphics, bit-mapped images can be included in output files.
- Encapsulated PS files add some additional information for inclusion of Postscript files in another document.

- Postscript page description language itself does not provide compression; in fact, Postscript files are just stored as ASCII.
- Another text + figures language has begun to supersede or at least parallel Postscript: Adobe Systems Inc. includes LZW compression in its Portable Document Format (PDF) file format.
- PDF files that do not include images have about the same compression ratio, 2:1 or 3:1, as do files compressed with other LZW-based compression tools.
- For files containing images PDF may achieve higher compression ratio by using JPEG compression for the image content.

Other Formats

- **Windows WMF :-** Windows MetaFile (WMF) is the native vector file format fir MS Windows operating environment.
- **Windows BMP** :- BitMap (BMP) is the major system standard graphics file format for MS Windows used in Paint & other programs.
- **Macintosh PAINT & PICT :-** PAINT was originally used in the MacPaint program, initially only for 1-bit monochrome images. PICT format is used in MacDraw (a vector-based drawing program) for storing structured graphics.
- **X Windows PPM (Portable PixMap) :-** the graphics format for the X Window system. PPM supports 24-bit color
- bitmaps, and can be manipulated using many public domain graphic editors.

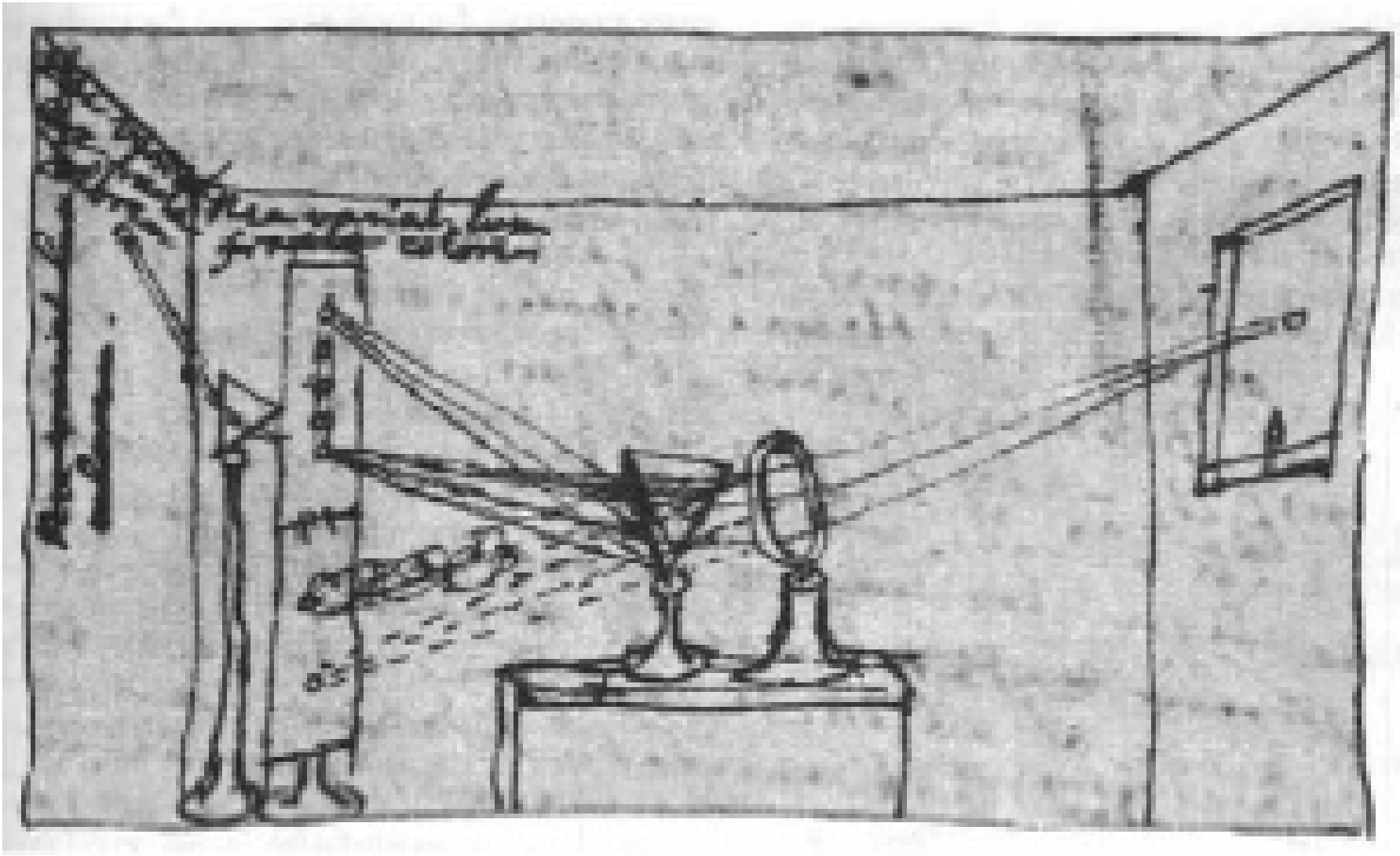
Color in Image and Video

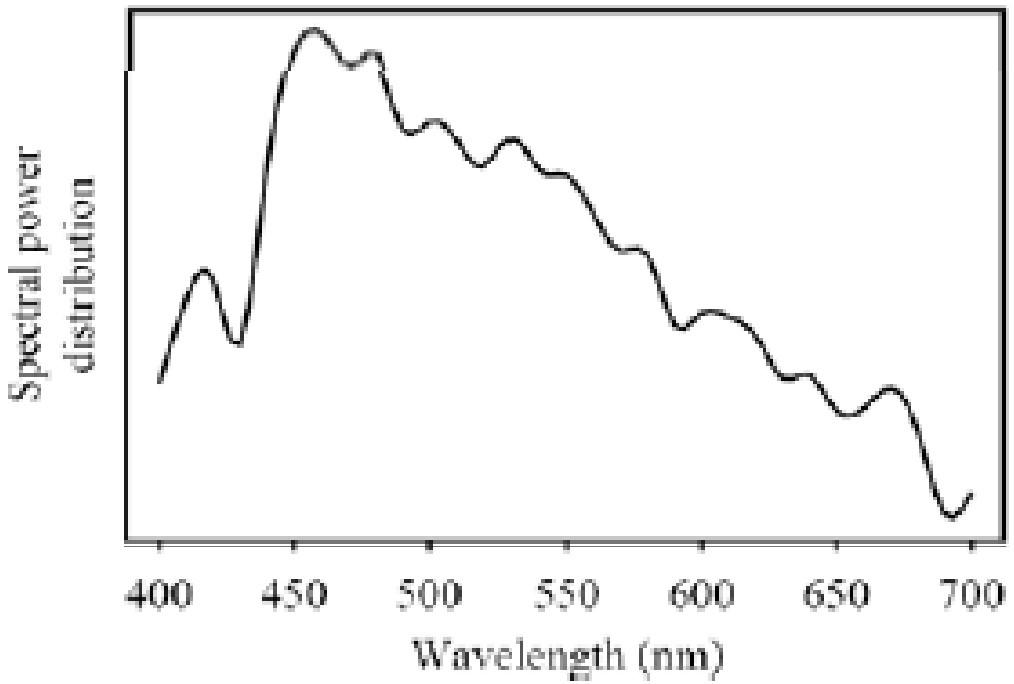
- Color images and videos are everywhere on the web and in multimedia production.
- Also we know that there is discrepancies between the color as seen by the people and displayed on the screens.
- Here we will study following topics.
  - Color Science.
  - Color Models in Images.
  - Color models in Videos.

Color Science

- Light and Spectra :- Light is an electromagnetic wave. Its color is characterized by the wavelength content of the light.
- Laser light consists of a single wavelength: e.g., a ruby laser produces a bright, scarlet-red beam.
- Most light sources produce contributions over many wavelengths.
- However, humans cannot detect all light, just contributions that fall in the "visible wavelengths".
- Short wavelengths produce a blue sensation, long wavelengths produce a red one.

- Spectrophotometer: device used to measure visible light, by reflecting light from a diffraction grating (a ruled surface) that spreads out the different wavelengths.
- Visible light is an electromagnetic wave in the 400 nm to 700 nm.





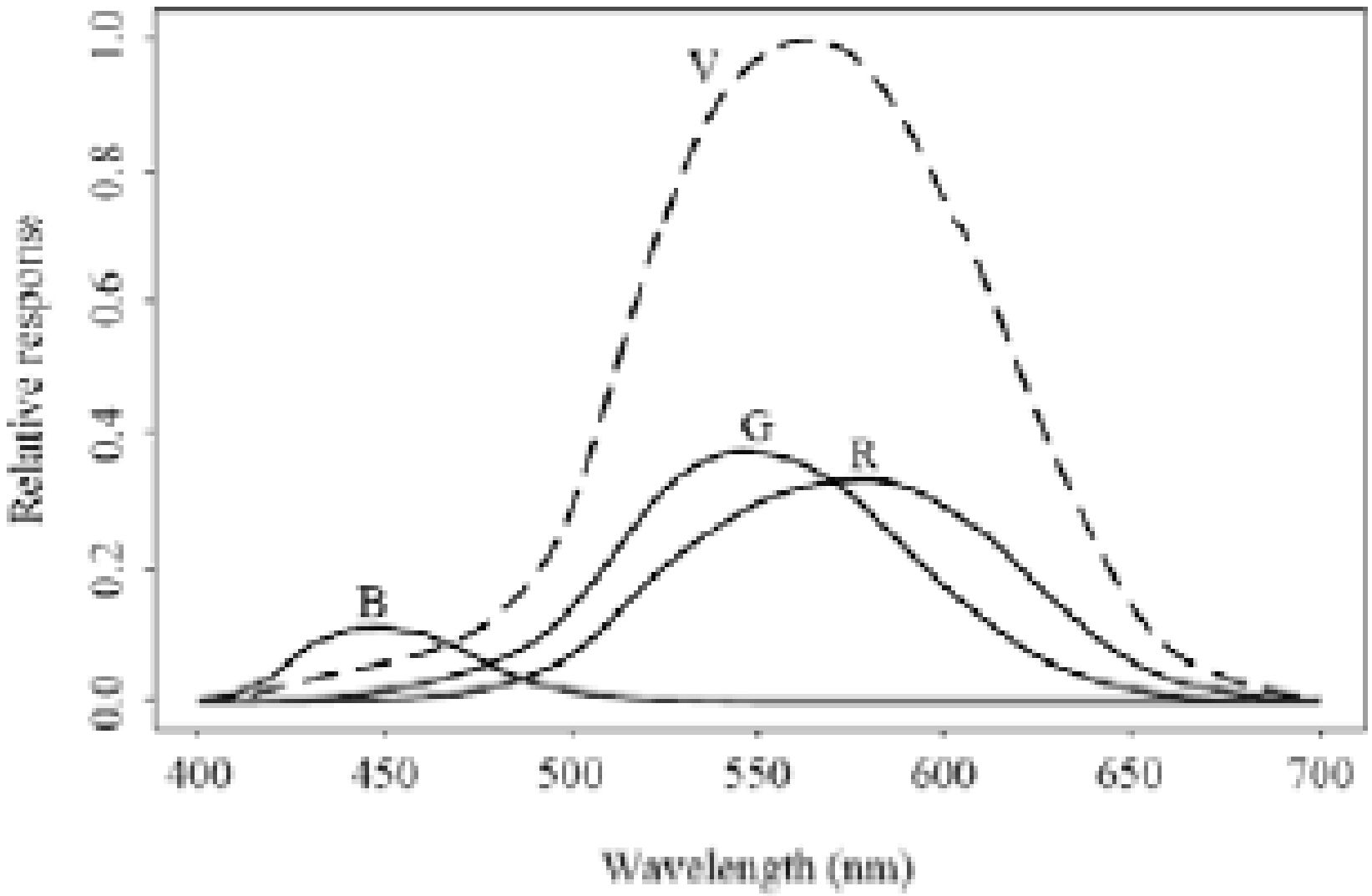
- Fig shows the relative power in each wavelength interval for typical outdoor light on a sunny day.
- This type of curve is called a Spectral Power Distribution (SPD) or a spectrum.
- The symbol for wavelength is λ. This curve is called E(λ).

Human Vision

- Eye works like Camera.
- Retina consists of array of rods (for low light levels and three kinds of cones (for higher light levels).
- The brain makes use of differences R-G, G-B, and B-R, as well as combining all of R, G and B into a high-light-level achromatic channel.

Spectral Sensitivity of the Eye

- The eye is most sensitive to light in the middle of the visible spectrum.
- Fig. shows the overall sensitivity as a dashed line this important curve is called the luminous-efficiency function.
- It is usually denoted V (λ) and is formed as the sum of the response curves for Red, Green, and Blue.



- The eye has about 6 million cones, but the proportions of R, G and B cones are different
- They likely are present in the ratios 40:20:1
- So the achromatic(without color) channel produced by the cones is approximately proportional to

  $2R + G + B/20.$
- These spectral sensitivity functions are usually denoted by letters other than "R, G, B“.
- We use a vector function q(λ), with components

  $q(\lambda)=[q_R(\lambda),q_G(\lambda),q_B(\lambda)]^T$

- The response in each color channel in the eye is proportional to the number of neurons firing.
- A laser light at wavelength λ would result in a certain number of neurons firing. An SPD (spectral power distribution) is a combination of single-frequency lights (like lasers), so we add up the cone responses for all wavelengths, weighted by the eye's relative response at that wavelength.

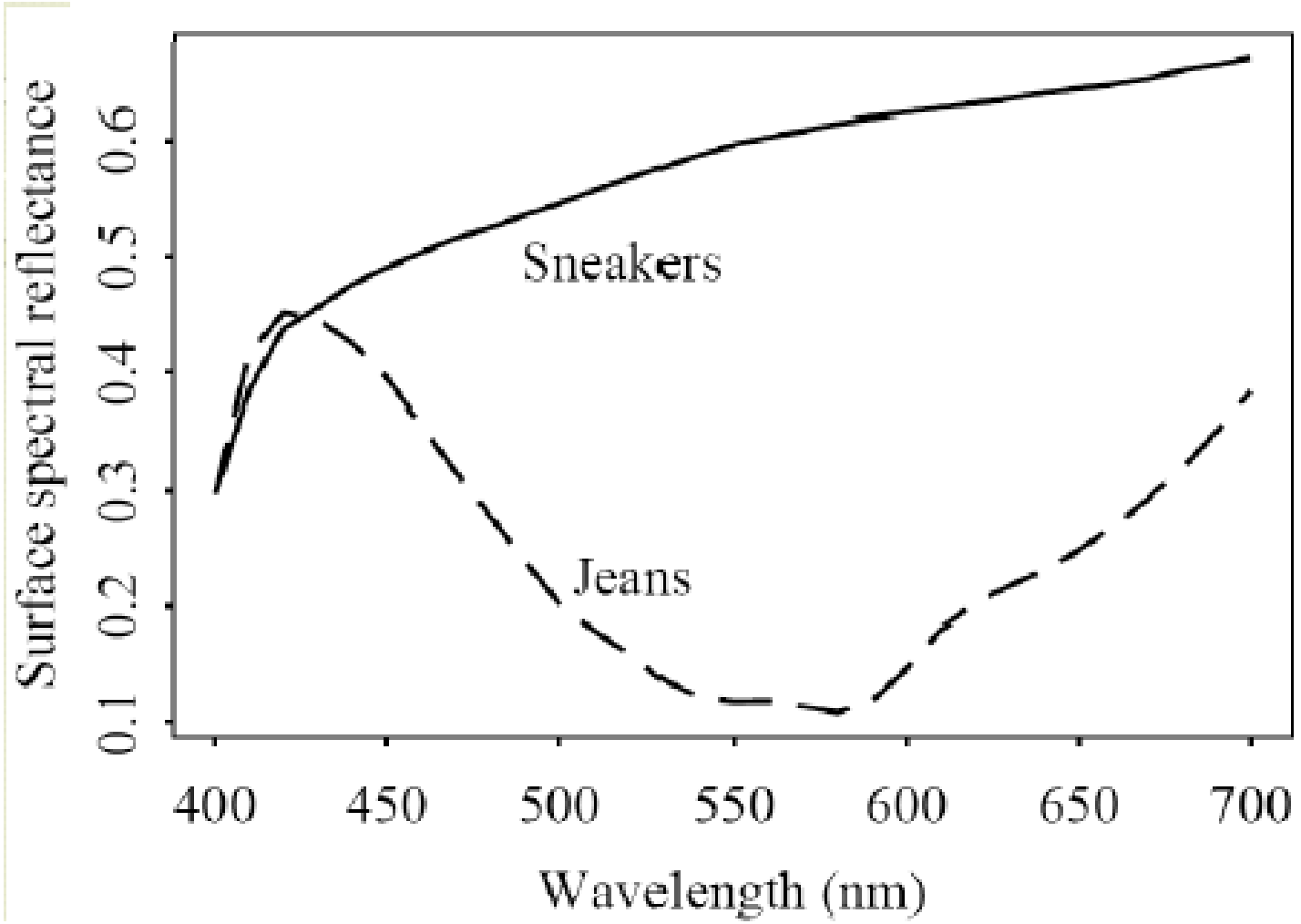
$$R = \int E(\lambda)\, q_R(\lambda)\, d\lambda$$

$$G = \int E(\lambda)\, q_G(\lambda)\, d\lambda$$

$$B = \int E(\lambda)\, q_B(\lambda)\, d\lambda$$

Image Formation

- The above equations applies only when we view a self-luminous object.
- In many situations, we image light reflected from a surface.
- Surfaces reflect different amounts of light at different wavelengths, and dark surfaces reflect less energy than light surfaces.



The surface spectral reflectance from (1) orange sneakers and (2) faded bluejeans .

The reflectance function is denoted S(λ).

Image formation is thus:

- Light from the illuminant with SPD E(λ) impinges on a surface, with surface spectral reflectance function S(λ), is reflected, and then is filtered by the eye's cone functions q (λ).
- Reflection is shown in Fig. below.
- The function C(λ) is called the color signal and consists of the product of E(λ), the p (), illuminant, times S(λ), the reflectance:

$$C(\lambda) = E(\lambda)\, S(\lambda).$$

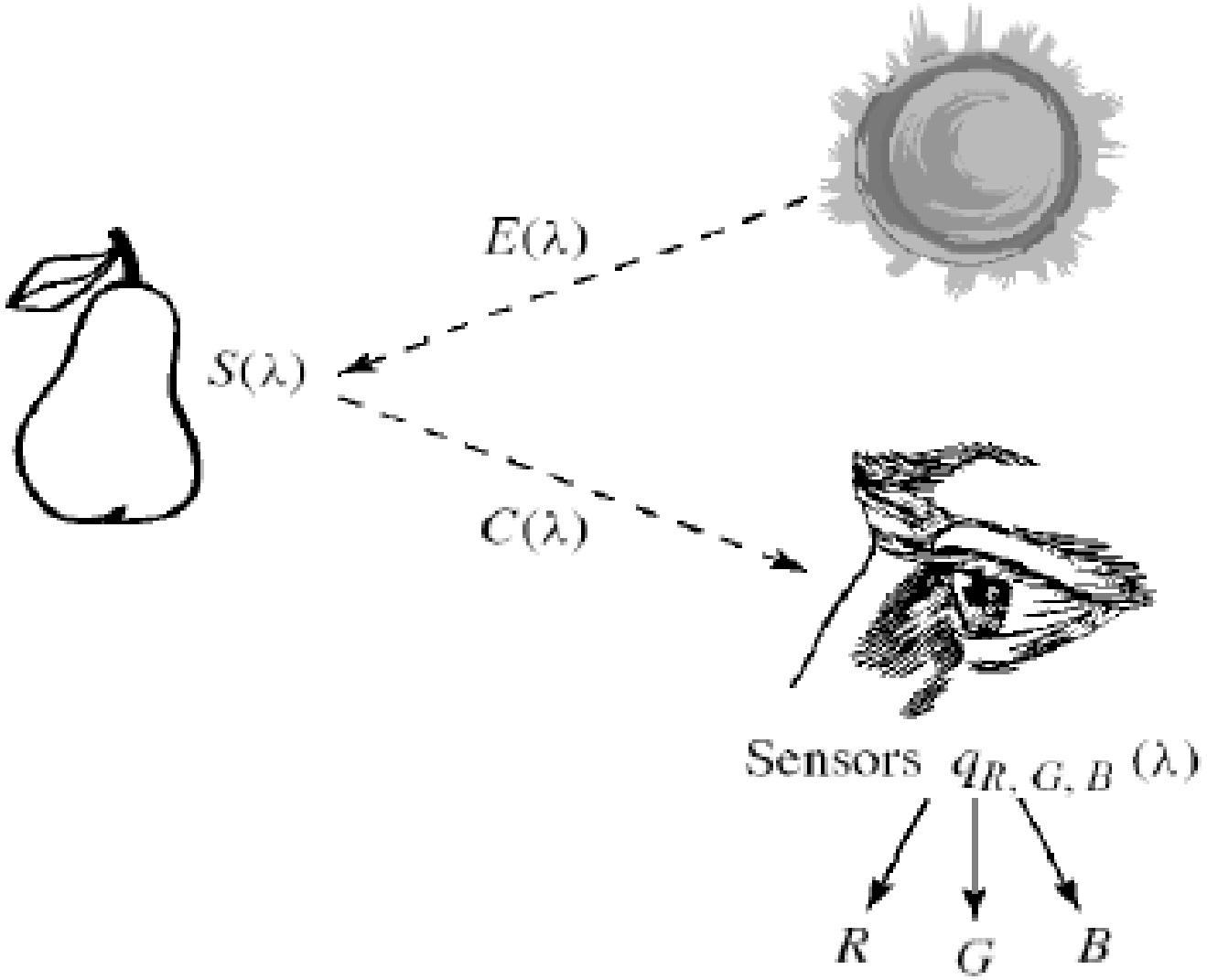


Fig. 4.5: Image formation model

- The equations that take into account the image formation model are:

$$R = \int E(\lambda)\, S(\lambda)\, q_R(\lambda)\, d\lambda$$

$$G = \int E(\lambda)\, S(\lambda)\, q_G(\lambda)\, d\lambda$$

$$B = \int E(\lambda)\, S(\lambda)\, q_B(\lambda)\, d\lambda \qquad (4$$

Camera Systems

Gamma Correction

- The light emitted is in fact roughly proportional to the voltage raised to a power; this power is called gamma, with symbol γ.

a) Thus, if the file value in the red channel is R, the screen emits light proportional to Rγ, with SPD equal to that of the red phosphor paint on the screen that is the target of the red channel electron gun. The value of gamma is around 2.2.

b) It is customary to append a prime to signals that are gamma corrected by raising to the gamma-power (1/γ) before transmission. Thus we arrive at linear signals:

$$R \rightarrow R' = R^{1/\gamma} \Rightarrow (R')^{\gamma} \rightarrow R$$

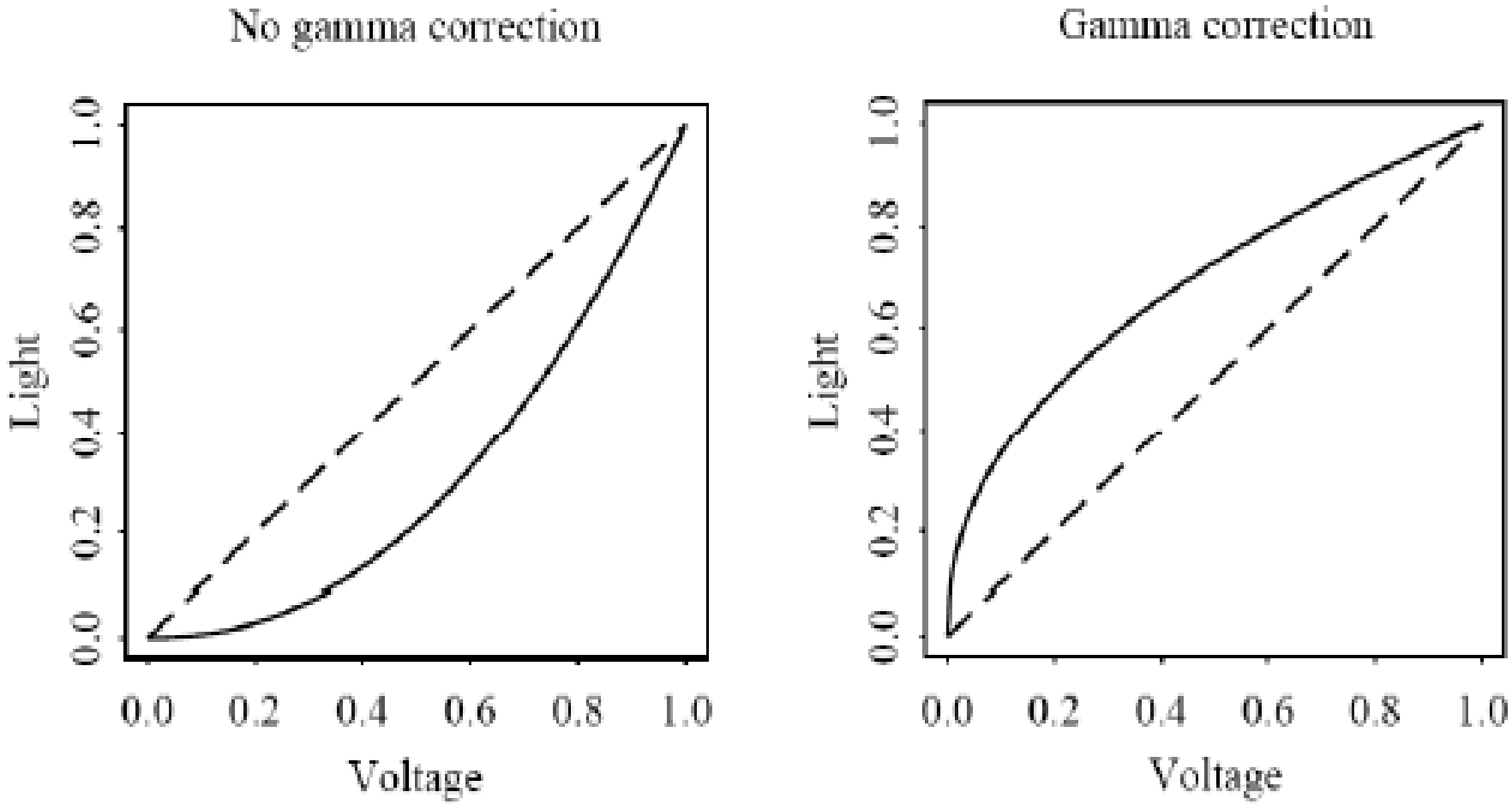


Fig. 4.6: (a): Effect of CRT on light emitted from screen (voltage is normalized to range 0..1). (b): Gamma correction of signal.

- Fig 4 6(a) shows light output with gamma-correction applied. We see that darker values are displayed too dark. This is also shown in Fig. 4.7(a), which displays a linear ramp from left to right.
- Fig 4 6(b) the effect of pre-correcting signals by applying the power law $R^{1/\gamma}$; it is customary to normalize voltage to the range [0,1].

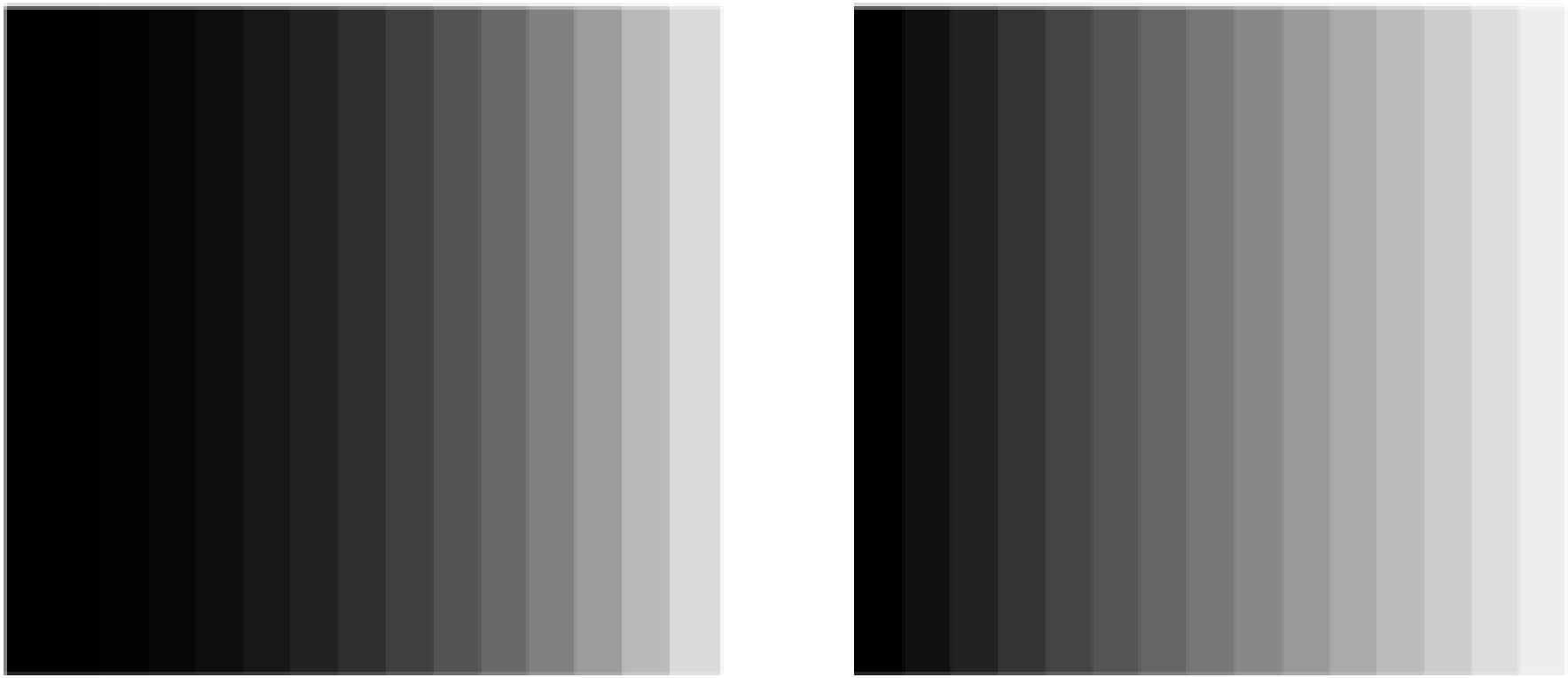


Fig. 4.7: (a): Display of ramp from 0 to 255, with no gamma correction. (b): Image with gamma correction applied

Color-Matching Functions

- Even without knowing the eye-sensitivity curves, a technique evolved in psychology for matching a combination of basic R, G, and B lights to given shade.
- The particular set of three basic lights used in an experiment are called the set of color primaries.
- To match a given color, a subject is asked to separately adjust the brightness of the three primaries using a set of controls until the resulting spot of light most closely matches the desired color.
- The basic situation is shown in Fig.

A device for carrying out such an experiment is called a colorimeter.

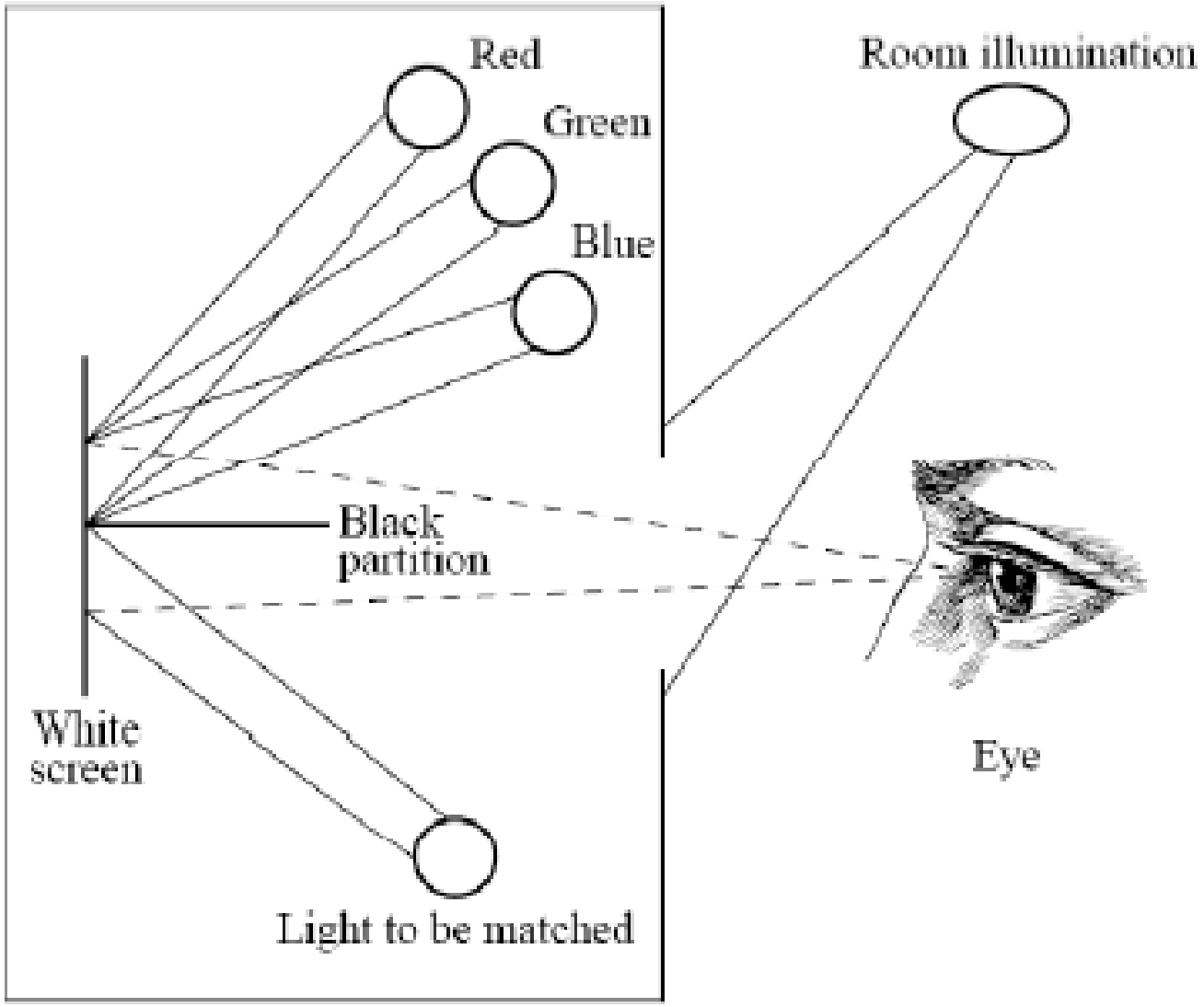


Fig. 4.8: Colorimeter experiment.

Color Monitor Specifications

- Color monitors are specified in part by the white point chromaticity that is desired if the RGB electron guns are all activated at their highest value (1.0, if we normalize to [0,1]).
- We want the monitor to display a specified white when R'=G'=B'=1.
- There are several monitor specifications in current use (Table 4.1).

Table 4.1: Chromaticities and White Points of Monitor Specifications

System	Red		Green		Blue		White Point	
	x_r	y_r	x_g	y_g	x_b	y_b	x_W	y_W
NTSC	0.67	0.33	0.21	0.71	0.14	0.08	0.3101	0.3162
SMPTE	0.630	0.340	0.310	0.595	0.155	0.070	0.3127	0.3291
EBU	0.64	0.33	0.29	0.60	0.15	0.06	0.3127	0.3291

Color Models in Images

- **RGB Color Model for CRT Displays**

1. We expect to be able to use bits per color channel for color that is accurate enough.
2. However, in fact we have to use about 12 bits per channel to avoid an aliasing effect in dark image areas —contour bands that result from gamma correction.
3. For images produced from computer graphics, we store integers proportional to intensity in the frame buffer. So should have a gamma correction LUT between the frame buffer and the CRT.
4. If gamma correction is applied to floats before quantizing to integers, before storage in the frame buffer, then in fact we can use only 8 bits per channel and still avoid contouring artifacts.

Subtractive Color: CMY Color Model

- So far we have effectively been dealing, only with additive color. Namely, when two light beams impinge on a target, their colors add; when two phosphors on a CRT screen are turned on, their colors add.
- But for ink deposited on paper, the opposite situation holds: yellow ink subtracts blue from white illumination, but reflects red and green; it appears yellow.
- Instead of red, green, and blue primaries, we need primaries that amount to -red, -green, -blue. I.e., we need to subtract R, or G, or B.
- These subtractive color primaries are Cyan (C), Magenta (M) and Yellow (Y)inks.

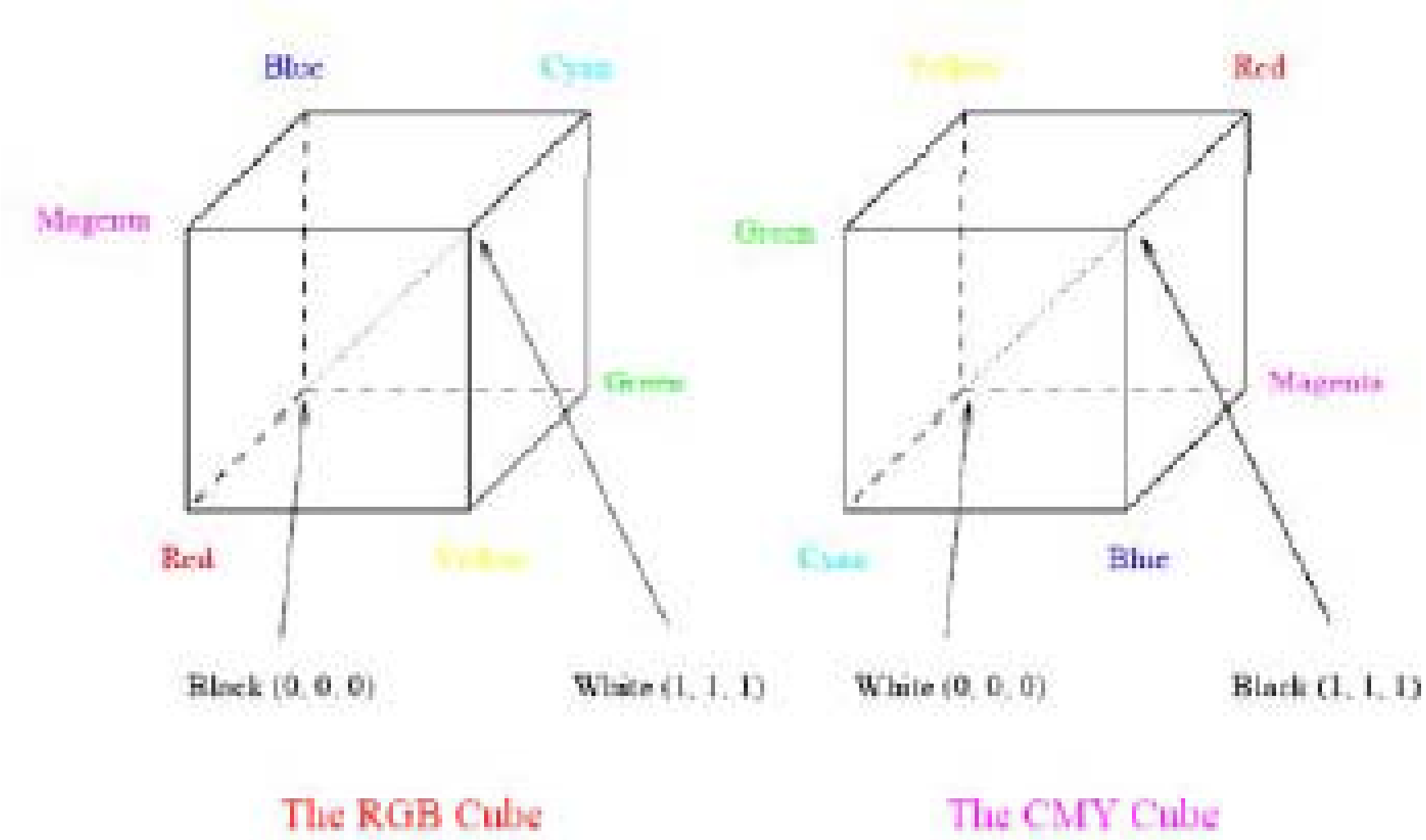


Fig. 4.15: RGB and CMY color cubes.

Transformation from RGB to CMY

- Simplest model we can invent to specify what ink density to lay down on paper, to make a certain desired RGB color.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- Then the inverse transform is:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

color combinations that result from combining primary colors available in the two situations, additive color and subtractive color.

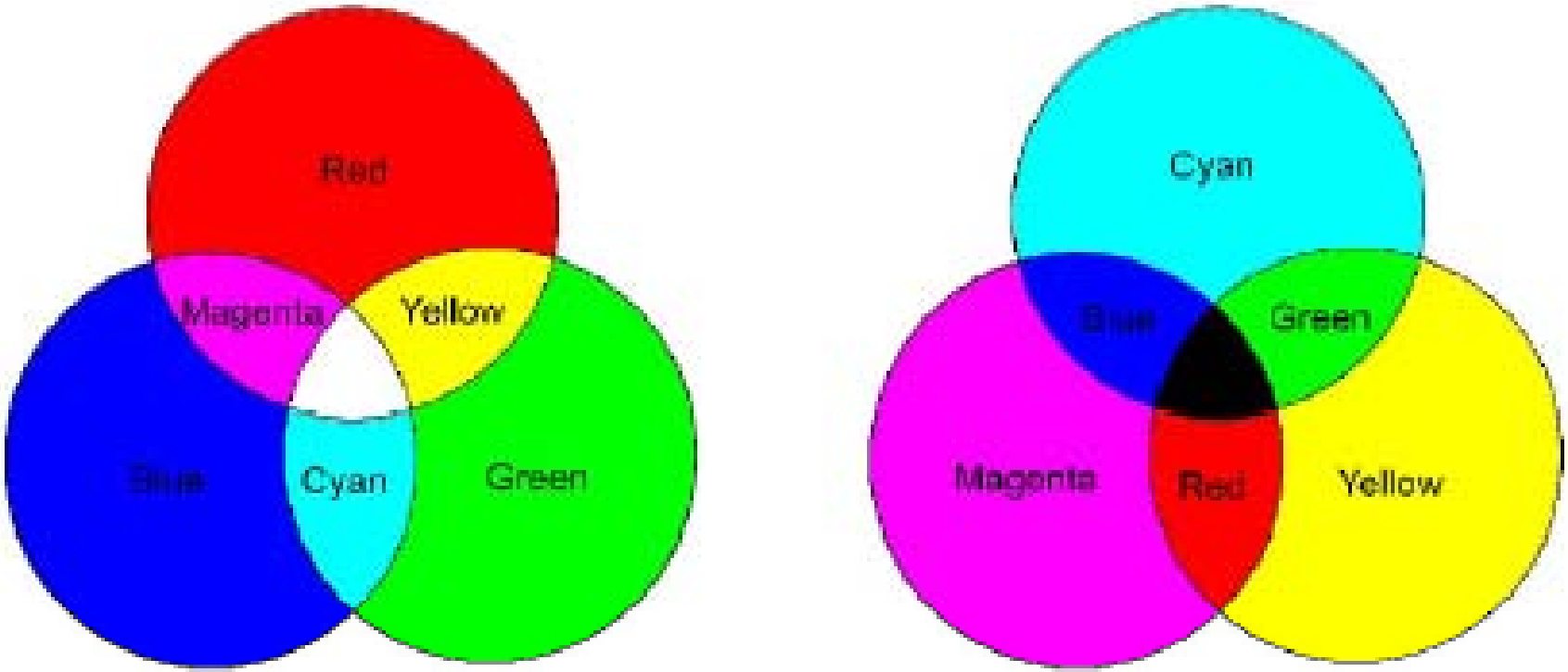


Fig. 4.16: Additive and subtractive color. (a): RGB is used to specify additive color. (b): CMY is used to specify subtractive color

Printer Gamuts

- Actual transmission curves overlap for the C, M, Y inks. This leads to "crosstalk“ between the color channels and difficulties in predicting colors achievable in printing.
- Fig (a) shows typical transmission for real "block dyes", and Fig.(b) shows the resulting color gamut for a color printer.

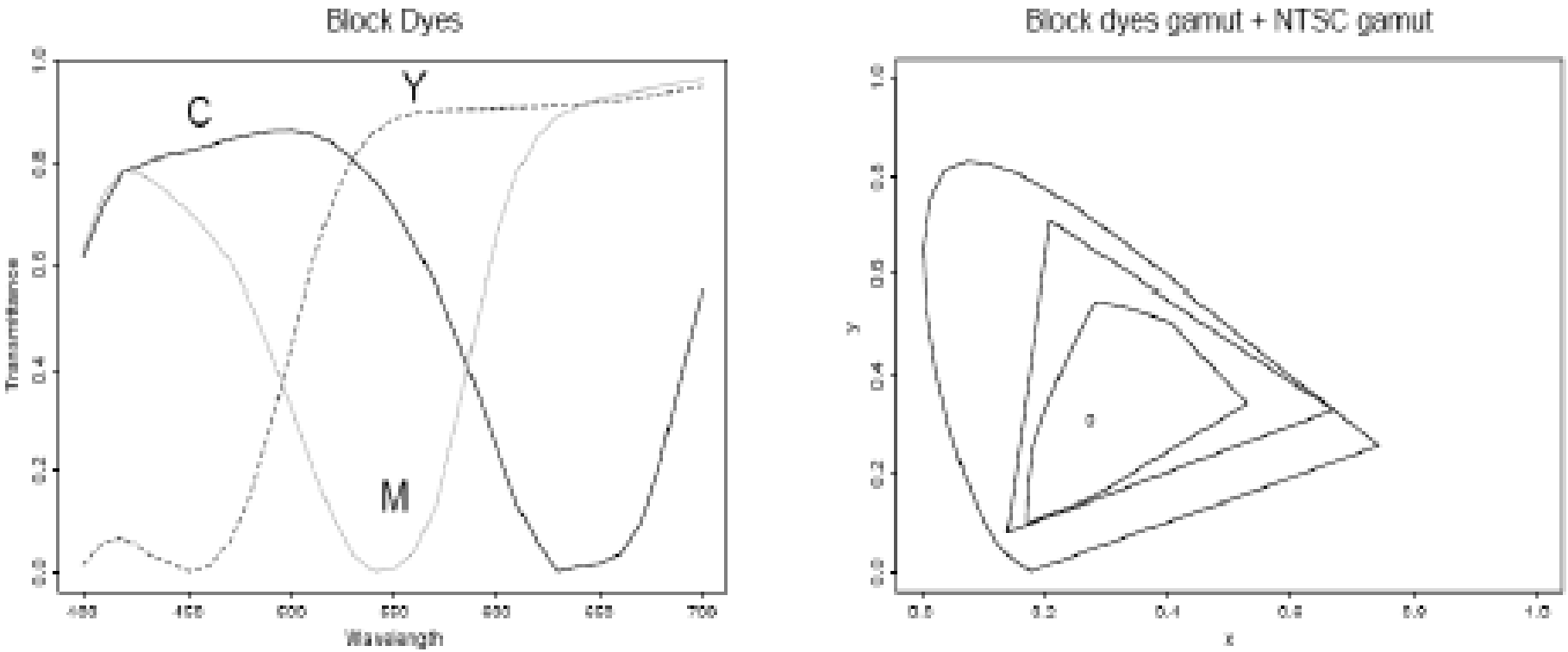


Fig. 4.17: (a): Transmission curves for block dyes. (b): Spectrum locus, triangular NTSC gamut, and 6-vertex printer gamut

Color Models in Video

Video Color Transforms

- Methods of dealing with color in digital video derive largely from older analog methods of coding color for TV. Luminance is separated from color information.
- For example, a matrix transform method called YIQ is used to transmit TV signals in North America and Japan.
- This coding also makes its way into VHS video tape coding in these countries since video tape technologies also use YIQ.
- In Europe, video tape uses the PAL or SECAM codings, which are based on TV that uses a matrix transform called YUV.
- Digital video mostly uses a matrix transform called YCbCr that is closely related to YUV.

YUV Color Model

- YUV codes a luminance signal (for gamma-corrected signals) equal to Y’ in Eq. (4.20). the "luma".
- Chrominance refers to the difference between a color and a reference white at the same luminance → use color differences U, V :
- U = B’–Y’ V = R’–Y’ (4.27)
- From Eq. (4.20) [Y’ =0.299·R’+0.587·G’+0.114·B’] & eq (4.27)

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} \quad (4.28)$$

- For gray, R’ = G’ = B’ , the luminance Y’ equals to that gray, since 0.299 + 0.587 + 0.114 = 1.0. And for a gray ("black & white") image the chrominance (U, V)is zero.
- In the actual implementation U and V are rescaled to have a more convenient maximum and minimum.
- For dealing with composite video, it turns out to be convenient to contain U, V within the range − 1/3to +4/3. So U and V are rescaled:
- U =0.492111 (B’− Y’)
- V =0.877283 (R’− Y’) (4.29)
- The chrominance signal = the composite signal C:
- $C = U \cdot \cos(\omega t) + V \cdot \sin(\omega t)$ (4.30)

- Zero is not the minimum value for U, V.
- U is approximately from blue (U> 0) to yellow (U< 0) in the RGB cube; V is approximately from red (V> 0) to cyan (V< 0).
- Fig. shows the decomposition of a color image into its Y', U, V components. Since both U and V go negative, in fact the images displayed are shifted and rescaled.

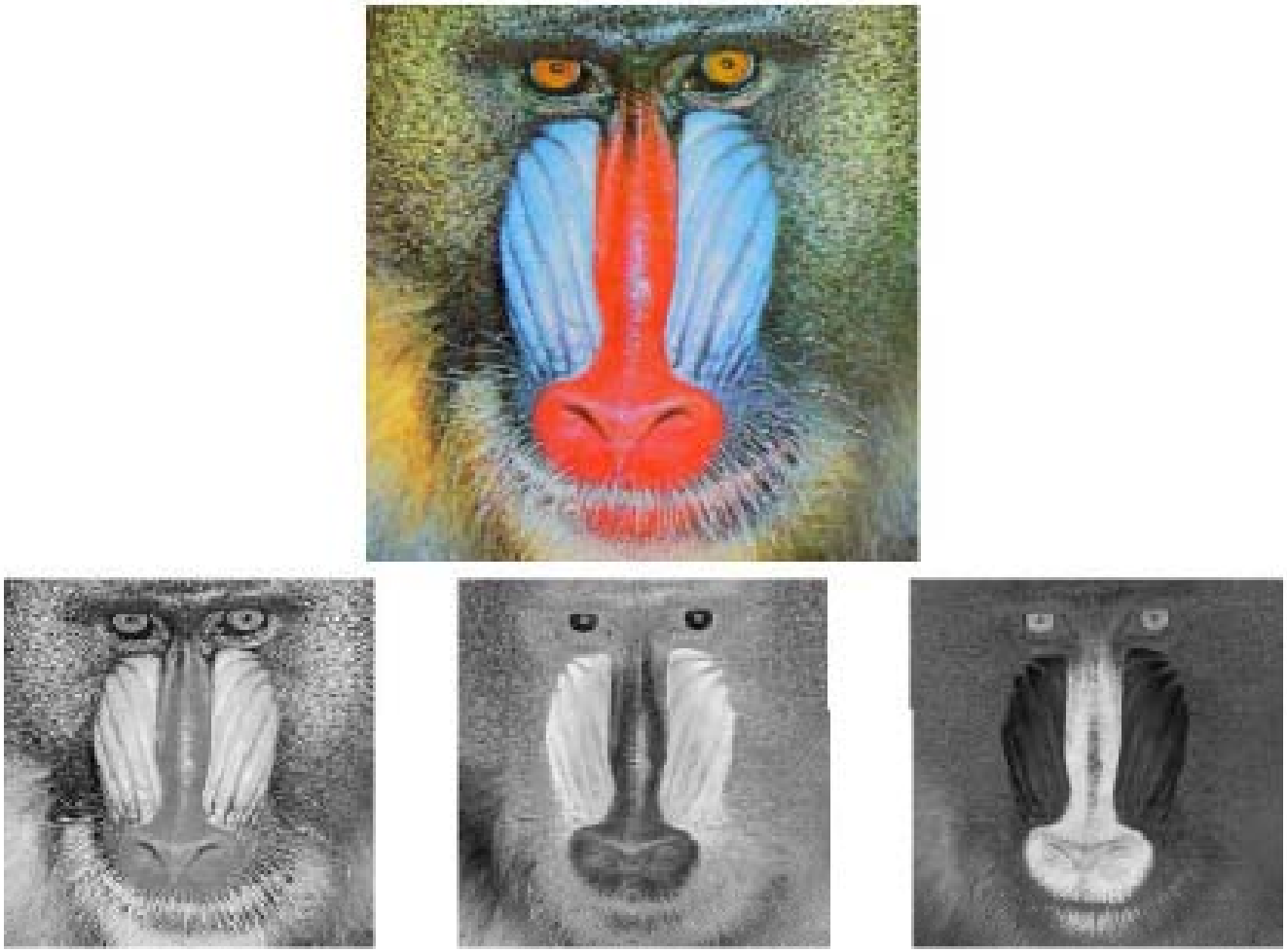
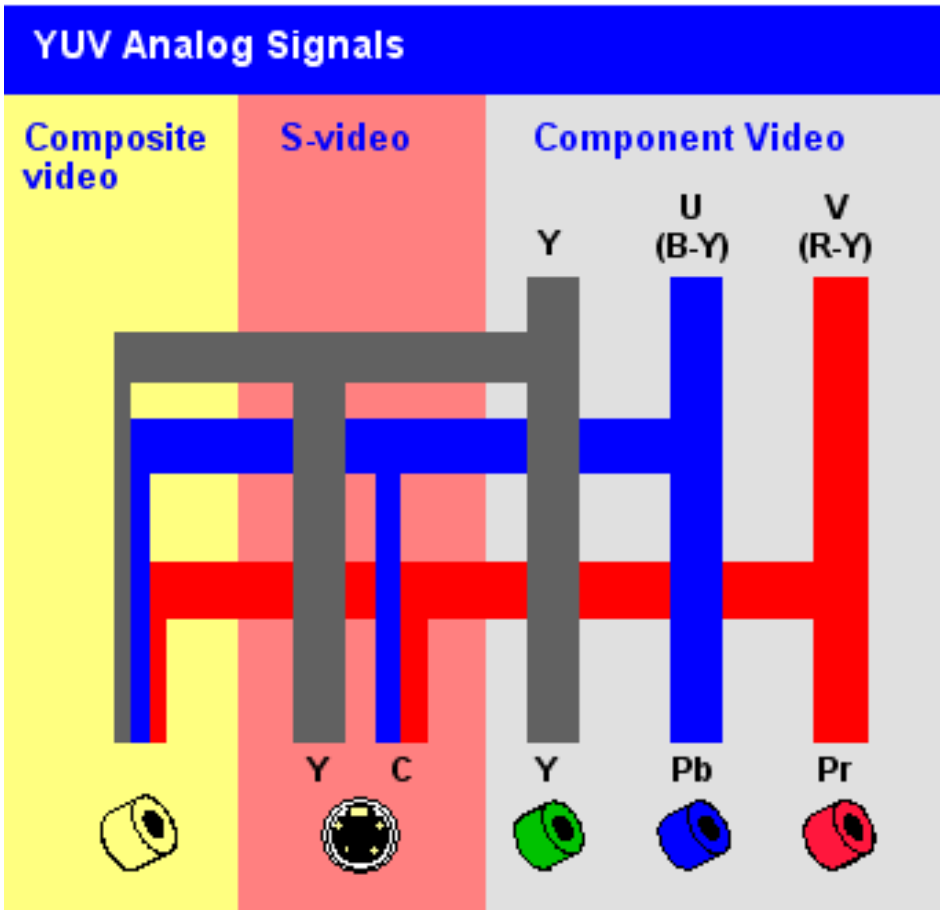


Fig. 4.18: Y'UV decomposition of color image. Top image (a) is original color image; (b) is Y'; (c,d) are (U,V)

UNIT II

Fundamental Concepts in Video

- Types of Video Signals
  - Component Video
  - Composite Video
  - S Video



Component Video Signals

- Higher-end video systems, such as studios make use of three separate video signals for the red, green, and blue image planes. This is referred as Component Video.
- Each color channel is sent as a separate video signal.

a) Most computer systems use Component Video, with separate signals for R, G, and B signals.

b) For any color separation scheme, Component Video gives the best color reproduction since there is no "crosstalk" between the three channels.

c) This is not the case for S-Video or Composite Video. Component video, however, requires more bandwidth and good synchronization of the three components .

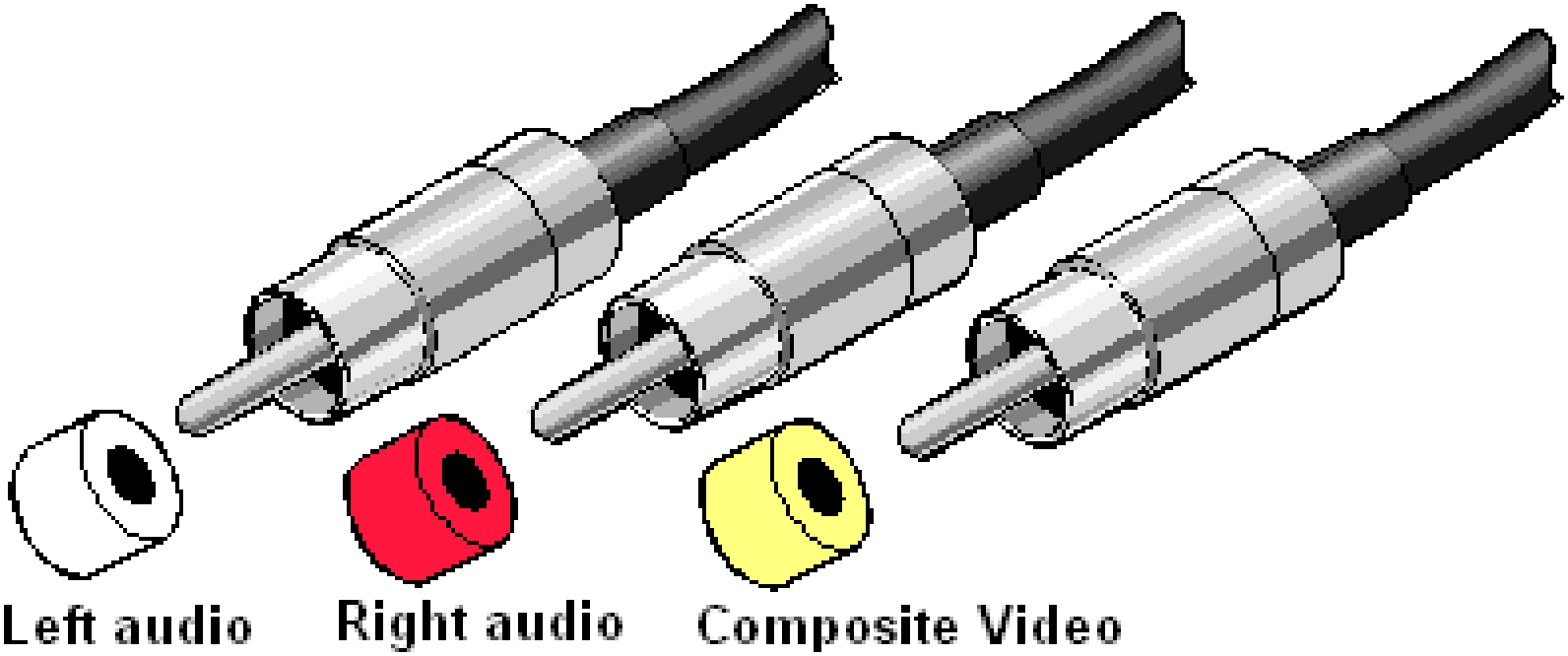
Composite Video Signal

- In Composite video color ("chrominance") and intensity ("luminance") signals are mixed into a single carrier wave.

a) Chrominance is a composition of two color components (I and Q, or U and V).)

b) In NTSC TV, e.g., I and Q are combined into a chroma signal, and a color subcarrier is then employed to put the chroma signal at the high-frequency end of the signal shared with the luminance signal.

c) The chrominance and luminance components can be separated at the receiver end and then the two color components can be further recovered .



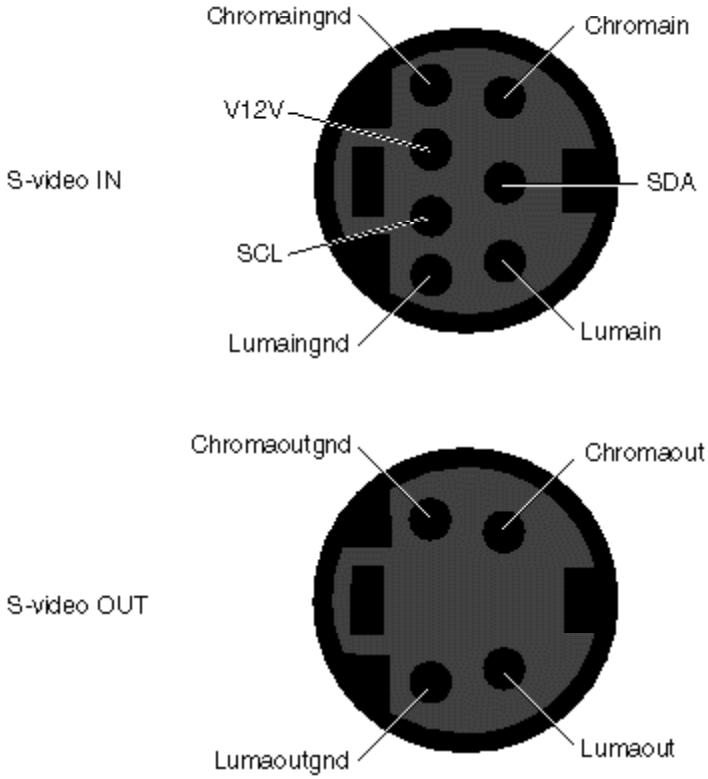
d) When connecting to TVs or VCRs, Composite Video uses only one wire and video color signals are mixed, not sent separately.

The audio and sync signals are additions to this one signal.

- Since color and intensity are wrapped into the same signal, some interference between the luminance and chrominance signals is inevitable.

S Video Signal

- S-Video: as a compromise, (Separated video, or Super-video, e.g., in S-VHS) uses two wires, one for luminance and another for a composite chrominance signal.
- As a result, there is less crosstalk between the color information and the crucial gray-scale information.
- The reason for placing luminance into its own part of the signal is that black-and-white information is most crucial for visual perception.



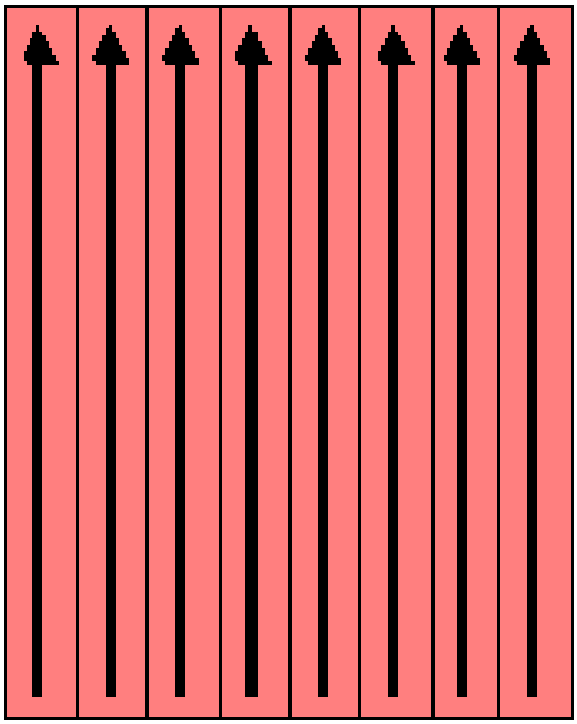
- In fact, humans are able to differentiate spatial resolution in gray-scale images with a much higher acuity than for the color part of color images.
- As a result, we can send less accurate color information than must be sent for intensity information — we can only see fairly large blobs of color, so it makes sense to send less color detail.

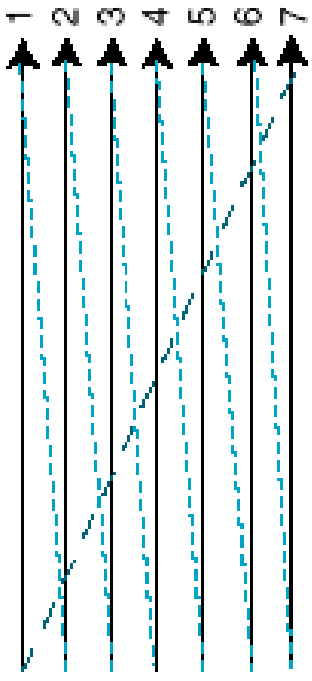
Analog Video

- An analog signal f(t) samples a time-varying image. So-called "progressive" scanning traces through a complete picture (a frame) row-wise for each time interval.
- In TV, and in some monitors and multimedia standards, another system, called "interlaced" scanning is used:

  a) The odd-numbered lines are traced first, and then the even-numbered lines are traced. This results in "odd" and "even" fields — two fields make up one frame.

  b) In fact, the odd lines (starting from 1) end up at the middle of a line at the end of the odd field, and the even scan starts at a half-way point.




1
2
3
4
5
6
7

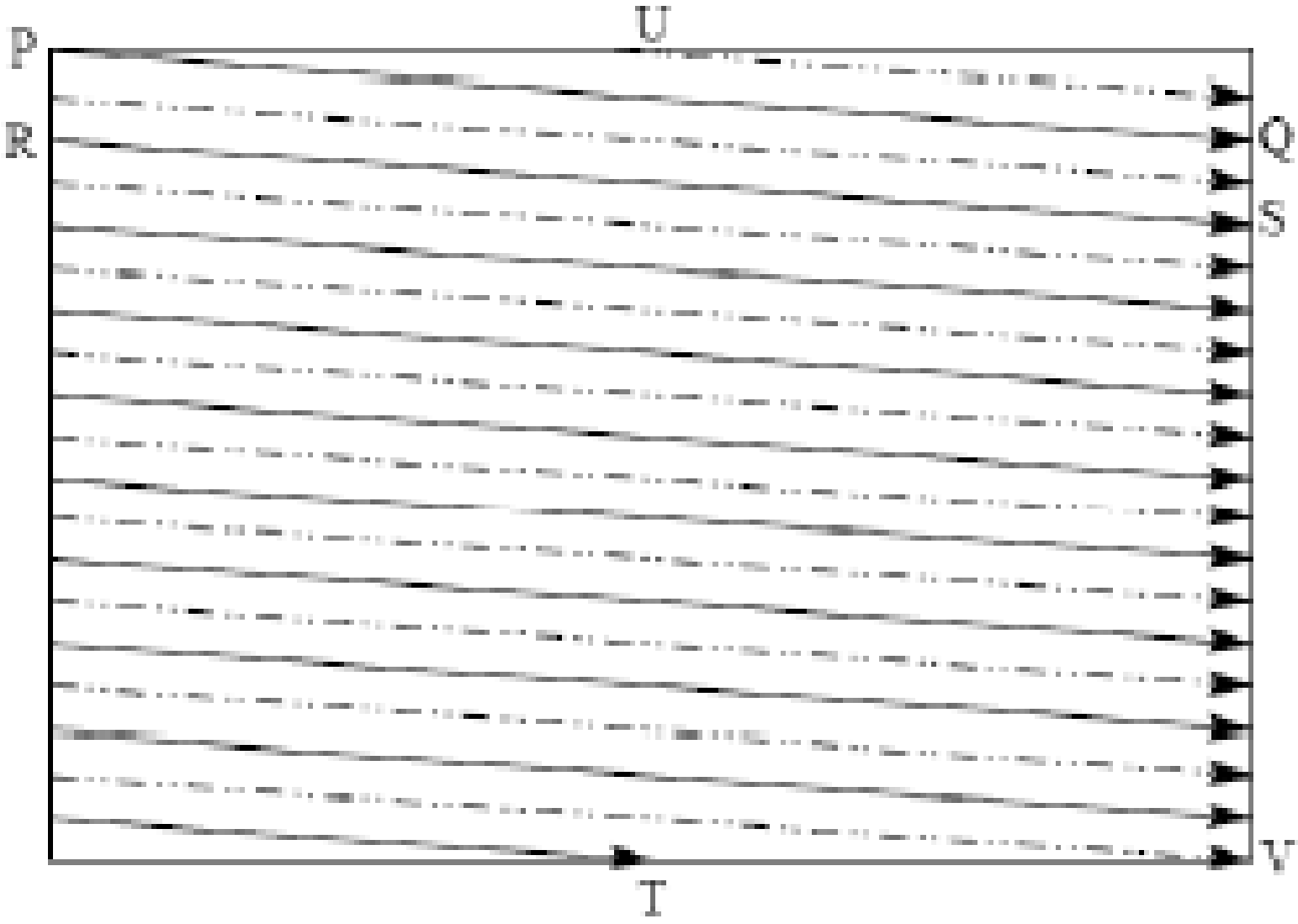


Fig. 5.1: Interlaced raster scan

The jump from Q to R, etc. in Figure is called the horizontal retrace, during which the electronic beam in the CRT is blanked. The jump from T to U or V to P is called the vertical retrace.

- Becau
  interl
  other
  very f
  blurri
- For
  helicc
  backg

m each
when
when
ing

(a)

(b)

(c)

(d)

- Since it is sometimes necessary to change the frame rate, resize, or even produce stills from an interlaced source video, various schemes are used to de-interlace it.

  a) The simplest de-interlacing method consists of discarding one field and duplicating the scan lines of the other field. The information in one field is lost completely using this simple technique.

  b) Other more complicated methods that retain information from both fields are also possible.

- Analog video use a small voltage offset from zero to indicate “black", and another value such as zero to indicate the start of a line.
- For example, we could use a "blacker-than-black" zero signal to indicate the beginning of a line.

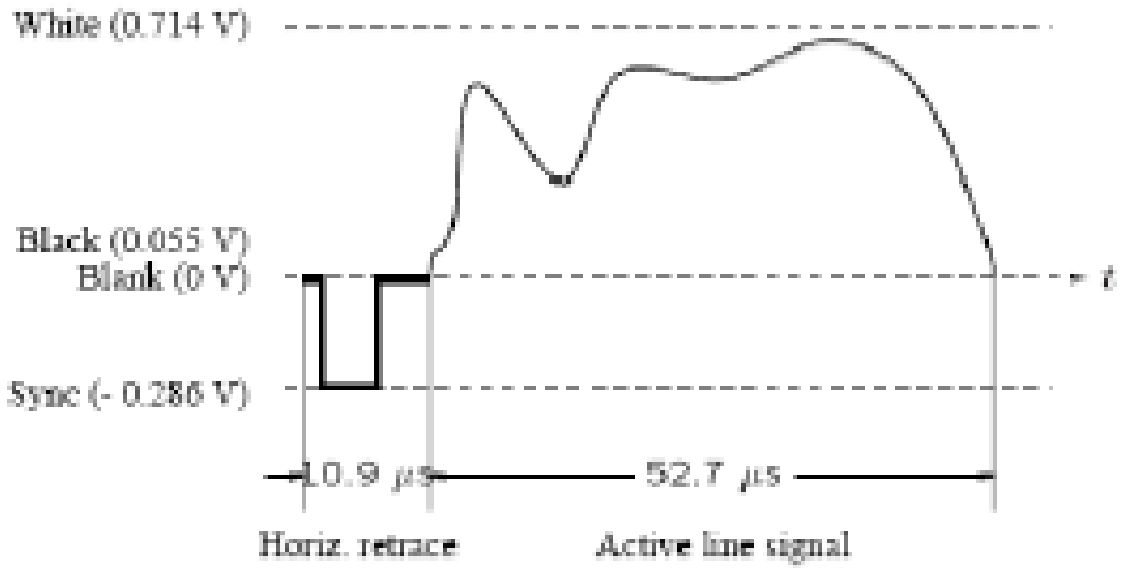


Fig. 5.3 Electronic signal for one NTSC scan line.

NTSC Video

- NTSC (National Television System Committee) TV standard is mostly used in North America and Japan. It uses the familiar 4:3 aspect ratio (i.e., the ratio of picture width to its height) and uses 525 scan lines per frame at 30 frames per second (fps).

a) NTSC follows the interlaced scanning system, and each frame is divided into two fields, with 262.5 lines/field.

b) Thus the horizontal sweep frequency is 525×29.97 ≈ 15, 734 lines/sec, so that each line is swept out in 1/15.734 × 103 sec ≈ 63.6μsec.

c) Since the horizontal retrace takes 10.9 μsec, this leaves 52.7 μsec for the active line signal during which image data is displayed.

PAL Video

- PAL (Phase Alternating Line) is a TV standard widely used in Western Europe, China, India, and many other parts of the world.
- PAL uses 625 scan frame at 25 frames/second with a 4:3 aspect ratio and interlaced fields.

a) PAL uses the YUV color model. It uses an 8 MHz channel and allocates a bandwidth of 5.5 MHz to Y, and 1.8 MHz each to U and V. The color subcarrier frequency is fsc ≈ 4.43 MHz.

b) In order to improve picture quality, chroma signals have alternate signs (e.g., +U and -U) in successive scan lines, hence the name "Phase Alternating Line".

c) This facilitates the use of a (line rate) comb filter at the receiver — the signals in consecutive lines are averaged so as to cancel the chroma signals (that always carry opposite signs) for separating Y and C and obtaining high quality Y signals.

SECAM Video

- SECAM stands for Syst `eme Electronique Couleur Avec M´emoire, the third major broadcast TV standard.
- SECAM also uses 625 scan lines per frame, at 25 frames per second, with a 4:3 aspect ratio and interlaced fields.
- SECAM and PAL are very similar. They differ slightly in their color coding scheme:

a) In SECAM, U and V signals are modulated using separate color subcarriers at 4.25 MHz and 4.41 MHz respectively.

b) They are sent in alternate lines, i.e., only one of the U or V signals will be sent on each scan line.

Table 5.2: Comparison of Analog Broadcast TV Systems

TV System	Frame Rate (fps)	# of Scan Lines	Total Channel Width (MHz)	Bandwidth Allocation (MHz)		
				Y	I or U	Q or V
NTSC	29.97	525	6.0	4.2	1.6	0.6
PAL	25	625	8.0	5.5	1.8	1.8
SECAM	25	625	8.0	6.0	2.0	2.0

Digital Video

The advantages of digital representation for video are many. For example:

a) Video can be stored on digital devices or in memory, ready to be processed (noise removal, cut and paste, etc.), and integrated to various multimedia applications;

b) Direct access is possible, which makes nonlinear video editing achievable as a simple, rather than a complex, task;

c) Repeated recording does not degrade image quality;

d) Ease of encryption and better tolerance to channel noise.

Chroma Subsampling

- Since humans see color with much less spatial resolution than they see black and white, it makes sense to "decimate" the chrominance signal.

Numbers are given stating how many pixel values, per four original pixels, are actually sent:

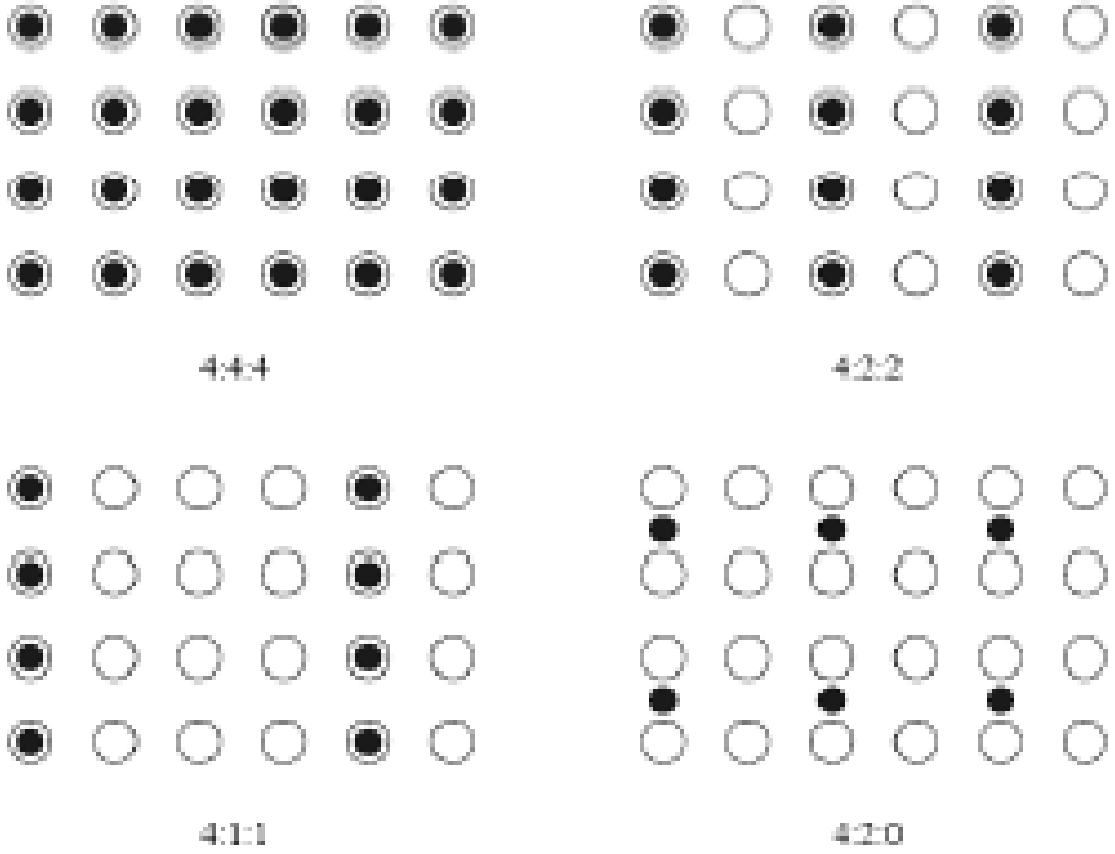
a) The chroma subsampling scheme "4:4:4" indicates that no chroma subsampling is used: each pixel's Y, Cb and Cr values are transmitted, 4 for each of Y, Cb, Cr.

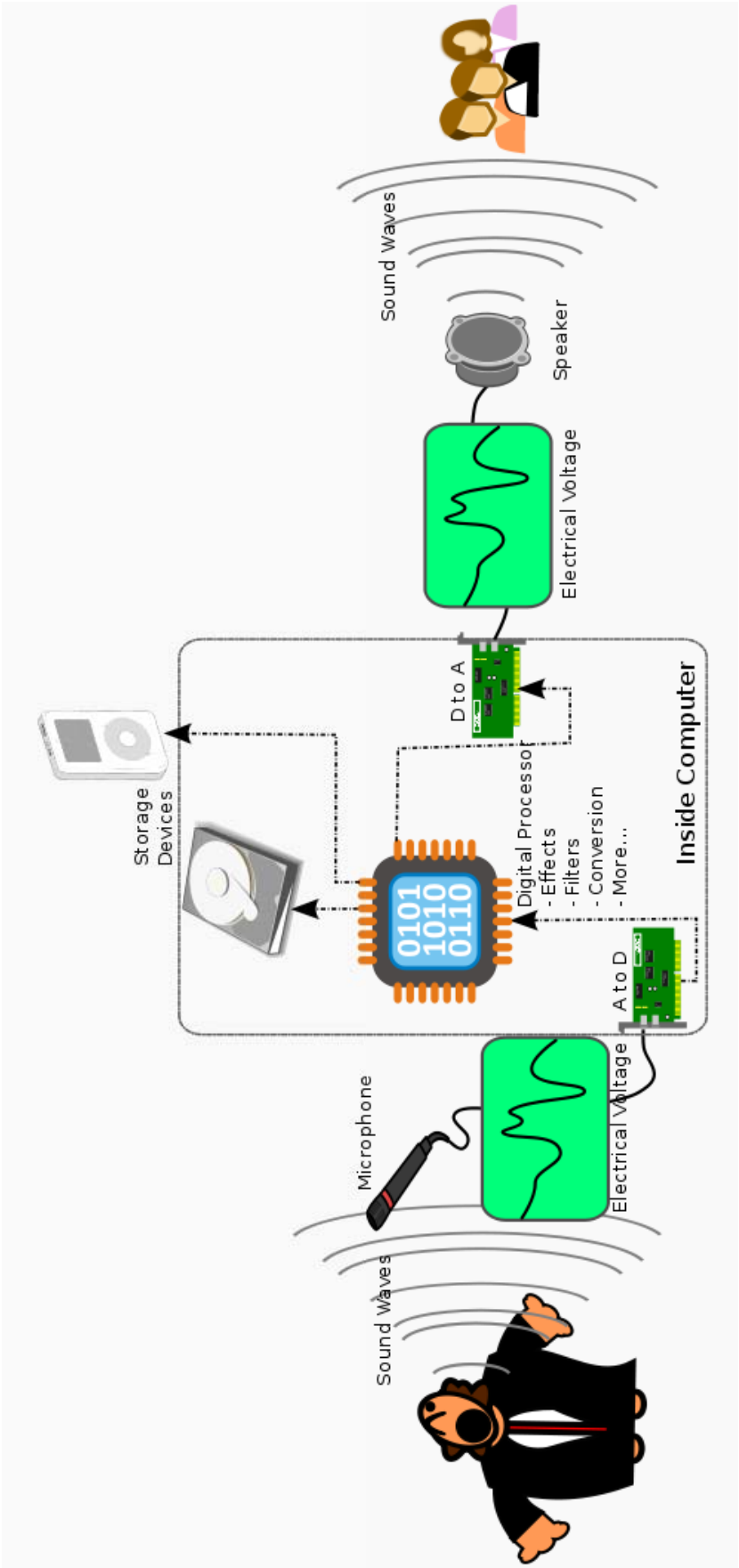
b) The scheme "4:2:2" indicates horizontal subsampling of the Cb, Cr signals by a factor of 2. That is, of four pixels horizontally labelled as 0 to 3, all four Ys are sent, and every two Cb's and two Cr's are sent, as (Cb0, Y0)(Cr0, Y1)(Cb2, Y2)(Cr2, Y3)(Cb4, Y4), and so on (or averaging is used).

c) The scheme "4:1:1" subsamples horizontally by a factor of 4.

d) The scheme "4:2:0" subsamples in both the horizontal and vertical dimensions by a factor of 2. Theoretically average chroma pixel is positioned between the rows and columns

Chroma Subsampling




Sound Waves
Speaker
Electrical Voltage
D to A
Inside Computer
Storage Devices
0101
1010
0110
Digital Processor
- Effects
- Filters
- Conversion
- More...
A to D
Electrical Voltage
Microphone
Sound Waves

Basics of Digital Audio

- **What is Sound?**
- Sound is a wave phenomenon like light, but is macroscopic and involves molecules of air being compressed and expanded under the action of some physical device.

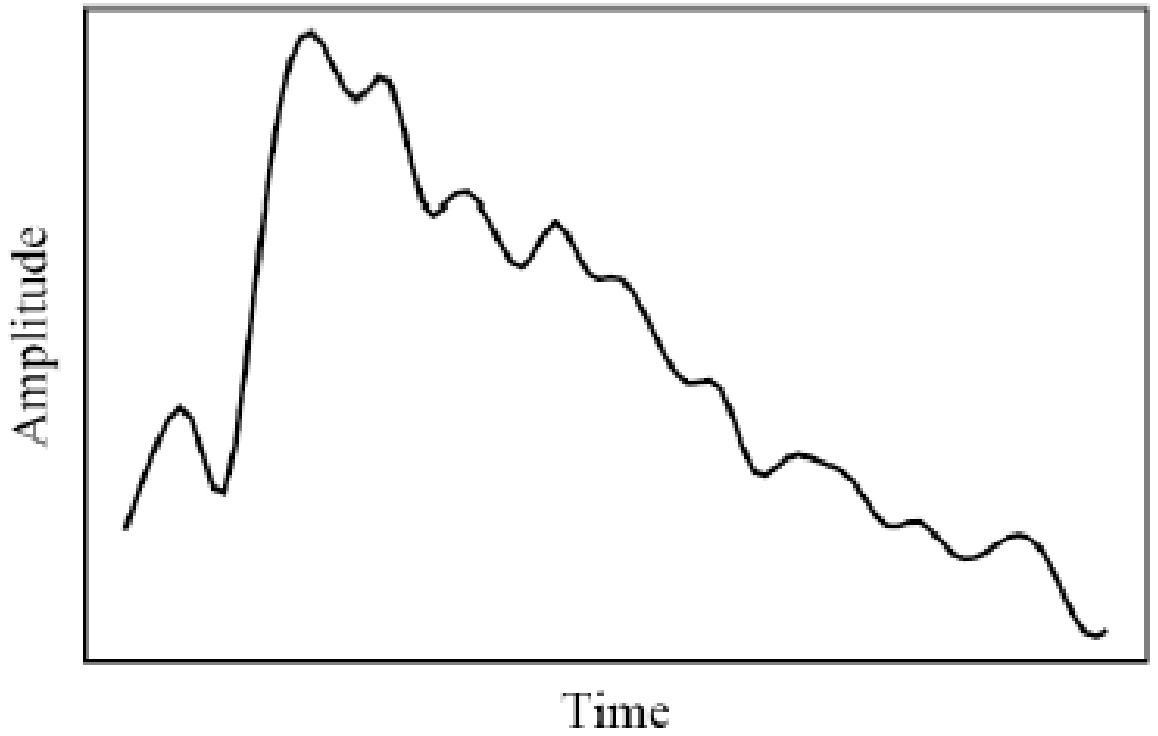
  a) For example, a speaker in an audio system vibrates back and forth and produces a longitudinal pressure wave that we perceive as sound.

  b) Since sound is a pressure wave, it takes on continuous values, as opposed to digitized ones.

- They have ordinary wave properties and behaviors, such as reflection (bouncing), refraction (change of angle when entering a medium with a different density) and diffraction (bending around an obstacle).
- If we wish to use a digital version of sound waves we must form digitized representations of audio information.

Digitization

- Digitization means conversion to a stream of numbers, and preferably these numbers should be integers for efficiency.



An analog signal: continuous measurement of pressure wave.

Sampling and Quantization

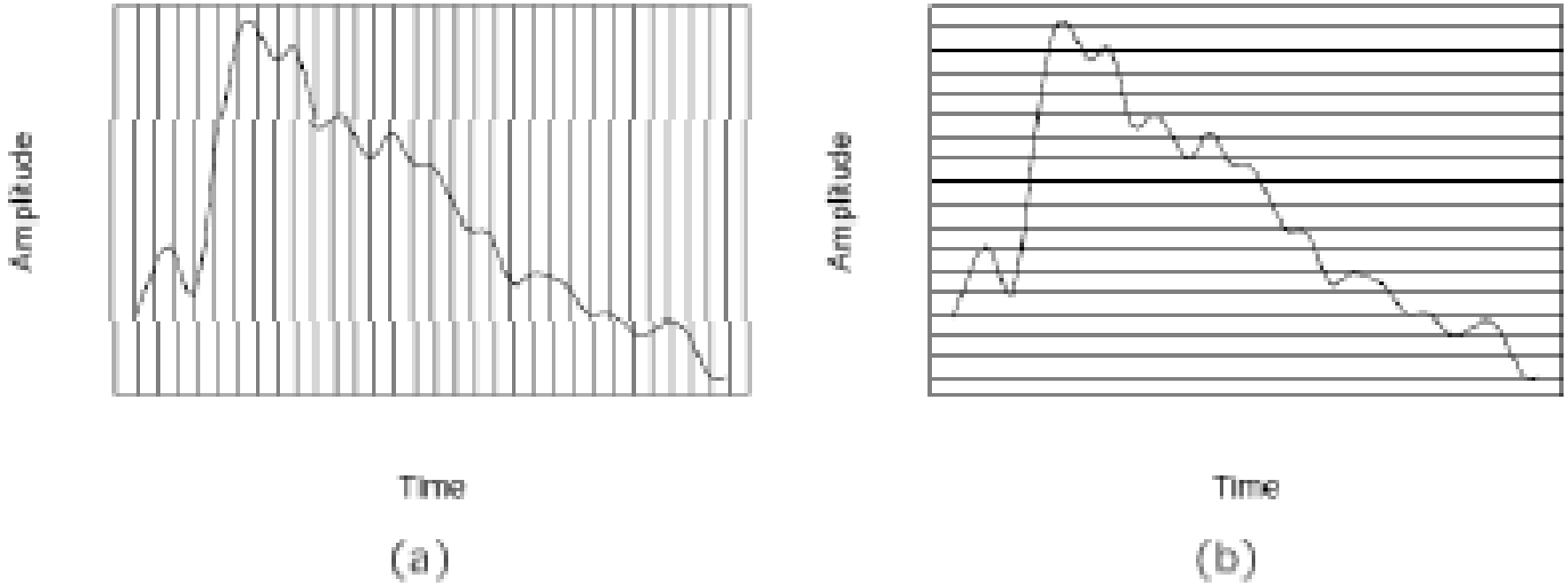
- The graph in Fig. has to be made digital in both time and amplitude. To digitize, the signal must be sampled in each dimension: in time, and amplitude.

a) Sampling means measuring the quantity we are interested in, usually at evenly-spaced intervals.

b) The first kind of sampling, using measurements only at evenly spaced time intervals, is simply called, sampling The rate at which it is performed is called the sampling frequency (see Fig. (a)).

c) For audio, typical sampling rates are from 8 kHz (8,000 samples per second) to 48 kHz. This range is determined by Nyquist theorem.

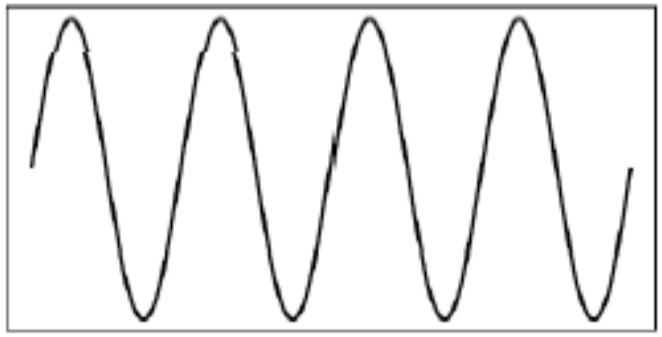
d) Sampling in the amplitude or voltage dimension is called quantization. Fig. (b) shows this kind of sampling.



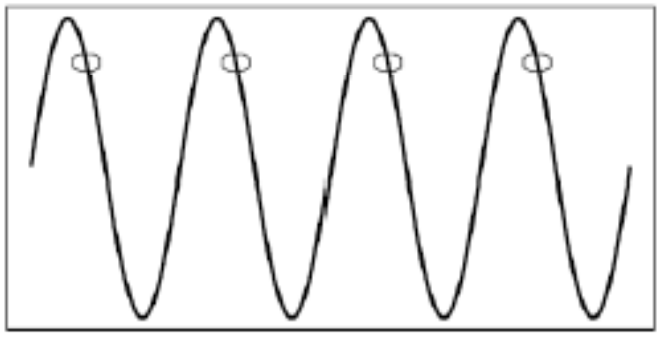
Sampling and Quantization.
(a): Sampling the analog signal in the time dimension.
(b): Quantization is sampling the analog signal in the amplitude dimension.

Nyquist Theorem

- The Nyquist theorem states how frequently we must sample in time to be able to recover the original sound.

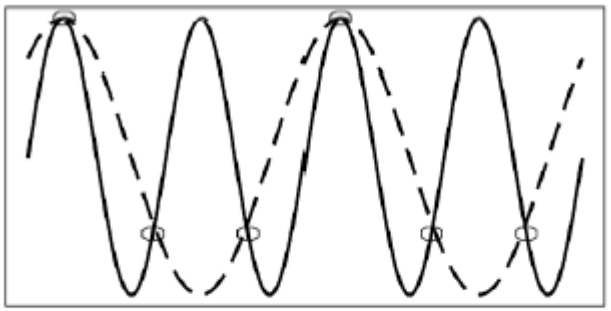


(a): A single frequency.



(b): Sampling at exactly the frequency produces a constant.

Fig (a) shows a single sinusoid: it is a single, pure frequency (only electronic instruments can create such
sounds).
If sampling rate just equals the actual frequency, Fig. (b) shows that a false signal is detected: it is simply a constant, with zero frequency.



(c): Sampling at 1.5 times per cycle produces an alias perceived frequency.

- Now sample at 1.5 times the actual frequency, Fig. (c) shows that we obtain an incorrect (alias) frequency that is lower than the correct one — it is half the correct one (the wavelength from peak wavelength, peak to peak, is double that of the actual signal).
- Thus for correct sampling we must use a sampling rate equal to at least twice the maximum frequency content in the signal. This rate is called the Nyquist rate.

- Nyquist Theorem: If a signal is band-limited, i.e., band there is a lower limit f1 and an upper limit f2 of frequency components in the signal, then the sampling rate should be at least 2(f2 − f1).

Signal to Noise Ratio (SNR)

- The ratio of the power of the correct signal and the noise is called the signal to noise ratio (SNR) a measure of the quality of the signal — signal.
- The SNR is usually measured in decibels (dB), where 1 dB is a tenth of a bel The SNR value in bel. value, units of dB, is defined in terms of base-10 logarithms of squared voltages, as follows:

$$SNR = 10 \log_{10} \frac{V_{signal}^2}{V_{noise}^2} = 20 \log_{10} \frac{V_{signal}}{V_{noise}}$$

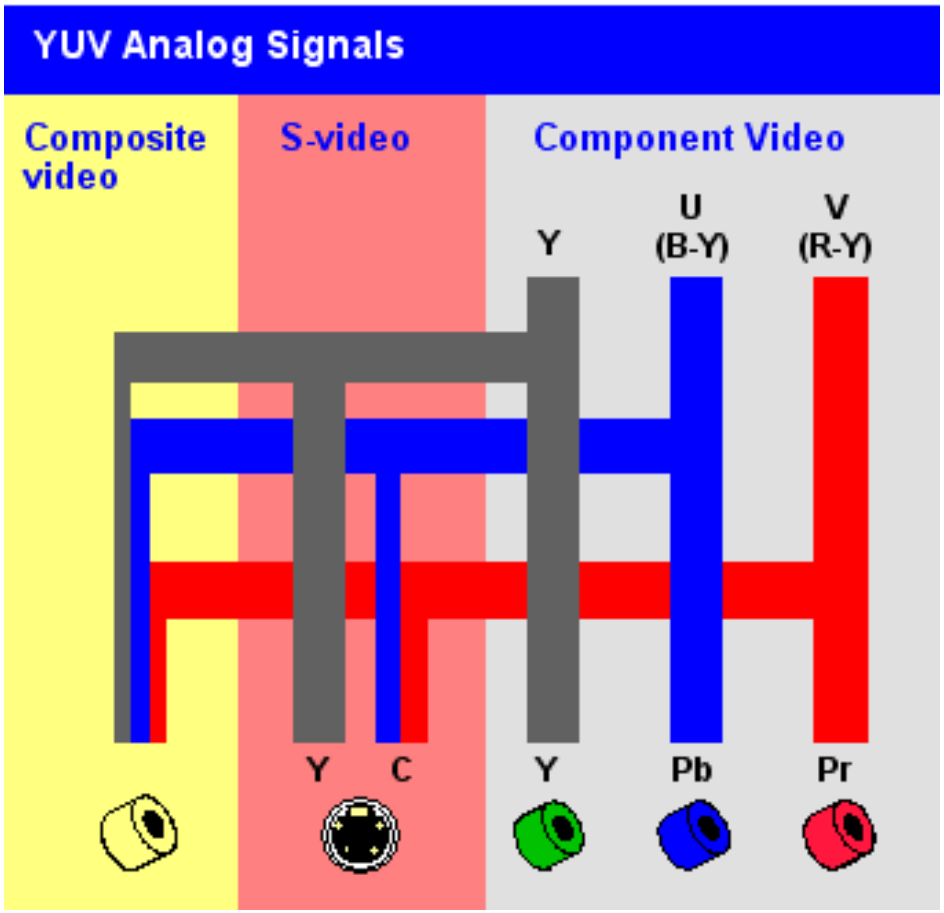
Table 6.1: Magnitude levels of common sounds, in decibels

Threshold of hearing	0
Rustle of leaves	10
Very quiet room	20
Average room	40
Conversation	60
Busy street	70
Loud radio	80
Train through station	90
Riveter	100
Threshold of discomfort	120
Threshold of pain	140
Damage to ear drum	160

UNIT II

Fundamental Concepts in Video

- Types of Video Signals
  - Component Video
  - Composite Video
  - S Video



Component Video Signals

- Higher-end video systems, such as studios make use of three separate video signals for the red, green, and blue image planes. This is referred as Component Video.
- Each color channel is sent as a separate video signal.

a) Most computer systems use Component Video, with separate signals for R, G, and B signals.

b) For any color separation scheme, Component Video gives the best color reproduction since there is no "crosstalk" between the three channels.

c) This is not the case for S-Video or Composite Video. Component video, however, requires more bandwidth and good synchronization of the three components .

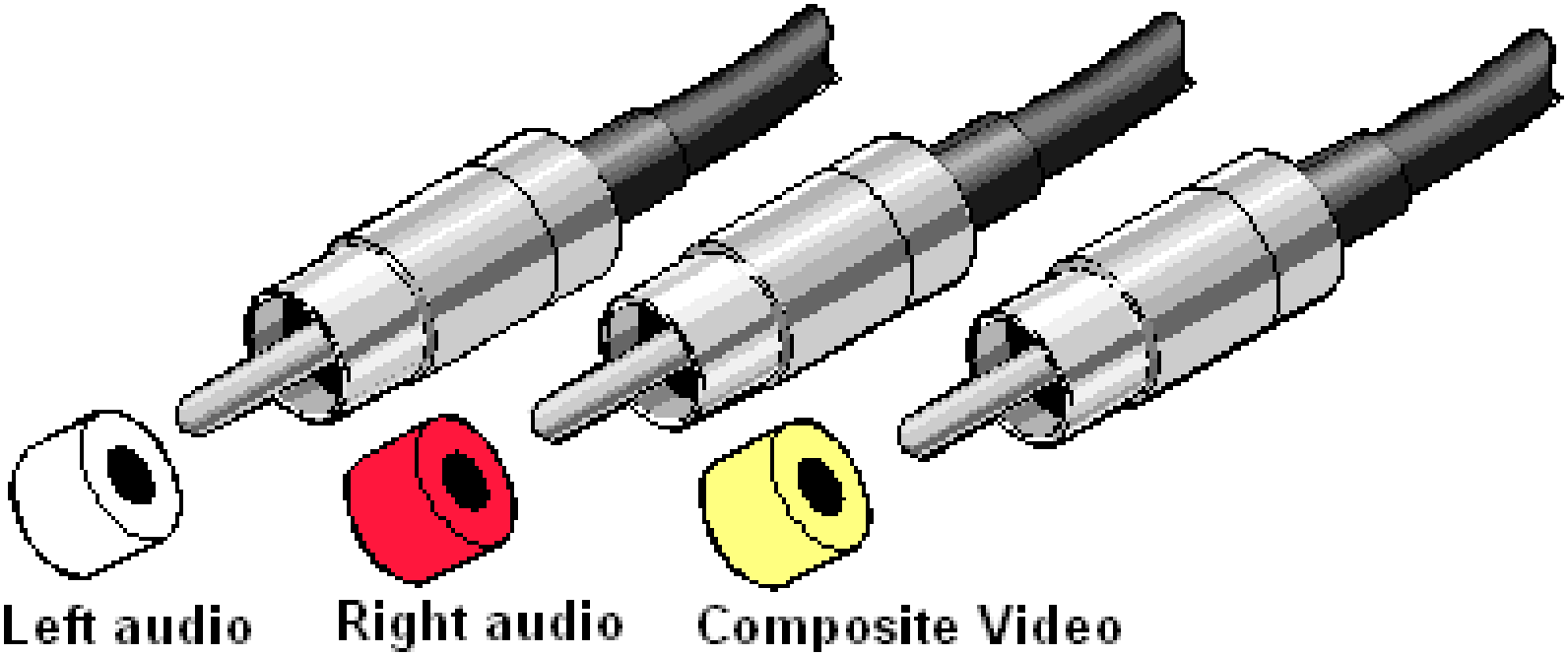
Composite Video Signal

- In Composite video color ("chrominance") and intensity ("luminance") signals are mixed into a single carrier wave.

a) Chrominance is a composition of two color components (I and Q, or U and V).)

b) In NTSC TV, e.g., I and Q are combined into a chroma signal, and a color subcarrier is then employed to put the chroma signal at the high-frequency end of the signal shared with the luminance signal.

c) The chrominance and luminance components can be separated at the receiver end and then the two color components can be further recovered .



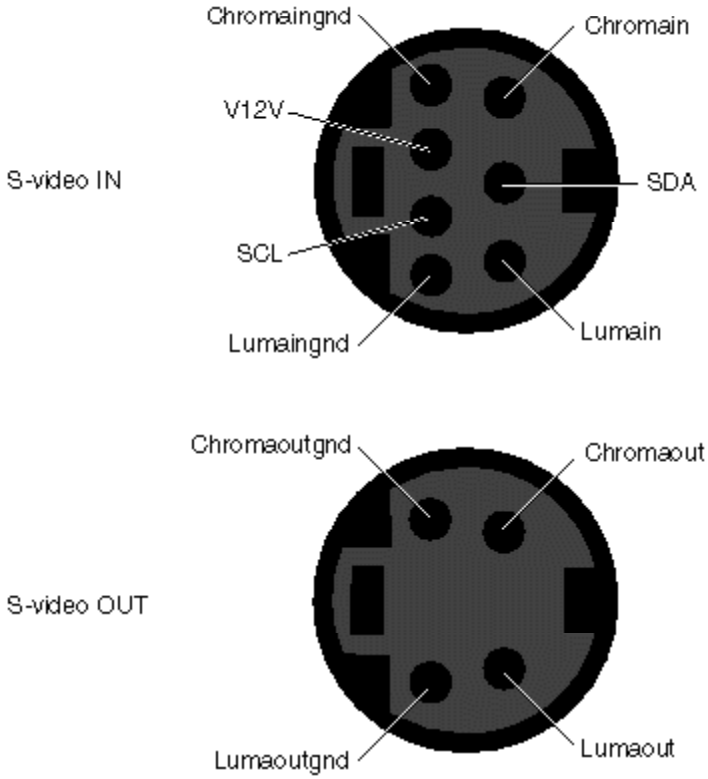
d) When connecting to TVs or VCRs, Composite Video uses only one wire and video color signals are mixed, not sent separately.

The audio and sync signals are additions to this one signal.

- Since color and intensity are wrapped into the same signal, some interference between the luminance and chrominance signals is inevitable.

S Video Signal

- S-Video: as a compromise, (Separated video, or Super-video, e.g., in S-VHS) uses two wires, one for luminance and another for a composite chrominance signal.
- As a result, there is less crosstalk between the color information and the crucial gray-scale information.
- The reason for placing luminance into its own part of the signal is that black-and-white information is most crucial for visual perception.



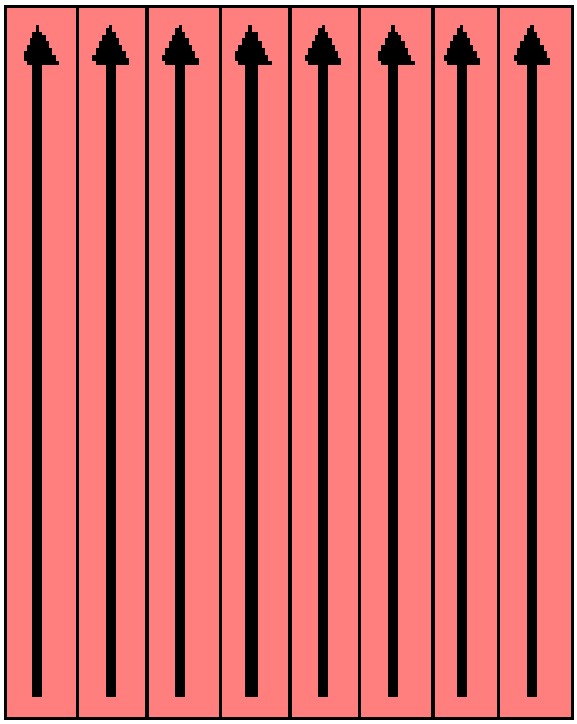
- In fact, humans are able to differentiate spatial resolution in gray-scale images with a much higher acuity than for the color part of color images.
- As a result, we can send less accurate color information than must be sent for intensity information — we can only see fairly large blobs of color, so it makes sense to send less color detail.

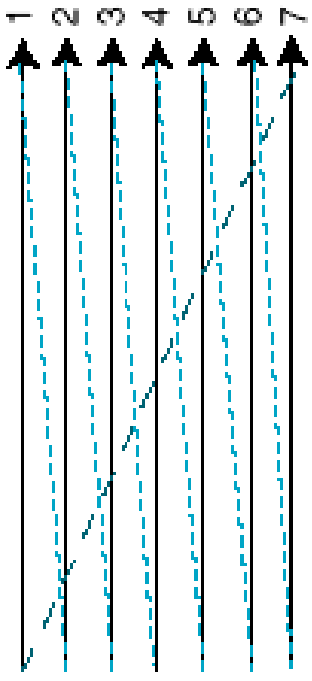
Analog Video

- An analog signal f(t) samples a time-varying image. So-called "progressive" scanning traces through a complete picture (a frame) row-wise for each time interval.
- In TV, and in some monitors and multimedia standards, another system, called "interlaced" scanning is used:

  a) The odd-numbered lines are traced first, and then the even-numbered lines are traced. This results in "odd" and "even" fields — two fields make up one frame.

  b) In fact, the odd lines (starting from 1) end up at the middle of a line at the end of the odd field, and the even scan starts at a half-way point.




1
2
3
4
5
6
7

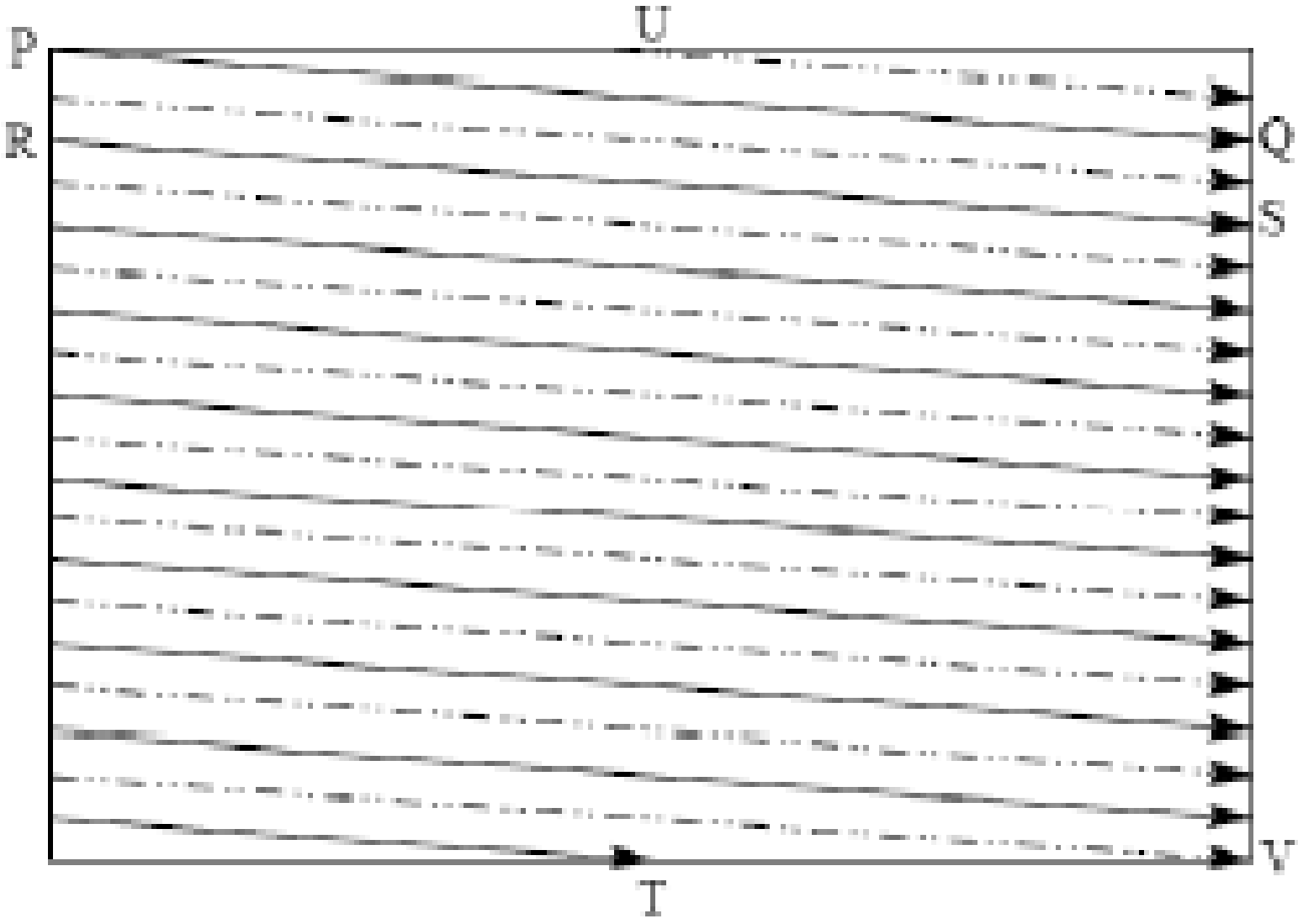


Fig. 5.1: Interlaced raster scan

The jump from Q to R, etc. in Figure is called the horizontal retrace, during which the electronic beam in the CRT is blanked. The jump from T to U or V to P is called the vertical retrace.

- Becau
  interl
  other
  very f
  blurri
- For
  helicc
  backg

m each
when
when
ing

(a)

(b)

(c)

(d)

- Since it is sometimes necessary to change the frame rate, resize, or even produce stills from an interlaced source video, various schemes are used to de-interlace it.

  a) The simplest de-interlacing method consists of discarding one field and duplicating the scan lines of the other field. The information in one field is lost completely using this simple technique.

  b) Other more complicated methods that retain information from both fields are also possible.

- Analog video use a small voltage offset from zero to indicate “black", and another value such as zero to indicate the start of a line.
- For example, we could use a "blacker-than-black" zero signal to indicate the beginning of a line.

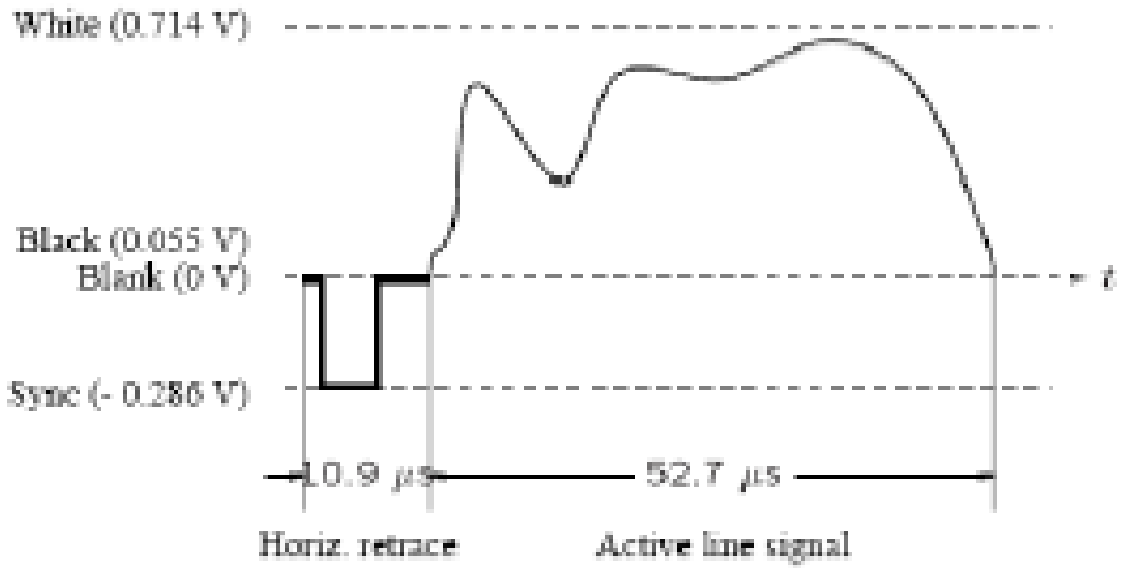


Fig. 5.3 Electronic signal for one NTSC scan line.

NTSC Video

- NTSC (National Television System Committee) TV standard is mostly used in North America and Japan. It uses the familiar 4:3 aspect ratio (i.e., the ratio of picture width to its height) and uses 525 scan lines per frame at 30 frames per second (fps).

a) NTSC follows the interlaced scanning system, and each frame is divided into two fields, with 262.5 lines/field.

b) Thus the horizontal sweep frequency is 525×29.97 ≈ 15, 734 lines/sec, so that each line is swept out in 1/15.734 × 103 sec ≈ 63.6μsec.

c) Since the horizontal retrace takes 10.9 μsec, this leaves 52.7 μsec for the active line signal during which image data is displayed.

PAL Video

- PAL (Phase Alternating Line) is a TV standard widely used in Western Europe, China, India, and many other parts of the world.
- PAL uses 625 scan frame at 25 frames/second with a 4:3 aspect ratio and interlaced fields.

a) PAL uses the YUV color model. It uses an 8 MHz channel and allocates a bandwidth of 5.5 MHz to Y, and 1.8 MHz each to U and V. The color subcarrier frequency is fsc ≈ 4.43 MHz.

b) In order to improve picture quality, chroma signals have alternate signs (e.g., +U and -U) in successive scan lines, hence the name "Phase Alternating Line".

c) This facilitates the use of a (line rate) comb filter at the receiver — the signals in consecutive lines are averaged so as to cancel the chroma signals (that always carry opposite signs) for separating Y and C and obtaining high quality Y signals.

SECAM Video

- SECAM stands for Syst `eme Electronique Couleur Avec M´emoire, the third major broadcast TV standard.
- SECAM also uses 625 scan lines per frame, at 25 frames per second, with a 4:3 aspect ratio and interlaced fields.
- SECAM and PAL are very similar. They differ slightly in their color coding scheme:

a) In SECAM, U and V signals are modulated using separate color subcarriers at 4.25 MHz and 4.41 MHz respectively.

b) They are sent in alternate lines, i.e., only one of the U or V signals will be sent on each scan line.

Table 5.2: Comparison of Analog Broadcast TV Systems

TV System	Frame Rate (fps)	# of Scan Lines	Total Channel Width (MHz)	Bandwidth Allocation (MHz)		
				Y	I or U	Q or V
NTSC	29.97	525	6.0	4.2	1.6	0.6
PAL	25	625	8.0	5.5	1.8	1.8
SECAM	25	625	8.0	6.0	2.0	2.0

Digital Video

The advantages of digital representation for video are many. For example:

a) Video can be stored on digital devices or in memory, ready to be processed (noise removal, cut and paste, etc.), and integrated to various multimedia applications;

b) Direct access is possible, which makes nonlinear video editing achievable as a simple, rather than a complex, task;

c) Repeated recording does not degrade image quality;

d) Ease of encryption and better tolerance to channel noise.

Chroma Subsampling

- Since humans see color with much less spatial resolution than they see black and white, it makes sense to "decimate" the chrominance signal.

Numbers are given stating how many pixel values, per four original pixels, are actually sent:

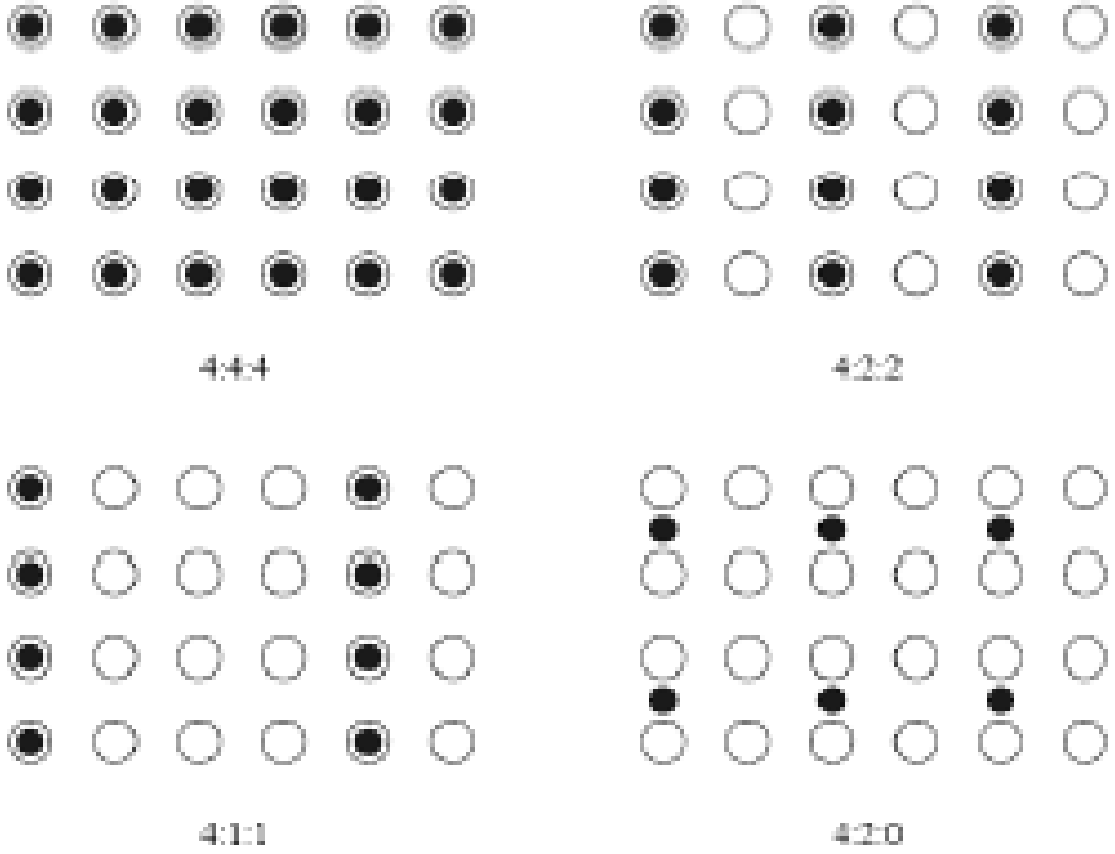
a) The chroma subsampling scheme "4:4:4" indicates that no chroma subsampling is used: each pixel's Y, Cb and Cr values are transmitted, 4 for each of Y, Cb, Cr.

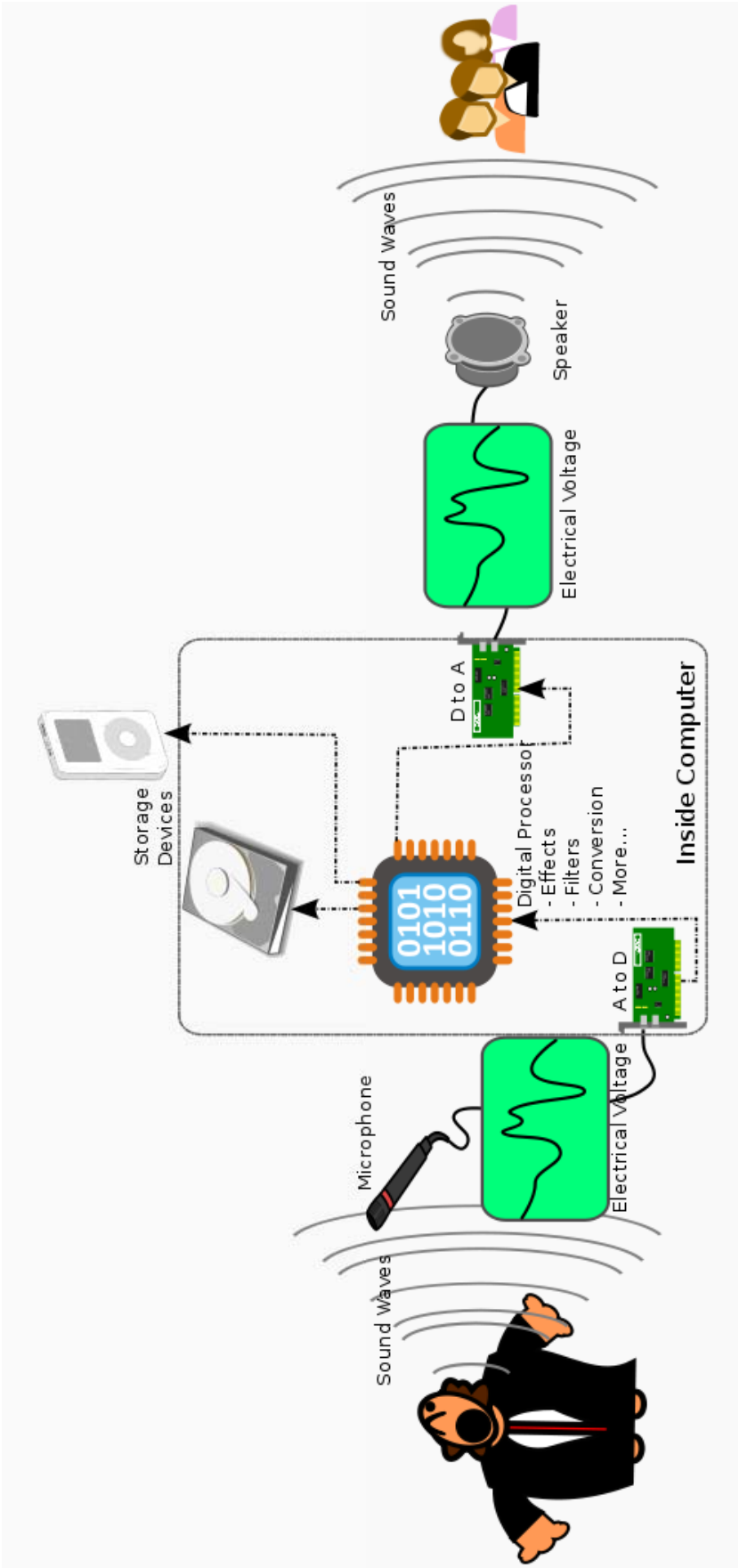
b) The scheme "4:2:2" indicates horizontal subsampling of the Cb, Cr signals by a factor of 2. That is, of four pixels horizontally labelled as 0 to 3, all four Ys are sent, and every two Cb's and two Cr's are sent, as (Cb0, Y0)(Cr0, Y1)(Cb2, Y2)(Cr2, Y3)(Cb4, Y4), and so on (or averaging is used).

c) The scheme "4:1:1" subsamples horizontally by a factor of 4.

d) The scheme "4:2:0" subsamples in both the horizontal and vertical dimensions by a factor of 2. Theoretically average chroma pixel is positioned between the rows and columns

Chroma Subsampling




Sound Waves
Speaker
Electrical Voltage
D to A
Inside Computer
Storage Devices
0101
1010
0110
Digital Processor
- Effects
- Filters
- Conversion
- More...
A to D
Electrical Voltage
Microphone
Sound Waves

Basics of Digital Audio

- **What is Sound?**
- Sound is a wave phenomenon like light, but is macroscopic and involves molecules of air being compressed and expanded under the action of some physical device.

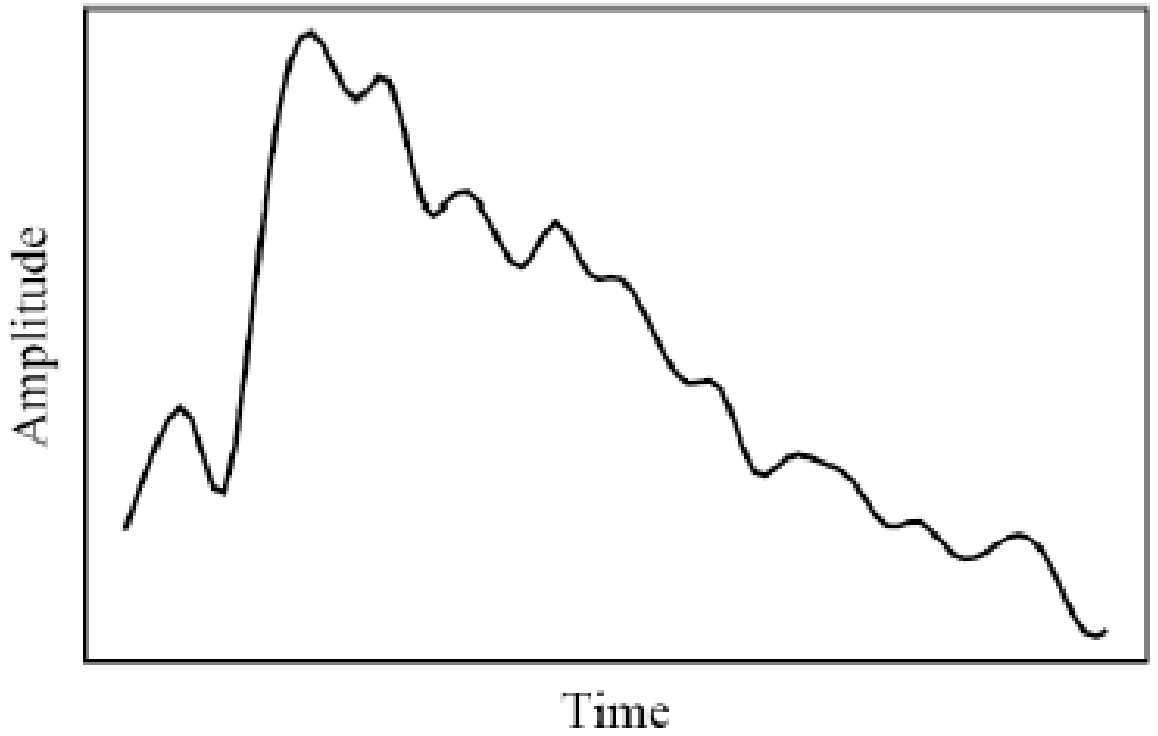
  a) For example, a speaker in an audio system vibrates back and forth and produces a longitudinal pressure wave that we perceive as sound.

  b) Since sound is a pressure wave, it takes on continuous values, as opposed to digitized ones.

- They have ordinary wave properties and behaviors, such as reflection (bouncing), refraction (change of angle when entering a medium with a different density) and diffraction (bending around an obstacle).
- If we wish to use a digital version of sound waves we must form digitized representations of audio information.

Digitization

- Digitization means conversion to a stream of numbers, and preferably these numbers should be integers for efficiency.



An analog signal: continuous measurement of pressure wave.

Sampling and Quantization

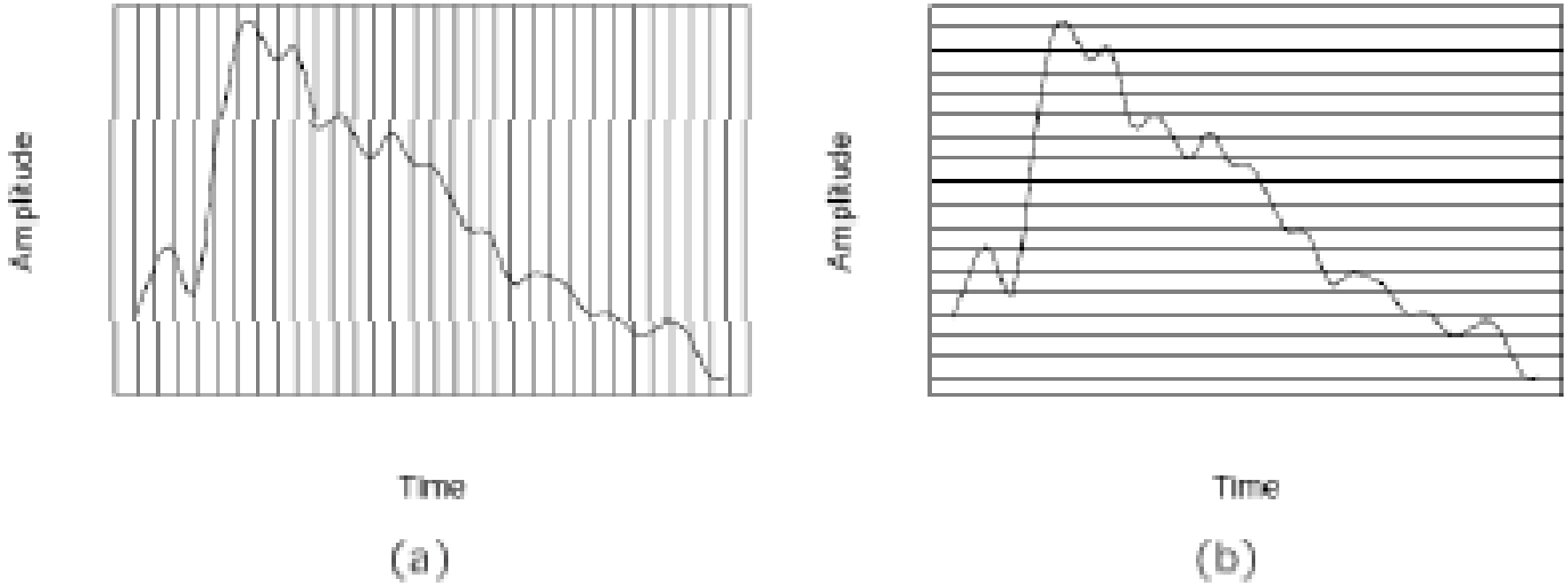
- The graph in Fig. has to be made digital in both time and amplitude. To digitize, the signal must be sampled in each dimension: in time, and amplitude.

a) Sampling means measuring the quantity we are interested in, usually at evenly-spaced intervals.

b) The first kind of sampling, using measurements only at evenly spaced time intervals, is simply called, sampling The rate at which it is performed is called the sampling frequency (see Fig. (a)).

c) For audio, typical sampling rates are from 8 kHz (8,000 samples per second) to 48 kHz. This range is determined by Nyquist theorem.

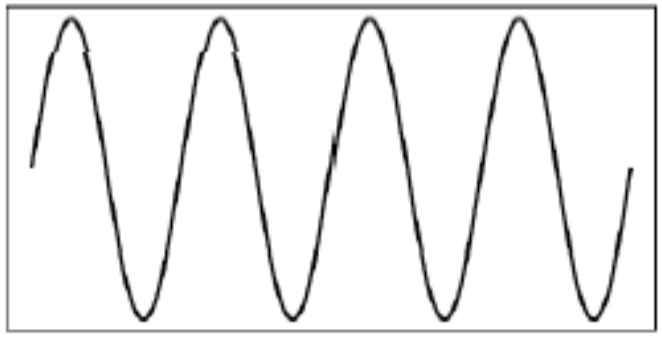
d) Sampling in the amplitude or voltage dimension is called quantization. Fig. (b) shows this kind of sampling.



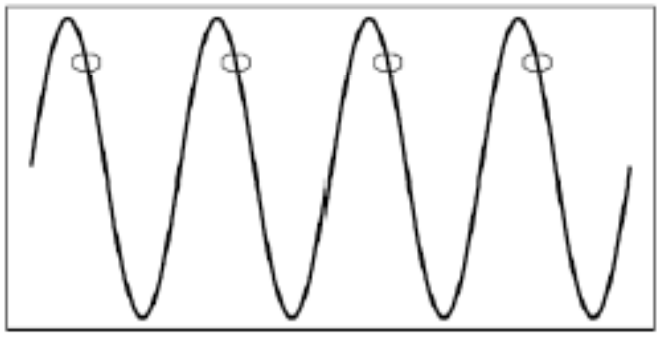
Sampling and Quantization.
(a): Sampling the analog signal in the time dimension.
(b): Quantization is sampling the analog signal in the amplitude dimension.

Nyquist Theorem

- The Nyquist theorem states how frequently we must sample in time to be able to recover the original sound.

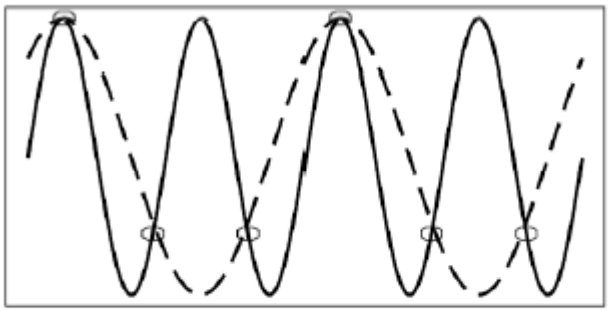


(a): A single frequency.



(b): Sampling at exactly the frequency produces a constant.

Fig (a) shows a single sinusoid: it is a single, pure frequency (only electronic instruments can create such
sounds).
If sampling rate just equals the actual frequency, Fig. (b) shows that a false signal is detected: it is simply a constant, with zero frequency.



(c): Sampling at 1.5 times per cycle produces an alias perceived frequency.

- Now sample at 1.5 times the actual frequency, Fig. (c) shows that we obtain an incorrect (alias) frequency that is lower than the correct one — it is half the correct one (the wavelength from peak wavelength, peak to peak, is double that of the actual signal).
- Thus for correct sampling we must use a sampling rate equal to at least twice the maximum frequency content in the signal. This rate is called the Nyquist rate.

- Nyquist Theorem: If a signal is band-limited, i.e., band there is a lower limit f1 and an upper limit f2 of frequency components in the signal, then the sampling rate should be at least 2(f2 − f1).

Signal to Noise Ratio (SNR)

- The ratio of the power of the correct signal and the noise is called the signal to noise ratio (SNR) a measure of the quality of the signal — signal.
- The SNR is usually measured in decibels (dB), where 1 dB is a tenth of a bel The SNR value in bel. value, units of dB, is defined in terms of base-10 logarithms of squared voltages, as follows:

$$SNR = 10 \log_{10} \frac{V_{signal}^2}{V_{noise}^2} = 20 \log_{10} \frac{V_{signal}}{V_{noise}}$$

Table 6.1: Magnitude levels of common sounds, in decibels

Threshold of hearing	0
Rustle of leaves	10
Very quiet room	20
Average room	40
Conversation	60
Busy street	70
Loud radio	80
Train through station	90
Riveter	100
Threshold of discomfort	120
Threshold of pain	140
Damage to ear drum	160

Unit III

Action Script I

ActionScript 2.0 Features

- ActionScript 2.0 adds relatively little *new runtime functionality* to the language but radically improves object-oriented development in Flash by formalizing objected-oriented programming *(OOP) syntax and methodology*.

- ActionScript 2.0 provides a class keyword for creating classes and an *extends* keyword for establishing *inheritance*. Those keywords were absent from ActionScript 1.0

Key ActionScript 2.0 Features introduced are:

- The *class* statement, used to create formal classes.
- The *extends* keyword, used to establish inheritance.
- The *interface* statement, used to create Java-style interfaces (i.e., abstract data types).
  - Classes provide implementations for interfaces using the *implements* keyword. ActionScript 1.0 did not support interfaces.
- The official file extension for class files is ***.as***. Formerly, classes could be defined in timeline code or in external .as files.
  - Class files can be edited in Flash MX Professional 2004's script editor or in an external text editor.

Key ActionScript 2.0 Features introduced are: (contd...)

- Formal method-definition syntax, used to *create instance methods and class methods* in a class body. In ActionScript 1.0, methods were added to a class via the class constructor's prototype property.
- Formal getter and setter method syntax, which replaces ActionScript 1.0's
- Formal property-definition syntax, used to create *instance properties and class properties* in a class body. In ActionScript 1.0, instance properties could be added in several ways—via the class constructor's prototype property, in the constructor function, or on each object directly.
- The *private* and *public* keywords, used to prevent certain methods and properties from being accessed outside of a class.

Key ActionScript 2.0 Features introduced are: (contd...)

- *Static typing* for variables, properties, parameters and return values used to declare the datatype for each item
- *Type casting*, used to tell the compiler to treat an object as though it were an instance of another datatype, as is sometimes required when using static typing.
- Classpaths, used to define the location of one or more central class repositories. This allows classes to be reused across projects and helps make source files easy to manage.
- *Exception handling* —including the *throw* and *try/catch/finally* statements—used to generate and respond to program errors.
- Easy linking between *movie clip symbols and ActionScript 2.0* classes via the symbol Linkage properties.

Features Introduced by Flash Player 7

- In addition to the ActionScript 2.0 language enhancements, Flash Player 7 introduces some important new classes and capabilities. They are:
- New *array-sorting capabilities*
- The ContextMenu and ContextMenuItem classes for customizing the Flash Player context menu
- *ID3 v2 tag* support for loaded MP3 files
- *Mouse wheel* support in text fields (Windows only)
- *Improved MovieClip* depth management methods
- The *MovieClipLoader* class for loading movie clips and images

Features Introduced by Flash Player 7 (Contd…)

- The *PrintJob class* for printing with greater control than was previously possible
- Support for *images in text fields*, including flowing text around images
- Improved *text metrics* (the ability to obtain more accurate measurements of the text in a text field than was possible in Flash Player 6)
- *Cascading stylesheet* (CSS) support for text fields, allowing the text in a movie to be formatted with a standard CSS stylesheet
- Improved *ActionScript runtime performance*
- Strict *case sensitivity*

Object Oriented ActionScript

Procedural Programming & Object-Oriented Programming

- Traditional programming consists of various instructions grouped into *procedures*.
- Procedures perform a *specific task* without any knowledge of or concern for the larger program.
- In a procedural-style Flash program, repeated tasks are stored in *functions* and data is stored in *variables*.
- The program runs by *executing functions* and *changing variable values*, typically for the purpose of handling input and generating output.
- Procedural programming is *sensible* for certain applications; however, as applications become larger or more *complex and the interactions between procedures*
- They can be *hard* to maintain, debug, and upgrade.

Procedural Prog. & Object-Oriented Prog. (Contd…)

- OOP is designed to make **complex applications** more manageable by breaking them down into self-contained, interacting modules.
- OOP lets us translate **abstract concepts** and tangible real-world things into corresponding parts of a program
- OOP adds a **level of conceptual organization** to a program.
- It groups related functions and variables together into separate ***classes*** , each of which is a *self-contained* part of the program with its own responsibilities.
- Classes are used to ***create individual objects*** that execute functions and set variables on one another, producing the program's behavior.
- Organizing the code into classes makes it easier to create a program that maps well to real-world problems with real-world components

Key Object-Oriented Programming Concepts

- An *object* is a self-contained software module that contains related ***functions*** (called its *methods*) and ***variables*** (called its *properties*).
- Individual objects are ***created from classes***, which provide the ***blueprint*** for an object's methods and properties
- A single class can be used to generate *any number of objects*, each with the same general structure
- To build an object-oriented program:
  - *Create one or more classes.*
  - *Make (i.e., instantiate) objects from those classes.*
  - *Tell the objects what to do.*

Key Object-Oriented Prog. Concepts (Contd…)

- We create, a program can use any of the classes built into the Flash Player.
- The built-in *Sound* class to create *Sound* objects, and use setVolume() & loadSound() methods.

Class Syntax

```
class SpaceShip {
public var speed:Number;
private var damage:Number;
public function SpaceShip ( )
{
 speed = 100;
 damage = 0; }
public function fireMissile ( ):Void
{  // Code that fires a missile goes here. }
public function thrust ( ):Void
{  // Code that propels the ship goes here. }
}
```

Key Object-Oriented Prog. Concepts (Contd…)

Object Creation

- *Objects are created (instantiated) with the new operator, as in:*
- *new ClassName( )*
  where ClassName is the name of the class from which the object will be created.
- *Ex. new SpaceShip( )*
- *var ship:SpaceShip = new SpaceShip( );*
- Each object is a discrete data value that can be stored in a variable, an array element, or even a property of another object.

Key Object-Oriented Prog. Concepts (Contd…)

Object Usage

- An object's methods provide its ***capabilities (i.e., behaviors)*** —things like "fire missile," "move," and "scroll down."
- Methods and properties that are defined as ***public*** by an object's class can be accessed from anywhere in a program.
- By contrast, methods and properties defined as ***private*** can be used only within the source code of the class or its subclasses.
- To invoke a method, we use the dot operator (i.e., a period) and the function call operator (i.e., parentheses).
- For example: ***ship.fireMissile( );***
- To set a property, we use the dot operator and an equals sign. For example: ***ship.speed = 120;***
- To retrieve a property's value, we use the dot operator on its own. For example: ***trace(ship.speed);***

Key Object-Oriented Prog. Concepts (Contd…)
Encapsulation

- Objects are said to *encapsulate* their *property values and method source code from the rest of the program.*
- *An object's **private properties** and the internal code used in its methods (including public methods)*
- It allows different programmers to work on different classes ***independently***
- A class can be tested thoroughly before being deployed. The same test code can be used to ***reverify*** the class's operation even if the code within the class is ***refactored***

Key Object-Oriented Prog. Concepts (Contd…)

Datatypes

- Each class in an object-oriented program can be thought of as defining a ***unique kind of data***, which is formally represented as a *datatype* in the program.
- *Date* datatype supports various ***properties and methods*** uniquely associated with dates
- Datatypes are used to impose limits on what can be stored in a variable, used as a parameter, or passed as a return value.
- Example: *public var speed:Number*

Key Object-Oriented Prog. Concepts (Contd…)

Inheritance

- Allow one class to ***adopt*** the method and property definitions of another
- Many classes can ***reuse*** the features of a single class
- For example, specific *Car*, *Boat*, and *Plane* classes could reuse the features of a generic *Vehicle* class, thus reducing redundancy in the application
- A class that **inherits** properties and methods from another class is called a ***subclass***.
- The class from which a subclass inherits properties and methods is called the subclass's ***superclass***.
- Naturally, a *subclass can define its own properties and methods in addition* to those it inherits from its superclass.

Key Object-Oriented Prog. Concepts (Contd…)
Packages

- we can create *packages* to contain **groups of classes**. A package lets us organize classes into logical groups and *prevents naming conflicts* between classes.
- **COMPILATION**
- When an OOP application is exported as a Flash movie (i.e., a .swf file), each class is *compiled*; that is, the compiler attempts to convert each class from source code to ***bytecode***—instructions that the Flash Player can understand and execute.
- If a class contains errors, compilation fails and the Flash compiler displays the *errors in the Output panel* in the Flash authoring tool.
- Even if the movie compiles successfully, errors may still occur while a program is running; these are called **runtime errors**

Key Object-Oriented Prog. Concepts (Contd…)

Starting an Objected-Oriented Application

- Create one or more classes in ***.as files.***
- Create a ***.fla*** file.
- On ***frame 1*** of the .fla file, add code that creates an object of a class.
- Optionally ***invoke a method*** on the object to start the application.
- ***Export*** a ***.swf*** file from the .fla file.
- ***Load*** the ***.swf*** file into the Flash Player.

HOW TO APPLY OOP

- Flash supports **both procedural and object-oriented programming** and allows you to combine both approaches in a single Flash movie.
- The fundamental organizing structure of a Flash document (a .fla file) is the ***timeline*** , which contains one or more *frames*.
- Each frame defines the content that is displayed on the graphical canvas called the ***Stage***.
- In the Flash Player, frames are displayed one at a time in a *linear sequence, producing an animated effect*—exactly like the frames in a ***filmstrip***.
- Timeline-based project containing predetermined animated sequences

- Consider using OOP when creating:
- Traditional desktop-style applications with few transitions and standardized user interfaces
- Applications that include server-side logic
- Functionality that is reused across multiple projects
- Components
- Games
- Highly customized user interfaces that include complex visual transitions

- Consider using procedural programming when creating:
- Animations with small scripts that control flow or basic interactivity
- Simple applications such as a one-page product order form or a newsletter subscription form
- Highly customized user interfaces that include complex visual transitions

Introduction : Datatypes and Type Checking

- ActionScript 2.0 defines a wide variety of datatypes.
- Some datatypes are native to the language itself ***(e.g., String, Number, and Boolean).***
- Others are included in the ***Flash Player*** and are available throughout all Flash movies ***(e.g., Color, Date, and TextField).***
- Still other datatypes are defined by components that can be added individually to Flash movies ***(e.g., List, RadioButton, and ScrollPane).***
- ActionScript 2.0 belongs to a datatype, whether built-in or programmer-defined.

  Ex. MovieClip class getTime( ), gotoAndPlay( ).

Introduction: Datatypes and Type Checking (Contd..)

```
var today;
today = new Date( ).getDay( );
// Sunday is 0, Monday is 1, ... Friday is 5.
if (today == 5) {
  trace("Looking forward to the weekend!");
}
```

- To help us recognize and isolate datatype-related problems in our code, we use ActionScript 2.0's **type checking capabilities.**
- ActionScript 2.0 requires that you formally declare the datatype of ***every variable, property, parameter, and return value*** that you want checked.

Introduction: Datatypes and Type Checking (Contd..)

- To declare the datatype of a variable or property, we use this general form, referred to as *post-colon syntax*:

  var variableOrPropertyName:datatype

  var count:Number;

- ActionScript 2.0 performs type checking on every variable, property, parameter, and return value that has a declared datatype.

  var today;

  today = new Date( ).getDay( );

- The preceding code simply stores the return value of *getDay( )* into today. Because the return value of *getDay( )* is a ***number, today stores a number, not a string.***

Introduction: Datatypes and Type Checking (Contd..)

- **Fixing a datatype mismatch error**

```
var today:Number
today = new Date( ).getDay( );
// Sunday is 0, Monday is 1, ... Friday is 5.
if (today == 5) {
  trace("Looking forward to the weekend!"); }
```

- **One way to derive a string from a number**

```
var today:Number = new Date( ).getDay( );
var dayNames:Array = ["Sunday", "Monday", "Tuesday",
   "Wednesday", “Thursday", “Friday", "Saturday"];
var todayName:String = dayNames[today];
currentDay_txt.text = todayName;
trace(todayName);
```

Introduction: Datatypes and Type Checking (Contd..)

- ActionScript 2.0's approach to datatyping is called **static typing .**
- In static typing, the datatype of variables and other data containers is ***fixed at compile time*** so that the compiler can guarantee the *validity of every method called and prop*erty accessed in the program.
- The converse of static typing is **dynamic typing** , in which ***each value is associated with a datatype at runtime***, not at compile time.
- With dynamic typing, data containers such as variables can change datatypes at runtime because type information is *associated with each value*, not with each data container

Why Static Typing?

- If a *type mismatch error occurs*, a program may be ***able to run***, but it wouldn't likely be able to perform one or more requested operations.
- Such errors often lead to program failures. Some compilers merely warn of type mismatch errors but still ***allow compilation to continue.***
- However, ActionScript 2.0's static typing facilities prevent the movie from ***compiling if type mismatch errors exist.***
- Type checking guarantees the validity of most operations at ***compile time***, *catching potential errors before they ever occur.*

Why Static Typing (Contd..)

- Type checking ***reduces*** the amount of ***manual data-verification*** code you need in your methods
- Dynamic type checking is often less restrictive and can make code easier to change than statically typed code.
- The main advantage of static type checking is that it can *check all your code at compile time*, whereas dynamic type checking requires that you *execute the code* in order to verify it.

Type Syntax

- ActionScript 2.0's compile-time type checking can be applied to:
  - ***A variable***
  - ***A property***
  - ***A function or method parameter***
  - ***A function or method return value***
- To enable type checking for any of the preceding items, we declare the item's type using ***post-colon syntax***
- Declaring an item's type tells the *compiler what kind of data it should contain.*

Type Syntax (Contd…)

- The compiler can warn us of two possible error conditions:
- If a value of an incompatible type is ***stored in a variable, passed as a function parameter***, or returned by a function, the compiler generates a *type* **mismatch error**.
- If a nonexistent property or method is accessed through a typed variable, function parameter, or function return value, the ***compiler generates an error*** explaining that the property or method cannot be found

Declaring Variable and Property Datatypes

- To declare the datatype of a variable or property, use the following syntax:

var variableOrPropertyName:datatype;

Ex1.Create a variable that can contain only data compatible with the *Date* datatype.

var currentTime:Date;

EX2. Store a *Date* instance in the *currentTime* variable.

currentTime = new Date( );

- We could also reduce the preceding two lines to a single step:

var currentTime:Date = new Date( );

Declaring Variable and Property Datatypes (Contd..)

- Once a variable's datatype is declared, it is *fixed until the variable is destroyed.*
- Type checking is intended to help you write better code more quickly

```
var currentTime:Date;
var currentTime:Number;
```

- In ActionScript 2.0, the preceding code does not change the datatype of currentTime to *Number*.
- The *second line is simply ignored* and later attempts to assign a numeric value to currentTime will *generate errors.*
- Instead of ***redeclaring a variable's datatype***, you should use ***two separate variables*** when you need to store values of two different datatypes.

Declaring Method Parameter and Return Value Datatypes

- Declaring the datatype of method or function parameters and return values:

```
function methodName (param1Name:param1Type,
   param2Name:param2Type):returnType {
  // ...
}
```

- Functions that return no value should specify a *returnType* of *Void*.
- Storing the return value of a function or method in a variable or property of an *incompatible type causes a type mismatch error.*

Post-Colon Syntax

Ex:

```
function sum (x:Number, y:Number):Number {
return x+y;
}
var result:SClass = sum(10, 20);
```

- ActionScript 2.0 uses the following, slightly unusual syntax for type declarations:

  ***variableOrPropertyName*:*datatype* = *value*;**

- By contrast, Java and C++ use:

  ***datatype variableName = value*;**

Compatible Types

- A variable of one type can store only a value of a **compatible type**.
- A type, *X*, is compatible with another type, *Y*, if *X* is of type *Y*, or if *X* is any subtype of *Y*.

  Ex: ***class Ball and a subclass Basketball.***

  var ball1:Ball = new Basketball( ); // Legal!
- Every *Ball* instance does not necessarily have the properties and methods of the *Basketball* class. *(Compile time type mismatch error)*

  var ball2:Basketball = new Ball( ); // Illegal!

Handling Any Datatype

- In ActionScript 2.0, we can make a variable, property, parameter or return value accept data of any type by specifying ***Object*** as the datatype.
- We declare the datatype of container as **Object.** We can subsequently store ***an instance of any class*** in it without error.

```
var container:Object = new Date( );     // No error.
container = new Color( );               // No error.
```

- This technique works because the Object class is the superclass of all *ActionScript classes*, so all types are compatible with the *Object* type

```
trace(container.toString( )); // Execute toString( ) normally.
container.blahblah( );// Invoke nonexistent method. No error.
```

- *toString( )method* excutes, because it is supported by all objects.

Compatibility with null and undefined

- AS 2.0 program store ***null or undefined in a variable*** as an indication of an *absence of data or an uninitialized variable*.

```
var target:MovieClip = null;  // Legal.
function square (x:Number):Number {
  return x*x;
}
square(null);  // Legal.
 function square (x:Number):Number {
  if (x == 0) {
    return null;  // Legal.
  }
  return x*x;
}
```

Compatibility with null and undefined (Contd..)

- This flexibility allows us to use the ***null type to indicate an empty value*** for any data container.

  var tf:TextFormat = new TextFormat(null, null, null, true);

- Compatibility with ***undefined*** also allows ActionScript to assign undefined to parameters, variables, and properties that have never been assigned a value.

```
public function displayMsg (msg:String,
                            sentBy:String):Void {
// Use "Anonymous" if a name was not supplied.
  if (sentBy == undefined) {
    sentBy = "Anonymous";   }
  // Display the message in a text field.
  output.text = sentBy + ": " + msg;
}
```

Built-in Dynamic Classes

- To allow new properties and methods to be added to a class's instances without **generating a compile-time error**
- We can define your own dynamic classes, but some built-in classes are dynamic by default.

  Array, ContextMenu, ContextMenuItem, TextField, Function,FunctionArguments ,LoadVars, MovieClip, Object
- When we attempt to access a nonexistent property or method on an object of one of the preceding dynamic classes, the ActionScript 2.0 compiler does not generate an error.

```
var dataSender:LoadVars = new LoadVars( );
dataSender.firstName = "Rebecca";  // No error
(Type mismatch)  var list:Array = new Date( );
```

Casting

- Casting is used to tell the compiler the ***type of an object when the compiler*** can't determine the object types on it's own.
- Syntax : ***Type(object);***
  - Where object is the object to cast and Type is the datatype.
    - ***E.g.: var obj:Object =new Object();***

      var tf:TextField = new TextField(obj);
- Casting doesn't convert an object from one class to another. It tells the compiler to treat the object or datum as though it were an instance of the specified datatype. It use a cast to tell the compiler the type of an object .

Casting (Contd…)

```
var ship: EnemyShip =theEnemyManager.getClosestShip();
  if(ship instanceof Bomber){
  Bomber(ship).bomb (); // cast to bomber
  }
  else if(ship instanceof Cruiser){
  Cruiser(ship).evade();
  }
  else if(ship instanceof Fighter){
  Fighter(ship).callReinforcements();
  Fighter(ship).fire();
  }
```

Casting Terminology

- Casting an object to one of its supertypes is known as ***upcast.***

  E.g.: var bball:BasketBall =new BasketBall();

  var genericball:Ball =new Ball(bball); // upcast bball

- Casting an object one of its subtypes is known as ***downcast.***

  E.g.: var b:Ball =new Ball();

  var bball:BasketBall =new BasketBall(b); // downcast of b

- An upcast is ***safe*** and downcast is ***unsafe***. Because supertype is not necessarily be an instance of its subtype.

  E.g.: var ball1: Ball =new BasketBall(); // without casting

  var ball1 : Ball=Ball(new BasketBall()); // safe upcast

Casting does not affect method or Property selection

- If a superclass define a method or property i.e., overhidden by a subclass, casting does not affect which version of a method or property to be used.
- When an *overridden* method is invoked on an instance of a subclass, the subclass version of the method is always used, even if the subclass instance is cast to the superclass.

Casting versus Conversion

- ***Casting*** merely tells the compiler to treat an object as though it were an instance of the specified datatype.
- ***Conversion*** does change an object from one datatype to another.

Casting to String, Number, Boolean, Date & Array

- The String(), Number(), Boolean() and Array() functions perform ***data conversions***
- The Date() function returns the ***current time as a string***
- Syntax: ***Number(value)*** // to convert a value to a number

E.g: Var obj:Object=[1,2,3];

```
function output (msg:Object):Void {
        // use typeof to detect the datatype of primitive values
if ( typeof msg==“string”)          {          trace(msg);    }
        // use Instanceof to check the class of an object
if (msg instanceof Array){
        trace(msg.join(“\n”));          }}
```

Output panel display 1 2 3 (line by line)

Defining Classes

- A *class* is a **template** for the creation of ***objects***. Classes are the building blocks of an object-oriented program.
- *class* declaration:

class *ClassIdentifier* { }

where ClassIdentifier is the name of the class

E.x.: class Box {}

- A class definition must reside in an external plain text file that has the extension .as.
- Everything between the curly braces in a *class* declaration constitutes the *class definition block*, or more informally, the *class body*.

The class body can contain:

- **A constructor function** (used to initialize instances of the class)
- **Variable definitions** (the class's properties)
- **Function definitions** (the class's methods)
- **#include directives,** which can include files containing properties, methods, and constructor functions
- **Metadata tags** used in Flash MX 2004 components
- **Comments**
- Class definitions cannot be nested, and no other code can appear in a class definition.

Example

```
class Box {
  private static var inited:Boolean = false;
  public function Box ( ) {
    if (!inited) {      init( );    }  }
 private static function init( ):Void {
    if (100 == (10*10)) {
      trace("One hundred is ten times ten");    }
    inited = true;  }}
```

- The term ***instance*** *member* refers to either an *instance* ***property or an instance method***, while the term *class member* refers to either a ***class property or a class method***

Constructor Functions (Take 1)

- When we create an object, *we also want to initialize it.*
- Ex. Box Class: Initialize the new instance's size
  (i.e., set its width and height properties)
- Represent the new instance on screen
  (i.e., call the *draw( )* method)
- To initialize and perform setup tasks for new objects of a class, we create a ***constructor function***.
- The constructor function executes ***automatically each time an instance is created.***
- Typically, when we create a class, we immediately add an empty constructor function to it.

Constructor Functions (Take 1) (Contd…)

- class Box {

  public function Box ( ) {

  }}

- Now that our *Box* class has an empty constructor function, we'll give the class some properties and methods.
- When no constructor is provided for a class, ActionScript adds an empty one automatically at compile time.

Properties

- Classes use *properties* to store information. But some information relates to a class as a whole

Class properties

- Named data containers associated with a class

Instance properties

- Named data containers associated with objects (i.e., instances of the class)
- Together, class properties and instance properties are sometimes referred to as ***fixed properties***
- In ActionScript and ECMAScript 4, instance properties are sometimes called ***instance variables***, and class properties are sometimes called ***class variables***

Instance properties:

- Are *stored individually* on each instance of a class
- Are accessed through *instances only*
- Can be set *uniquely* for one instance without affecting any other instances
- An instance property is *declared once for the entire class*, but each instance of the class maintains its own value.
- The general syntax is:

var propertyName:datatype = value;

- If *datatype* is ***omitted***, no type checking is performed on the property
- var *propertyName* = *value*;

// No type checking for this property.

Access control modifiers in AS, Java, & C++

Access control modifier	ActionScript	Java	C++
public	No access restrictions	No access restrictions	No access restrictions
private	Class and subclass access only	Class access only	Class access only
no modifier	Same as public	Access allowed from class's own package only	Same as private
protected	Not supported	Access allowed from class's own package and class's subclasses in any other package	Access allowed from class and subclass only

Compile-Time Constant Expressions

- An instance property definition can assign a default value to a property, provided that the value is a so-called ***compile-time constant expression*.**
- A compile-time constant expression is an expression whose value can be *fully determined at compile time*. They include :
- ***null, numeric, boolean, and string constants***
- The following operators (used only on numbers, booleans, strings, null, or undefined): *+ (unary and binary), - (unary and binary), ~, !, *, /, %, <<, >>, >>>, <, >, <=, >=, instanceof, ==, !=, ===, !==, &, ^, |, &&, ^^, ||, and ?: (ternary operator)*
- ***Array literals, Object literals, and instances*** of the following classes: ***Array, Boolean, Number, Object, and String***
- References to other ***compile-time constant expressions***
- Example: *4 + 5, "Hello" + " world", null, [4, 5], new Array(4, 5)*

Methods

- Methods are functions that determine the ***behavior of a class***.
- If a parameter's *type* is ***omitted*** in the method definition, *no type checking is performed for that parameter* .
- If *returnType* is omitted, ***no type checking is performed for the return value.***
- A function with return type ***Void*** may ***not return a value***.

```
class ClassName {
function methodName (param1:type,
param2:type,...paramn:type):returnType {
            statements          }    }

Ex. function square (x:Number):Number {
  return x * x; }
```

Referring to the Current Object with the Keyword this

- Using this helps our class to ***remain encapsulated***, eliminating the need for the outside world to worry about the object's internal requirements.
- The *this* reference to the ***current object*** allows an object to refer to itself without the need for an additional parameter, and is therefore much cleaner.

Passing the current object to a method

- The this keyword is most often used when passing the current object to another object's method

```
function someMethod ( ):Void {
 // Pass this as a parameter to a completely separate object
someOtherObject.someOtherMethod(this);  }
```

Garbage collection

- When an object registers as a *listener of another object*, it should *always unregister itself before it is deleted*.
- This sloppiness can cause *serious waste of memory*. A class should always provide a means of ***cleaning up*** stray object references ***before an object is deleted***.
- ActionScript doesn't garbage-collect an object (i.e., free up the memory used by an object) until no more references to it remain. Cleanup is done in a custom ***die( ) or destroy( )*** method that must be invoked before an object is deleted.
- **B**y providing a ***die( ) method***, a class guarantees a safe means of *deleting its instances*

```
function die ( ):Void {
    disableReset( );
var b:Box = new Box( );
b.enableReset( );
b.die( );       delete b;
```

Nesting Functions in Methods

- ActionScript supports *nested functions*, which means that functions can be declared within methods or even within other functions.
- A local variable, a nested function is accessible only to its parent function (the function in which it is declared). Code outside the parent function cannot execute the nested function:

```
class Box {
  private var width:Number;
  private var height:Number;
  public function getArea ( ):Number {
     return multiply(width, height);
     function multiply
           (a:Number, b:Number):Number {
       return a * b;
     }   }  }
```

Accessing the current object from a function nested in a method

- A function nested in a method does not have direct access to the current object ***(the object on which the method was called).***
- The current object by storing a reference to it in a ***local variable.***

```
public function debugDimensions ( ):Void {
  var boxObj:Box = this;
  setInterval(displayDimensions, 1000);
  function displayDimensions ( ):Void {
    trace("Width: " + boxObj.width + ",
                Height: " + boxObj.height);
  }}
```

Getter and Setter Methods

- To bridge the gap between the convenience of property assignment and the safety of accessor methods, *ActionScript 2.0 supports "getter" and "setter" methods.*
- Getter and setter methods are ***accessor-like methods***, defined within a class body, that are *invoked automatically when a developer tries to get or set a property directly*.

```
function get propertyName ( ):returnType {
                    // getter method   statements    }
function set propertyName (newValue:type):Void {
                    // setter method  statements   }
```

- Getter and setter methods ***cannot be declared with the private attribute.***
- When invoked getter and setter methods does not require use of the function call operator, ( ).

A class with getter and setter methods

```
class Box {
  private var width_internal:Number;
  private var height_internal:Number;
  public function get width ( ):Number {
    return width_internal;  }
  public function set width (w:Number):Void {
    width_internal = w;  }
  public function get height ( ):Number {
    return height_internal;  }
  public function set height (h:Number):Void {
    height_internal = h;  } }
var b:Box = new Box( );
b.width  = 20;     // Calls the width setter.
trace(b.width);    // Displays: 20
b.height = 10;     // Calls the height setter.
trace(b.height);   // Displays: 10
```

Extra or Missing Arguments

- Extra arguments are not (indeed cannot be) type checked automatically by the compiler.

```
public function setWidth (w:Number):Void {
 // If w is undefined, then quit
if (w == undefined) {
        return;        }
width = w;      }
```

- The arguments object also stores:
  - A reference to the method currently executing (arguments.callee)
  - A reference to the method that called the method currently executing, if any (arguments.caller)

4.5 Constructor Functions (Take 2)

- A constructor function is the ***metaphoric womb*** of a class; it isn't responsible for creating new instances, but it can be ***used to initialize each new instance***.
- When we create a new instance of a class using the new operator, the class's constructor function runs. Within the constructor function, we can customize the *newly created instance by setting its properties or invoking its methods*.

```
class Box {
        public function Box ( ) {
        } }
```

- To define a constructor function, we use the *function statement within a class body.*
- The constructor function's name ***must match its class's name exactly*** (case sensitivity matters).

Constructor Functions (Take 2) (Contd…)

- The constructor function's definition ***must not specify a return type*** (not even *Void*).
- The constructor function ***must not return a value***
- The constructor function's definition ***must not include the static attribute****, but it can be public or private.*
- ActionScript automatically provides a ***default constructor*** that takes ***no parameters and performs no initialization*** on new instances of the class.
- A constructor function's parameter values are passed to it via the **new operator at object-creation time**

.

new SomeClass(value1, value2,...valuen);

new Box(2, 3);

Simulating Multiple Constructor Functions

Unlike Java, ActionScript does not support multiple constructor functions for a single class (*overloaded constructors* in Java).

```
class Box {
  public var width:Number;
  public var height:Number;
    if (arguments.length == 0) {
      boxNoArgs( );     }
    else if (typeof a1 == "string") {
      boxString(a1);     }
    else if (typeof a1 == "number" && typeof a2 == "number") {
      boxNumberNumber(a1, a2);     }
    else {
      trace("Unexpected number of arguments to constructor.");
    }  }
  private function boxNoArgs ( ):Void {
    if (arguments.caller != Box) {
      return;     }
    width  = 1;
    height = 1;  }
```

overloaded constructors

```
private function boxString (size):Void {
  if (arguments.caller != Box) {
    return;     }
  if (size == "large") {
    width  = 100;        height = 100;
  } else if (size == "small") {
    width  = 10;         height = 10;
  } else {trace("Invalid box size specified"); }}
private function boxNumberNumber (w, h):Void {
  if (arguments.caller != Box) {
    return;     }
  width  = w;  height = h;  }}

// Usage:
var b1:Box = new Box( );
trace(b1.width);   // Displays: 1
var b2:Box = new Box("large");
trace(b2.width);   // Displays: 100
var b3:Box = new Box(25, 35);
trace(b3.width);   // Displays: 25
```

Completing the Box Class

- First decide whether the display elements will render themselves or be rendered by a central class.
- Decide how often the displayed elements should be updated—either when some event occurs (such as a mouseclick) or repeatedly (as fast as frames are displayed in the Flash Player).
- Decide on the rendering technique. Each visual element in a Flash movie must be displayed in a movie clip.
- Decide on a screen refresh strategy.

```
class Box {
  private var width:Number;
  private var height:Number;
  private var container_mc:MovieClip;
  public function Box (w:Number, h:Number, x:Number, y:Number,
                       target:MovieClip, depth:Number) {
    container_mc = target.createEmptyMovieClip
                       ("boxcontainer" + depth, depth);
    setWidth(w);     setHeight(h);
    setX(x);     setY(y);   }
```

A Box class complete with drawing routines

```
public function getWidth ( ):Number {
   return width;   }
public function setWidth (w:Number):Void {
   width = w;      draw( );   }
public function getHeight ( ):Number {
   return height;   }
public function setHeight (h:Number):Void {
   height = h;      draw( );   }

public function getX ( ):Number {
   return container_mc._x;   }

public function setX (x:Number):Void {
   container_mc._x = x;   }

public function getY ( ):Number {
   return container_mc._y;   }

public function setY (y:Number):Void {
   container_mc._y = y;   }
```

```
public function draw ( ):Void {
 container_mc.clear( );
 container_mc.lineStyle(1, 0x000000);
 container_mc.moveTo(0, 0);
 container_mc.beginFill(0xFFFFFF, 100);
 container_mc.lineTo(width, 0);
 container_mc.lineTo(width, height);
 container_mc.lineTo(0, height);
 container_mc.lineTo(0, 0);
 container_mc.endFill( );
 } }
// Usage: Code in flash document timeline
var b:Box = new Box(250, 260, 100, 110, this, 1);
b.setX(400);        b.setY(400);
b.setWidth(10);     b.setHeight(20);
trace(b.getX( ));          // Displays: 400
trace(b.getY( ));          // Displays: 400
trace(b.getWidth( ));      // Displays: 10
trace(b.getHeight( ));     // Displays: 20
```

A Box class complete with drawing routines (Contd..)

```
public function draw ( ):Void {
 container_mc.clear( );
 container_mc.lineStyle(1, 0x000000);
 container_mc.moveTo(0, 0);
 container_mc.beginFill(0xFFFFFF, 100);
 container_mc.lineTo(width, 0);
 container_mc.lineTo(width, height);
 container_mc.lineTo(0, height);
 container_mc.lineTo(0, 0);
 container_mc.endFill( );
 } }
// Usage: Code in flash document timeline
var b:Box = new Box(250, 260, 100, 110, this, 1);
b.setX(400);        b.setY(400);
b.setWidth(10);     b.setHeight(20);
trace(b.getX( ));           // Displays: 400
trace(b.getY( ));           // Displays: 400
trace(b.getWidth( ));       // Displays: 10
trace(b.getHeight( ));      // Displays: 20
```

Authoring An ActionScript 2.0 Class

Class Authoring Quick Start

- To create an *ActionScript 2.0 class*, follow these general steps:
- Create a new ***text file*** with the **.as** extension using any plain text editor or *Flash MX Professional 2004's built-in editor.*
- The .as file's name must ***match the class name exactly*** (case sensitivity matters).
- Add the class definition to the .as file.

```
class NameOfClass {
                    // Class body goes here
                    }
```

To use an ActionScript 2.0 class in a Flash movie, Steps:

1. Create a .fla file with any name, and ***place it in the same folder as the .as file*** from Step 1 in the preceding procedure.
2. Optionally specify the ***class's export frame*** for the .fla file via File Publish Settings Flash ActionScript Version Settings Export Frame for Classes. This determines when the class loads and when it becomes available in the movie. *The export frame is usually set to some frame after your movie's preloader.*
3. Use the *class as desired throughout the .fla file*, but after the export frame specified in Step 2 (if any).
4. Export a .swf file using one of the following: ***File Publish, Control Test Movie, or File Export Export Movie.***

Designing the ImageViewer Class

- With our *ImageViewer* class's name we have a list of functionality that could be required of the *ImageViewer* class:
  - *Load an image*
  - *Display an image*
  - *Crop an image to a particular rectangular "view region"*
  - *Display a border around the image*
  - *Display image load progress*
  - *Reposition the view region*
  - *Resize the view region*
  - *Pan (reposition) the image within the view region*
  - *Zoom (resize) the image within the view region*

ImageViewer Implementation (Take 1)

- To create the ImageViewer.as file using Flash MX Professional 2004, follow these steps:
- *Choose File New.*
- *In the New Document dialog box, on the General tab, for the document Type, choose ActionScript File.*
- *Click OK. The script editor launches with an empty file.*
- *Choose File Save As.*
- *In the Save As dialog box, specify **ImageViewer.as** as the filename (using upper- and lowercase as shown) and save the file in the imageviewer folder you created.*
- Notice that the class name, *ImageViewer*, and the filename, ImageViewer.as, ***must match exactly*** (apart from the .as file extension).

The ImageViewer class with constructor and loadImage( ) method, properties (Take 1)

```
class ImageViewer {
 private var container_mc:MovieClip;

public function ImageViewer (target:MovieClip, depth:Number)
{
    container_mc = target.createEmptyMovieClip
                   ("container_mc" + depth, depth);
}
public function loadImage (URL:String):Void
{
    container_mc.loadMovie(URL);
}}
```

Using ImageViewer in a Movie

- *Find a (nonprogressive format) JPEG image on your system.*
- *Name the JPEG image picture.jpg and place it in the same **folder as the ImageViewer.as** file you created earlier.*
- *Next we'll create a Flash document (.fla file) from which we'll publish a Flash movie (.swf file) containing an instance of our ImageViewer class.*
- *For now, we'll place the **.fla file, .as file, .swf file, and .jpg file all in the same folder,** making it easy for each file to access the other files.*
- Create the Flash document that uses the *ImageViewer* class. Follow these steps:
  - *In the Flash authoring tool, choose File -> New.*
  - *In the New Document dialog box, on the General tab, for the document Type, choose Flash Document, then click OK.*

Using ImageViewer in a Movie (Contd…)

- Use File -> Save As to save the Flash document as **imageViewer.fla** in the same folder as the ImageViewer.as file.
- In imageViewer.fla's main timeline, rename *Layer 1* to **scripts**
- Follow these steps to instantiate *ImageViewer* on frame 1 of imageViewer.fla's main timeline:
  - Use Window -> Development Panels ->Actions (F9) to open the Actions panel.
  - Select frame 1 in imageViewer.fla's main timeline.
  - Into the Actions panel, enter the following code:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

Using ImageViewer in a Movie (Contd…)

- Let's export our imageViewer.swf file and test it in Flash's Test Movie mode, as follows:
  - *Choose Control -> Test Movie. The .swf file should play, and your image should load and appear.*
  - *When you're finished marveling at your work, choose File -> Close to return to the imageViewer.fla file.*
- To change imageViewer.fla's export settings to support playback in Flash Player 6, follow these steps:
  - *Choose File -> Publish Settings.*
  - *On the Flash tab of the Publish Settings dialog box, select Flash Player 6 as the Version option.*
  - *Click OK.*

Preloading the ImageViewer Class

- Let's change our imageViewer.fla file so that the classes it uses aren't loaded until frame 10:
- *Choose File -> Publish Settings.*
- *In the Publish Settings dialog box, on the Flash tab, next to the ActionScript Version, click Settings.*
- *In the ActionScript Settings dialog box, for Export Frame for Classes, enter **10**.*
- *Click OK to confirm the ActionScript Settings.*
- *Click OK to confirm the Publish Settings.*
- Now let's add a very basic preloader to our imageViewer.fla file so load progress is reported while the *ImageViewer* class loads. When loading is complete, we'll advance the playhead to frame 15 where we'll instantiate *ImageViewer* (as we previously did on frame 1).

Preloading the ImageViewer Class (Contd..)

- First, we'll make the timeline 15 frames long, as follows:
  - *In the main timeline of imageViewer.fla, select frame 15 of the scripts layer.*
  - *Choose Insert -> Timeline ->Keyframe (F6).*
- Next, we'll add a *labels* layer with two frame labels, loading and main. The labels designate the application's loading state and startup point, respectively.
- *Choose Insert ->Timeline ->Layer.*
- *Double-click the new layer's name and change it to **labels**.*
- *At frames 4 and 15 of the labels layer, add a new keyframe (using Insert ->Timeline ->Keyframe).*
- *With frame 4 of the labels layer selected, in the Properties panel, under Frame, change <Frame Label> to **loading**.*
- *With frame 15 of the labels layer selected, in the Properties panel, under Frame, change <Frame Label> to **main**.*

Now add the preloader script to the *scripts* layer:

- At frame 5 of the *scripts* layer, add a new keyframe (using Insert ->timeline ->Keyframe).
- With frame 5 of the *scripts* layer selected, enter the following code into the Actions panel:

```
if (_framesloaded == _totalframes) {
gotoAndStop("main");}
else {  gotoAndPlay("loading");   }
```

- Next, we'll move the code that creates our *ImageViewer* instance from frame 1 to frame 15 of the *scripts* layer:
- Select frame 1 of the *scripts* layer.
- In the Actions panel, cut (delete using Ctrl-X or Cmd-X) the following code from frame:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

- With frame 15 of the *scripts* layer selected, paste (using Ctrl-V) the code you deleted in Step 2 into the Actions panel.
- Finally, we'll add a loading message that displays while the *ImageViewer* class loads:
- With frame 1 of the *scripts* layer selected, enter the following code into the Actions panel:

```
this.createTextField("loadmsg_txt", 0, 200, 200, 0, 0);
loadmsg_txt.autoSize = true;
loadmsg_txt.text = "Loading...Please wait.";
```

- With frame 15 of the *scripts* layer selected, enter the following code at the end of the Actions panel

```
loadmsg_txt.removeTextField( );
```

ImageViewer Implementation (Take 2)

- The third and fourth requirements: ***cropping the image and giving it a border***. We must apply a ***mask*** to it, which hides unwanted areas of the image from view.
- We'll add parameters to the *ImageViewer* constructor to specify the *size of the mask and the position of the cropped image.*
- To create border, we'll *create a movie clip, then draw a square outline in it.*

Splitting out the work into separate methods has the following benefits:

- It makes the code more intelligible.
- It simplifies the testing process (testing each method individually is easier than testing a large block of code that performs many operations).
- It allows assets to be created and re-created independently (e.g., the border on an image can change without reloading the image).
- It allows asset-creation operations to be modified or overridden independently (e.g., a new border-creation routine can be coded without disturbing other code).

Table The ImageViewer class's new instance methods that create movie clips

Method name	*Method description*
buildViewer( )	*Invokes individual methods to create the container, image, mask, and border clips*
createMainContainer( )	*Creates the* `container_mc` *clip*
createImageClip( )	*Creates the* `image_mc` *clip*
createImageClipMask( )	*Creates the* `mask_mc` *clip*
createBorder( )	*Creates the* `border_mc` *clip*

- **A reference to the target clip to which container_mc should be attached**
- **A list of depths indicating the visual stacking order of the container, image, mask, and border clips**
- **The style (line weight and color) of the border around the image**

Table The ImageViewer class's instance and class properties

Property name	Type	Property description
`container_mc`	*Instance*	*A reference to the main container clip, which contains all movie clips used by each ImageViewer instance*
`target_mc`	*Instance*	*A reference to the clip that will contain the* `container_mc` *clip, as specified by the ImageViewer constructor*
`containerDepth`	*Instance*	*The depth on which* `container_mc` *is created in* `target_mc`*, as specified by the ImageViewer constructor*
`imageDepth`	*Class*	*The depth on which* `image_mc` *is created in* `container_mc`
`maskDepth`	*Class*	*The depth on which* `mask_mc` *is created in* `container_mc`
`borderDepth`	*Class*	*The depth on which* `border_mc` *is created in* `container_mc`
`borderThickness`	*Instance*	*The thickness, in pixels, of the border around the* `image_mc` *clip*
`borderColor`	*Instance*	*The integer RGB color of the border around the* `image_mc` *clip*

The ImageViewer class, take 2

```
class ImageViewer { // Movie clip references
  private var container_mc:MovieClip;
  Private var target_mc:MovieClip;
  // Movie clip depths
  private var containerDepth:Number;
  private static var imageDepth:Number  = 0;
  private static var maskDepth:Number   = 1;
  private static var borderDepth:Number = 2;
  private var borderThickness:Number;
  private var borderColor:Number;
  public function ImageViewer (target:MovieClip,
          depth:Number,x:Number,y:Number,
          w:Number,h:Number,borderThickness:Number,
          borderColor:Number) {// Assign property values.
    target_mc = target;
    containerDepth = depth;
    this.borderThickness = borderThickness;
    this.borderColor = borderColor;
    // Set up the visual assets for this ImageViewer.
    buildViewer(x, y, w, h);  }
```

The ImageViewer class, take 2

```
private function buildViewer (x:Number,
        y:Number,w:Number,h:Number):Void {
        createMainContainer(x, y); createImageClip( );
        createImageClipMask(w, h); createBorder(w, h);
}
private function createMainContainer (x:Number,
y:Number):Void { container_mc =
        target_mc.createEmptyMovieClip("container_mc" ,
containerDepth, containerDepth);
container_mc._x = x; container_mc._y = y;
}
private function createImageClip ( ):Void {
         container_mc.createEmptyMovieClip("image_mc",
         imageDepth);   }
private function createImageClipMask
          (w:Number,h:Number):Void {
      if (!(w > 0 && h > 0)) {
        return;
      }
```

The ImageViewer class, take 2

```
container_mc.createEmptyMovieClip("mask_mc",
maskDepth);
    container_mc.mask_mc.moveTo(0, 0);
    container_mc.mask_mc.beginFill(0x0000FF);
    container_mc.mask_mc.lineTo(w, 0);
    container_mc.mask_mc.lineTo(w, h);
    container_mc.mask_mc.lineTo(0, h);
    container_mc.mask_mc.lineTo(0, 0);
    container_mc.mask_mc.endFill( );
    container_mc.mask_mc._visible = false;
    }
  private function createBorder (w:Number,
                  h:Number):Void {
    if (!(w > 0 && h > 0)) {
      return;    }
```

The ImageViewer class, take 2

```
container_mc.createEmptyMovieClip("border_mc",
                                   borderDepth);
container_mc.border_mc.lineStyle(borderThickness,
                                   borderColor);
      container_mc.border_mc.moveTo(0, 0);
      container_mc.border_mc.lineTo(w, 0);
    container_mc.border_mc.lineTo(w, h);
    container_mc.border_mc.lineTo(0, h);
    container_mc.border_mc.lineTo(0, 0);
  }
  public function loadImage (URL:String):Void {
    container_mc.image_mc.loadMovie(URL);
    container_mc.onEnterFrame = function ( ):Void {
      this.image_mc.setMask(this.mask_mc);
      delete this.onEnterFrame;
    }  }
}
```

- Now that our *ImageViewer* class can crop an image and add a border to it.
- When we used the first version of the *ImageViewer* class, we placed the following code on frame 15 of imageViewer.fla:

Unit III

Action Script I

ActionScript 2.0 Features

- ActionScript 2.0 adds relatively little *new runtime functionality* to the language but radically improves object-oriented development in Flash by formalizing objected-oriented programming *(OOP) syntax and methodology*.

- ActionScript 2.0 provides a class keyword for creating classes and an *extends* keyword for establishing *inheritance*. Those keywords were absent from ActionScript 1.0

Key ActionScript 2.0 Features introduced are:

- The *class* statement, used to create formal classes.
- The *extends* keyword, used to establish inheritance.
- The *interface* statement, used to create Java-style interfaces (i.e., abstract data types).
  - Classes provide implementations for interfaces using the *implements* keyword. ActionScript 1.0 did not support interfaces.
- The official file extension for class files is ***.as***. Formerly, classes could be defined in timeline code or in external .as files.
  - Class files can be edited in Flash MX Professional 2004's script editor or in an external text editor.

Key ActionScript 2.0 Features introduced are: (contd...)

- Formal method-definition syntax, used to *create instance methods and class methods* in a class body. In ActionScript 1.0, methods were added to a class via the class constructor's prototype property.
- Formal getter and setter method syntax, which replaces ActionScript 1.0's
- Formal property-definition syntax, used to create *instance properties and class properties* in a class body. In ActionScript 1.0, instance properties could be added in several ways—via the class constructor's prototype property, in the constructor function, or on each object directly.
- The *private* and *public* keywords, used to prevent certain methods and properties from being accessed outside of a class.

Key ActionScript 2.0 Features introduced are: (contd...)

- *Static typing* for variables, properties, parameters and return values used to declare the datatype for each item
- *Type casting*, used to tell the compiler to treat an object as though it were an instance of another datatype, as is sometimes required when using static typing.
- Classpaths, used to define the location of one or more central class repositories. This allows classes to be reused across projects and helps make source files easy to manage.
- *Exception handling* —including the *throw* and *try/catch/finally* statements—used to generate and respond to program errors.
- Easy linking between *movie clip symbols and ActionScript 2.0* classes via the symbol Linkage properties.

Features Introduced by Flash Player 7

- In addition to the ActionScript 2.0 language enhancements, Flash Player 7 introduces some important new classes and capabilities. They are:
- New *array-sorting capabilities*
- The ContextMenu and ContextMenuItem classes for customizing the Flash Player context menu
- *ID3 v2 tag* support for loaded MP3 files
- *Mouse wheel* support in text fields (Windows only)
- *Improved MovieClip* depth management methods
- The *MovieClipLoader* class for loading movie clips and images

Features Introduced by Flash Player 7 (Contd…)

- The *PrintJob class* for printing with greater control than was previously possible
- Support for *images in text fields*, including flowing text around images
- Improved *text metrics* (the ability to obtain more accurate measurements of the text in a text field than was possible in Flash Player 6)
- *Cascading stylesheet* (CSS) support for text fields, allowing the text in a movie to be formatted with a standard CSS stylesheet
- Improved *ActionScript runtime performance*
- Strict *case sensitivity*

Object Oriented ActionScript

Procedural Programming & Object-Oriented Programming

- Traditional programming consists of various instructions grouped into *procedures*.
- Procedures perform a *specific task* without any knowledge of or concern for the larger program.
- In a procedural-style Flash program, repeated tasks are stored in *functions* and data is stored in *variables*.
- The program runs by *executing functions* and *changing variable values*, typically for the purpose of handling input and generating output.
- Procedural programming is *sensible* for certain applications; however, as applications become larger or more *complex and the interactions between procedures*
- They can be *hard* to maintain, debug, and upgrade.

Procedural Prog. & Object-Oriented Prog. (Contd…)

- OOP is designed to make **complex applications** more manageable by breaking them down into self-contained, interacting modules.
- OOP lets us translate **abstract concepts** and tangible real-world things into corresponding parts of a program
- OOP adds a **level of conceptual organization** to a program.
- It groups related functions and variables together into separate ***classes*** , each of which is a *self-contained* part of the program with its own responsibilities.
- Classes are used to ***create individual objects*** that execute functions and set variables on one another, producing the program's behavior.
- Organizing the code into classes makes it easier to create a program that maps well to real-world problems with real-world components

Key Object-Oriented Programming Concepts

- An *object* is a self-contained software module that contains related ***functions*** (called its *methods*) and ***variables*** (called its *properties*).
- Individual objects are ***created from classes***, which provide the ***blueprint*** for an object's methods and properties
- A single class can be used to generate *any number of objects*, each with the same general structure
- To build an object-oriented program:
  - *Create one or more classes.*
  - *Make (i.e., instantiate) objects from those classes.*
  - *Tell the objects what to do.*

Key Object-Oriented Prog. Concepts (Contd…)

- We create, a program can use any of the classes built into the Flash Player.
- The built-in *Sound* class to create *Sound* objects, and use setVolume() & loadSound() methods.

Class Syntax

```
class SpaceShip {
public var speed:Number;
private var damage:Number;
public function SpaceShip ( )
{
 speed = 100;
 damage = 0; }
public function fireMissile ( ):Void
{  // Code that fires a missile goes here. }
public function thrust ( ):Void
{  // Code that propels the ship goes here. }
}
```

Key Object-Oriented Prog. Concepts (Contd…)

Object Creation

- *Objects are created (instantiated) with the new operator, as in:*
- *new ClassName( )*
  where ClassName is the name of the class from which the object will be created.
- *Ex. new SpaceShip( )*
- *var ship:SpaceShip = new SpaceShip( );*
- Each object is a discrete data value that can be stored in a variable, an array element, or even a property of another object.

Key Object-Oriented Prog. Concepts (Contd…)

Object Usage

- An object's methods provide its ***capabilities (i.e., behaviors)*** —things like "fire missile," "move," and "scroll down."
- Methods and properties that are defined as ***public*** by an object's class can be accessed from anywhere in a program.
- By contrast, methods and properties defined as ***private*** can be used only within the source code of the class or its subclasses.
- To invoke a method, we use the dot operator (i.e., a period) and the function call operator (i.e., parentheses).
- For example: ***ship.fireMissile( );***
- To set a property, we use the dot operator and an equals sign. For example: ***ship.speed = 120;***
- To retrieve a property's value, we use the dot operator on its own. For example: ***trace(ship.speed);***

Key Object-Oriented Prog. Concepts (Contd…)
Encapsulation

- Objects are said to *encapsulate* their *property values and method source code from the rest of the program.*
- *An object's **private properties** and the internal code used in its methods (including public methods)*
- It allows different programmers to work on different classes ***independently***
- A class can be tested thoroughly before being deployed. The same test code can be used to ***reverify*** the class's operation even if the code within the class is ***refactored***

Key Object-Oriented Prog. Concepts (Contd…)

Datatypes

- Each class in an object-oriented program can be thought of as defining a ***unique kind of data***, which is formally represented as a *datatype* in the program.
- *Date* datatype supports various ***properties and methods*** uniquely associated with dates
- Datatypes are used to impose limits on what can be stored in a variable, used as a parameter, or passed as a return value.
- Example: *public var speed:Number*

Key Object-Oriented Prog. Concepts (Contd…)

Inheritance

- Allow one class to ***adopt*** the method and property definitions of another
- Many classes can ***reuse*** the features of a single class
- For example, specific *Car*, *Boat*, and *Plane* classes could reuse the features of a generic *Vehicle* class, thus reducing redundancy in the application
- A class that **inherits** properties and methods from another class is called a ***subclass***.
- The class from which a subclass inherits properties and methods is called the subclass's ***superclass***.
- Naturally, a *subclass can define its own properties and methods in addition* to those it inherits from its superclass.

Key Object-Oriented Prog. Concepts (Contd…)
Packages

- we can create *packages* to contain **groups of classes**. A package lets us organize classes into logical groups and *prevents naming conflicts* between classes.
- **COMPILATION**
- When an OOP application is exported as a Flash movie (i.e., a .swf file), each class is *compiled*; that is, the compiler attempts to convert each class from source code to ***bytecode***—instructions that the Flash Player can understand and execute.
- If a class contains errors, compilation fails and the Flash compiler displays the *errors in the Output panel* in the Flash authoring tool.
- Even if the movie compiles successfully, errors may still occur while a program is running; these are called **runtime errors**

Key Object-Oriented Prog. Concepts (Contd…)

Starting an Objected-Oriented Application

- Create one or more classes in ***.as files.***
- Create a ***.fla*** file.
- On ***frame 1*** of the .fla file, add code that creates an object of a class.
- Optionally ***invoke a method*** on the object to start the application.
- ***Export*** a ***.swf*** file from the .fla file.
- ***Load*** the ***.swf*** file into the Flash Player.

HOW TO APPLY OOP

- Flash supports **both procedural and object-oriented programming** and allows you to combine both approaches in a single Flash movie.
- The fundamental organizing structure of a Flash document (a .fla file) is the ***timeline*** , which contains one or more *frames*.
- Each frame defines the content that is displayed on the graphical canvas called the ***Stage***.
- In the Flash Player, frames are displayed one at a time in a *linear sequence, producing an animated effect*—exactly like the frames in a ***filmstrip***.
- Timeline-based project containing predetermined animated sequences

- Consider using OOP when creating:
- Traditional desktop-style applications with few transitions and standardized user interfaces
- Applications that include server-side logic
- Functionality that is reused across multiple projects
- Components
- Games
- Highly customized user interfaces that include complex visual transitions

- Consider using procedural programming when creating:
- Animations with small scripts that control flow or basic interactivity
- Simple applications such as a one-page product order form or a newsletter subscription form
- Highly customized user interfaces that include complex visual transitions

Introduction : Datatypes and Type Checking

- ActionScript 2.0 defines a wide variety of datatypes.
- Some datatypes are native to the language itself ***(e.g., String, Number, and Boolean).***
- Others are included in the ***Flash Player*** and are available throughout all Flash movies ***(e.g., Color, Date, and TextField).***
- Still other datatypes are defined by components that can be added individually to Flash movies ***(e.g., List, RadioButton, and ScrollPane).***
- ActionScript 2.0 belongs to a datatype, whether built-in or programmer-defined.

  Ex. MovieClip class getTime( ), gotoAndPlay( ).

Introduction: Datatypes and Type Checking (Contd..)

```
var today;
today = new Date( ).getDay( );
// Sunday is 0, Monday is 1, ... Friday is 5.
if (today == 5) {
  trace("Looking forward to the weekend!");
}
```

- To help us recognize and isolate datatype-related problems in our code, we use ActionScript 2.0's **type checking capabilities.**
- ActionScript 2.0 requires that you formally declare the datatype of ***every variable, property, parameter, and return value*** that you want checked.

Introduction: Datatypes and Type Checking (Contd..)

- To declare the datatype of a variable or property, we use this general form, referred to as *post-colon syntax*:

  var variableOrPropertyName:datatype

  var count:Number;

- ActionScript 2.0 performs type checking on every variable, property, parameter, and return value that has a declared datatype.

  var today;

  today = new Date( ).getDay( );

- The preceding code simply stores the return value of *getDay( )* into today. Because the return value of *getDay( )* is a ***number, today stores a number, not a string.***

Introduction: Datatypes and Type Checking (Contd..)

- **Fixing a datatype mismatch error**

```
var today:Number
today = new Date( ).getDay( );
// Sunday is 0, Monday is 1, ... Friday is 5.
if (today == 5) {
  trace("Looking forward to the weekend!"); }
```

- **One way to derive a string from a number**

```
var today:Number = new Date( ).getDay( );
var dayNames:Array = ["Sunday", "Monday", "Tuesday",
   "Wednesday", “Thursday", “Friday", "Saturday"];
var todayName:String = dayNames[today];
currentDay_txt.text = todayName;
trace(todayName);
```

Introduction: Datatypes and Type Checking (Contd..)

- ActionScript 2.0's approach to datatyping is called **static typing .**
- In static typing, the datatype of variables and other data containers is ***fixed at compile time*** so that the compiler can guarantee the *validity of every method called and prop*erty accessed in the program.
- The converse of static typing is **dynamic typing** , in which ***each value is associated with a datatype at runtime***, not at compile time.
- With dynamic typing, data containers such as variables can change datatypes at runtime because type information is *associated with each value*, not with each data container

Why Static Typing?

- If a *type mismatch error occurs*, a program may be ***able to run***, but it wouldn't likely be able to perform one or more requested operations.
- Such errors often lead to program failures. Some compilers merely warn of type mismatch errors but still ***allow compilation to continue.***
- However, ActionScript 2.0's static typing facilities prevent the movie from ***compiling if type mismatch errors exist.***
- Type checking guarantees the validity of most operations at ***compile time***, *catching potential errors before they ever occur.*

Why Static Typing (Contd..)

- Type checking ***reduces*** the amount of ***manual data-verification*** code you need in your methods
- Dynamic type checking is often less restrictive and can make code easier to change than statically typed code.
- The main advantage of static type checking is that it can *check all your code at compile time*, whereas dynamic type checking requires that you *execute the code* in order to verify it.

Type Syntax

- ActionScript 2.0's compile-time type checking can be applied to:
  - ***A variable***
  - ***A property***
  - ***A function or method parameter***
  - ***A function or method return value***
- To enable type checking for any of the preceding items, we declare the item's type using ***post-colon syntax***
- Declaring an item's type tells the *compiler what kind of data it should contain.*

Type Syntax (Contd…)

- The compiler can warn us of two possible error conditions:
- If a value of an incompatible type is ***stored in a variable, passed as a function parameter***, or returned by a function, the compiler generates a *type* **mismatch error**.
- If a nonexistent property or method is accessed through a typed variable, function parameter, or function return value, the ***compiler generates an error*** explaining that the property or method cannot be found

Declaring Variable and Property Datatypes

- To declare the datatype of a variable or property, use the following syntax:

var variableOrPropertyName:datatype;

Ex1.Create a variable that can contain only data compatible with the *Date* datatype.

var currentTime:Date;

EX2. Store a *Date* instance in the *currentTime* variable.

currentTime = new Date( );

- We could also reduce the preceding two lines to a single step:

var currentTime:Date = new Date( );

Declaring Variable and Property Datatypes (Contd..)

- Once a variable's datatype is declared, it is *fixed until the variable is destroyed.*
- Type checking is intended to help you write better code more quickly

```
var currentTime:Date;
var currentTime:Number;
```

- In ActionScript 2.0, the preceding code does not change the datatype of currentTime to *Number*.
- The *second line is simply ignored* and later attempts to assign a numeric value to currentTime will *generate errors.*
- Instead of ***redeclaring a variable's datatype***, you should use ***two separate variables*** when you need to store values of two different datatypes.

Declaring Method Parameter and Return Value Datatypes

- Declaring the datatype of method or function parameters and return values:

```
function methodName (param1Name:param1Type,
   param2Name:param2Type):returnType {
  // ...
}
```

- Functions that return no value should specify a *returnType* of *Void*.
- Storing the return value of a function or method in a variable or property of an *incompatible type causes a type mismatch error.*

Post-Colon Syntax

Ex:

```
function sum (x:Number, y:Number):Number {
return x+y;
}
var result:SClass = sum(10, 20);
```

- ActionScript 2.0 uses the following, slightly unusual syntax for type declarations:

  ***variableOrPropertyName*:*datatype* = *value*;**

- By contrast, Java and C++ use:

  ***datatype variableName = value*;**

Compatible Types

- A variable of one type can store only a value of a **compatible type**.
- A type, *X*, is compatible with another type, *Y*, if *X* is of type *Y*, or if *X* is any subtype of *Y*.

  Ex: ***class Ball and a subclass Basketball.***

  var ball1:Ball = new Basketball( ); // Legal!
- Every *Ball* instance does not necessarily have the properties and methods of the *Basketball* class. *(Compile time type mismatch error)*

  var ball2:Basketball = new Ball( ); // Illegal!

Handling Any Datatype

- In ActionScript 2.0, we can make a variable, property, parameter or return value accept data of any type by specifying ***Object*** as the datatype.
- We declare the datatype of container as **Object.** We can subsequently store ***an instance of any class*** in it without error.

```
var container:Object = new Date( );     // No error.
container = new Color( );               // No error.
```

- This technique works because the Object class is the superclass of all *ActionScript classes*, so all types are compatible with the *Object* type

```
trace(container.toString( )); // Execute toString( ) normally.
container.blahblah( );// Invoke nonexistent method. No error.
```

- *toString( )method* excutes, because it is supported by all objects.

Compatibility with null and undefined

- AS 2.0 program store ***null or undefined in a variable*** as an indication of an *absence of data or an uninitialized variable*.

```
var target:MovieClip = null;  // Legal.
function square (x:Number):Number {
  return x*x;
}
square(null);  // Legal.
 function square (x:Number):Number {
  if (x == 0) {
    return null;  // Legal.
  }
  return x*x;
}
```

Compatibility with null and undefined (Contd..)

- This flexibility allows us to use the ***null type to indicate an empty value*** for any data container.

  var tf:TextFormat = new TextFormat(null, null, null, true);

- Compatibility with ***undefined*** also allows ActionScript to assign undefined to parameters, variables, and properties that have never been assigned a value.

```
public function displayMsg (msg:String,
                            sentBy:String):Void {
// Use "Anonymous" if a name was not supplied.
  if (sentBy == undefined) {
    sentBy = "Anonymous";   }
  // Display the message in a text field.
  output.text = sentBy + ": " + msg;
}
```

Built-in Dynamic Classes

- To allow new properties and methods to be added to a class's instances without **generating a compile-time error**
- We can define your own dynamic classes, but some built-in classes are dynamic by default.

  Array, ContextMenu, ContextMenuItem, TextField, Function,FunctionArguments ,LoadVars, MovieClip, Object
- When we attempt to access a nonexistent property or method on an object of one of the preceding dynamic classes, the ActionScript 2.0 compiler does not generate an error.

```
var dataSender:LoadVars = new LoadVars( );
dataSender.firstName = "Rebecca";  // No error
(Type mismatch)  var list:Array = new Date( );
```

Casting

- Casting is used to tell the compiler the ***type of an object when the compiler*** can't determine the object types on it's own.
- Syntax : ***Type(object);***
  - Where object is the object to cast and Type is the datatype.
    - ***E.g.: var obj:Object =new Object();***

      var tf:TextField = new TextField(obj);
- Casting doesn't convert an object from one class to another. It tells the compiler to treat the object or datum as though it were an instance of the specified datatype. It use a cast to tell the compiler the type of an object .

Casting (Contd…)

```
var ship: EnemyShip =theEnemyManager.getClosestShip();
  if(ship instanceof Bomber){
  Bomber(ship).bomb (); // cast to bomber
  }
  else if(ship instanceof Cruiser){
  Cruiser(ship).evade();
  }
  else if(ship instanceof Fighter){
  Fighter(ship).callReinforcements();
  Fighter(ship).fire();
  }
```

Casting Terminology

- Casting an object to one of its supertypes is known as ***upcast.***

  E.g.: var bball:BasketBall =new BasketBall();

  var genericball:Ball =new Ball(bball); // upcast bball

- Casting an object one of its subtypes is known as ***downcast.***

  E.g.: var b:Ball =new Ball();

  var bball:BasketBall =new BasketBall(b); // downcast of b

- An upcast is ***safe*** and downcast is ***unsafe***. Because supertype is not necessarily be an instance of its subtype.

  E.g.: var ball1: Ball =new BasketBall(); // without casting

  var ball1 : Ball=Ball(new BasketBall()); // safe upcast

Casting does not affect method or Property selection

- If a superclass define a method or property i.e., overhidden by a subclass, casting does not affect which version of a method or property to be used.
- When an *overridden* method is invoked on an instance of a subclass, the subclass version of the method is always used, even if the subclass instance is cast to the superclass.

Casting versus Conversion

- ***Casting*** merely tells the compiler to treat an object as though it were an instance of the specified datatype.
- ***Conversion*** does change an object from one datatype to another.

Casting to String, Number, Boolean, Date & Array

- The String(), Number(), Boolean() and Array() functions perform ***data conversions***
- The Date() function returns the ***current time as a string***
- Syntax: ***Number(value)*** // to convert a value to a number

E.g: Var obj:Object=[1,2,3];

```
function output (msg:Object):Void {
        // use typeof to detect the datatype of primitive values
if ( typeof msg==“string”)          {          trace(msg);    }
        // use Instanceof to check the class of an object
if (msg instanceof Array){
        trace(msg.join(“\n”));          }}
```

Output panel display 1 2 3 (line by line)

Defining Classes

- A *class* is a **template** for the creation of ***objects***. Classes are the building blocks of an object-oriented program.
- *class* declaration:

class *ClassIdentifier* { }

where ClassIdentifier is the name of the class

E.x.: class Box {}

- A class definition must reside in an external plain text file that has the extension .as.
- Everything between the curly braces in a *class* declaration constitutes the *class definition block*, or more informally, the *class body*.

The class body can contain:

- **A constructor function** (used to initialize instances of the class)
- **Variable definitions** (the class's properties)
- **Function definitions** (the class's methods)
- **#include directives,** which can include files containing properties, methods, and constructor functions
- **Metadata tags** used in Flash MX 2004 components
- **Comments**
- Class definitions cannot be nested, and no other code can appear in a class definition.

Example

```
class Box {
  private static var inited:Boolean = false;
  public function Box ( ) {
    if (!inited) {      init( );    }  }
 private static function init( ):Void {
    if (100 == (10*10)) {
      trace("One hundred is ten times ten");    }
    inited = true;  }}
```

- The term ***instance*** *member* refers to either an *instance* ***property or an instance method***, while the term *class member* refers to either a ***class property or a class method***

Constructor Functions (Take 1)

- When we create an object, *we also want to initialize it.*
- Ex. Box Class: Initialize the new instance's size
  (i.e., set its width and height properties)
- Represent the new instance on screen
  (i.e., call the *draw( )* method)
- To initialize and perform setup tasks for new objects of a class, we create a ***constructor function***.
- The constructor function executes ***automatically each time an instance is created.***
- Typically, when we create a class, we immediately add an empty constructor function to it.

Constructor Functions (Take 1) (Contd…)

- class Box {

  public function Box ( ) {

  }}

- Now that our *Box* class has an empty constructor function, we'll give the class some properties and methods.
- When no constructor is provided for a class, ActionScript adds an empty one automatically at compile time.

Properties

- Classes use *properties* to store information. But some information relates to a class as a whole

Class properties

- Named data containers associated with a class

Instance properties

- Named data containers associated with objects (i.e., instances of the class)
- Together, class properties and instance properties are sometimes referred to as ***fixed properties***
- In ActionScript and ECMAScript 4, instance properties are sometimes called ***instance variables***, and class properties are sometimes called ***class variables***

Instance properties:

- Are *stored individually* on each instance of a class
- Are accessed through *instances only*
- Can be set *uniquely* for one instance without affecting any other instances
- An instance property is *declared once for the entire class*, but each instance of the class maintains its own value.
- The general syntax is:

var propertyName:datatype = value;

- If *datatype* is ***omitted***, no type checking is performed on the property
- var *propertyName* = *value*;

// No type checking for this property.

Access control modifiers in AS, Java, & C++

Access control modifier	ActionScript	Java	C++
public	No access restrictions	No access restrictions	No access restrictions
private	Class and subclass access only	Class access only	Class access only
no modifier	Same as public	Access allowed from class's own package only	Same as private
protected	Not supported	Access allowed from class's own package and class's subclasses in any other package	Access allowed from class and subclass only

Compile-Time Constant Expressions

- An instance property definition can assign a default value to a property, provided that the value is a so-called ***compile-time constant expression*.**
- A compile-time constant expression is an expression whose value can be *fully determined at compile time*. They include :
- ***null, numeric, boolean, and string constants***
- The following operators (used only on numbers, booleans, strings, null, or undefined): *+ (unary and binary), - (unary and binary), ~, !, *, /, %, <<, >>, >>>, <, >, <=, >=, instanceof, ==, !=, ===, !==, &, ^, |, &&, ^^, ||, and ?: (ternary operator)*
- ***Array literals, Object literals, and instances*** of the following classes: ***Array, Boolean, Number, Object, and String***
- References to other ***compile-time constant expressions***
- Example: *4 + 5, "Hello" + " world", null, [4, 5], new Array(4, 5)*

Methods

- Methods are functions that determine the ***behavior of a class***.
- If a parameter's *type* is ***omitted*** in the method definition, *no type checking is performed for that parameter* .
- If *returnType* is omitted, ***no type checking is performed for the return value.***
- A function with return type ***Void*** may ***not return a value***.

```
class ClassName {
function methodName (param1:type,
param2:type,...paramn:type):returnType {
            statements          }    }

Ex. function square (x:Number):Number {
  return x * x; }
```

Referring to the Current Object with the Keyword this

- Using this helps our class to ***remain encapsulated***, eliminating the need for the outside world to worry about the object's internal requirements.
- The *this* reference to the ***current object*** allows an object to refer to itself without the need for an additional parameter, and is therefore much cleaner.

Passing the current object to a method

- The this keyword is most often used when passing the current object to another object's method

```
function someMethod ( ):Void {
 // Pass this as a parameter to a completely separate object
someOtherObject.someOtherMethod(this);  }
```

Garbage collection

- When an object registers as a *listener of another object*, it should *always unregister itself before it is deleted*.
- This sloppiness can cause *serious waste of memory*. A class should always provide a means of ***cleaning up*** stray object references ***before an object is deleted***.
- ActionScript doesn't garbage-collect an object (i.e., free up the memory used by an object) until no more references to it remain. Cleanup is done in a custom ***die( ) or destroy( )*** method that must be invoked before an object is deleted.
- **B**y providing a ***die( ) method***, a class guarantees a safe means of *deleting its instances*

```
function die ( ):Void {
    disableReset( );
var b:Box = new Box( );
b.enableReset( );
b.die( );       delete b;
```

Nesting Functions in Methods

- ActionScript supports *nested functions*, which means that functions can be declared within methods or even within other functions.
- A local variable, a nested function is accessible only to its parent function (the function in which it is declared). Code outside the parent function cannot execute the nested function:

```
class Box {
  private var width:Number;
  private var height:Number;
  public function getArea ( ):Number {
     return multiply(width, height);
     function multiply
           (a:Number, b:Number):Number {
       return a * b;
     }   }  }
```

Accessing the current object from a function nested in a method

- A function nested in a method does not have direct access to the current object ***(the object on which the method was called).***
- The current object by storing a reference to it in a ***local variable.***

```
public function debugDimensions ( ):Void {
  var boxObj:Box = this;
  setInterval(displayDimensions, 1000);
  function displayDimensions ( ):Void {
    trace("Width: " + boxObj.width + ",
                Height: " + boxObj.height);
  }}
```

Getter and Setter Methods

- To bridge the gap between the convenience of property assignment and the safety of accessor methods, *ActionScript 2.0 supports "getter" and "setter" methods.*
- Getter and setter methods are ***accessor-like methods***, defined within a class body, that are *invoked automatically when a developer tries to get or set a property directly*.

```
function get propertyName ( ):returnType {
                    // getter method   statements    }
function set propertyName (newValue:type):Void {
                    // setter method  statements   }
```

- Getter and setter methods ***cannot be declared with the private attribute.***
- When invoked getter and setter methods does not require use of the function call operator, ( ).

A class with getter and setter methods

```
class Box {
  private var width_internal:Number;
  private var height_internal:Number;
  public function get width ( ):Number {
    return width_internal;  }
  public function set width (w:Number):Void {
    width_internal = w;  }
  public function get height ( ):Number {
    return height_internal;  }
  public function set height (h:Number):Void {
    height_internal = h;  } }
var b:Box = new Box( );
b.width  = 20;     // Calls the width setter.
trace(b.width);    // Displays: 20
b.height = 10;     // Calls the height setter.
trace(b.height);   // Displays: 10
```

Extra or Missing Arguments

- Extra arguments are not (indeed cannot be) type checked automatically by the compiler.

```
public function setWidth (w:Number):Void {
 // If w is undefined, then quit
if (w == undefined) {
        return;        }
width = w;      }
```

- The arguments object also stores:
  - A reference to the method currently executing (arguments.callee)
  - A reference to the method that called the method currently executing, if any (arguments.caller)

4.5 Constructor Functions (Take 2)

- A constructor function is the ***metaphoric womb*** of a class; it isn't responsible for creating new instances, but it can be ***used to initialize each new instance***.
- When we create a new instance of a class using the new operator, the class's constructor function runs. Within the constructor function, we can customize the *newly created instance by setting its properties or invoking its methods*.

```
class Box {
        public function Box ( ) {
        } }
```

- To define a constructor function, we use the *function statement within a class body.*
- The constructor function's name ***must match its class's name exactly*** (case sensitivity matters).

Constructor Functions (Take 2) (Contd…)

- The constructor function's definition ***must not specify a return type*** (not even *Void*).
- The constructor function ***must not return a value***
- The constructor function's definition ***must not include the static attribute****, but it can be public or private.*
- ActionScript automatically provides a ***default constructor*** that takes ***no parameters and performs no initialization*** on new instances of the class.
- A constructor function's parameter values are passed to it via the **new operator at object-creation time**

.

new SomeClass(value1, value2,...valuen);

new Box(2, 3);

Simulating Multiple Constructor Functions

Unlike Java, ActionScript does not support multiple constructor functions for a single class (*overloaded constructors* in Java).

```
class Box {
  public var width:Number;
  public var height:Number;
    if (arguments.length == 0) {
      boxNoArgs( );     }
    else if (typeof a1 == "string") {
      boxString(a1);     }
    else if (typeof a1 == "number" && typeof a2 == "number") {
      boxNumberNumber(a1, a2);     }
    else {
      trace("Unexpected number of arguments to constructor.");
    }  }
  private function boxNoArgs ( ):Void {
    if (arguments.caller != Box) {
      return;     }
    width  = 1;
    height = 1;  }
```

overloaded constructors

```
private function boxString (size):Void {
  if (arguments.caller != Box) {
    return;     }
  if (size == "large") {
    width  = 100;        height = 100;
  } else if (size == "small") {
    width  = 10;         height = 10;
  } else {trace("Invalid box size specified"); }}
private function boxNumberNumber (w, h):Void {
  if (arguments.caller != Box) {
    return;     }
  width  = w;  height = h;  }}

// Usage:
var b1:Box = new Box( );
trace(b1.width);   // Displays: 1
var b2:Box = new Box("large");
trace(b2.width);   // Displays: 100
var b3:Box = new Box(25, 35);
trace(b3.width);   // Displays: 25
```

Completing the Box Class

- First decide whether the display elements will render themselves or be rendered by a central class.
- Decide how often the displayed elements should be updated—either when some event occurs (such as a mouseclick) or repeatedly (as fast as frames are displayed in the Flash Player).
- Decide on the rendering technique. Each visual element in a Flash movie must be displayed in a movie clip.
- Decide on a screen refresh strategy.

```
class Box {
  private var width:Number;
  private var height:Number;
  private var container_mc:MovieClip;
  public function Box (w:Number, h:Number, x:Number, y:Number,
                       target:MovieClip, depth:Number) {
    container_mc = target.createEmptyMovieClip
                       ("boxcontainer" + depth, depth);
    setWidth(w);     setHeight(h);
    setX(x);     setY(y);   }
```

A Box class complete with drawing routines

```
public function getWidth ( ):Number {
   return width;   }
public function setWidth (w:Number):Void {
   width = w;      draw( );   }
public function getHeight ( ):Number {
   return height;   }
public function setHeight (h:Number):Void {
   height = h;      draw( );   }

public function getX ( ):Number {
   return container_mc._x;   }

public function setX (x:Number):Void {
   container_mc._x = x;   }

public function getY ( ):Number {
   return container_mc._y;   }

public function setY (y:Number):Void {
   container_mc._y = y;   }
```

```
public function draw ( ):Void {
 container_mc.clear( );
 container_mc.lineStyle(1, 0x000000);
 container_mc.moveTo(0, 0);
 container_mc.beginFill(0xFFFFFF, 100);
 container_mc.lineTo(width, 0);
 container_mc.lineTo(width, height);
 container_mc.lineTo(0, height);
 container_mc.lineTo(0, 0);
 container_mc.endFill( );
 } }
// Usage: Code in flash document timeline
var b:Box = new Box(250, 260, 100, 110, this, 1);
b.setX(400);        b.setY(400);
b.setWidth(10);     b.setHeight(20);
trace(b.getX( ));          // Displays: 400
trace(b.getY( ));          // Displays: 400
trace(b.getWidth( ));      // Displays: 10
trace(b.getHeight( ));     // Displays: 20
```

A Box class complete with drawing routines (Contd..)

```
public function draw ( ):Void {
 container_mc.clear( );
 container_mc.lineStyle(1, 0x000000);
 container_mc.moveTo(0, 0);
 container_mc.beginFill(0xFFFFFF, 100);
 container_mc.lineTo(width, 0);
 container_mc.lineTo(width, height);
 container_mc.lineTo(0, height);
 container_mc.lineTo(0, 0);
 container_mc.endFill( );
 } }
// Usage: Code in flash document timeline
var b:Box = new Box(250, 260, 100, 110, this, 1);
b.setX(400);        b.setY(400);
b.setWidth(10);     b.setHeight(20);
trace(b.getX( ));           // Displays: 400
trace(b.getY( ));           // Displays: 400
trace(b.getWidth( ));       // Displays: 10
trace(b.getHeight( ));      // Displays: 20
```

Authoring An ActionScript 2.0 Class

Class Authoring Quick Start

- To create an *ActionScript 2.0 class*, follow these general steps:
- Create a new ***text file*** with the **.as** extension using any plain text editor or *Flash MX Professional 2004's built-in editor.*
- The .as file's name must ***match the class name exactly*** (case sensitivity matters).
- Add the class definition to the .as file.

```
class NameOfClass {
                    // Class body goes here
                    }
```

To use an ActionScript 2.0 class in a Flash movie, Steps:

1. Create a .fla file with any name, and ***place it in the same folder as the .as file*** from Step 1 in the preceding procedure.
2. Optionally specify the ***class's export frame*** for the .fla file via File Publish Settings Flash ActionScript Version Settings Export Frame for Classes. This determines when the class loads and when it becomes available in the movie. *The export frame is usually set to some frame after your movie's preloader.*
3. Use the *class as desired throughout the .fla file*, but after the export frame specified in Step 2 (if any).
4. Export a .swf file using one of the following: ***File Publish, Control Test Movie, or File Export Export Movie.***

Designing the ImageViewer Class

- With our *ImageViewer* class's name we have a list of functionality that could be required of the *ImageViewer* class:
  - *Load an image*
  - *Display an image*
  - *Crop an image to a particular rectangular "view region"*
  - *Display a border around the image*
  - *Display image load progress*
  - *Reposition the view region*
  - *Resize the view region*
  - *Pan (reposition) the image within the view region*
  - *Zoom (resize) the image within the view region*

ImageViewer Implementation (Take 1)

- To create the ImageViewer.as file using Flash MX Professional 2004, follow these steps:
- *Choose File New.*
- *In the New Document dialog box, on the General tab, for the document Type, choose ActionScript File.*
- *Click OK. The script editor launches with an empty file.*
- *Choose File Save As.*
- *In the Save As dialog box, specify **ImageViewer.as** as the filename (using upper- and lowercase as shown) and save the file in the imageviewer folder you created.*
- Notice that the class name, *ImageViewer*, and the filename, ImageViewer.as, ***must match exactly*** (apart from the .as file extension).

The ImageViewer class with constructor and loadImage( ) method, properties (Take 1)

```
class ImageViewer {
 private var container_mc:MovieClip;

public function ImageViewer (target:MovieClip, depth:Number)
{
    container_mc = target.createEmptyMovieClip
                   ("container_mc" + depth, depth);
}
public function loadImage (URL:String):Void
{
    container_mc.loadMovie(URL);
}}
```

Using ImageViewer in a Movie

- *Find a (nonprogressive format) JPEG image on your system.*
- *Name the JPEG image picture.jpg and place it in the same **folder as the ImageViewer.as** file you created earlier.*
- *Next we'll create a Flash document (.fla file) from which we'll publish a Flash movie (.swf file) containing an instance of our ImageViewer class.*
- *For now, we'll place the **.fla file, .as file, .swf file, and .jpg file all in the same folder,** making it easy for each file to access the other files.*
- Create the Flash document that uses the *ImageViewer* class. Follow these steps:
  - *In the Flash authoring tool, choose File -> New.*
  - *In the New Document dialog box, on the General tab, for the document Type, choose Flash Document, then click OK.*

Using ImageViewer in a Movie (Contd…)

- Use File -> Save As to save the Flash document as **imageViewer.fla** in the same folder as the ImageViewer.as file.
- In imageViewer.fla's main timeline, rename *Layer 1* to **scripts**
- Follow these steps to instantiate *ImageViewer* on frame 1 of imageViewer.fla's main timeline:
  - Use Window -> Development Panels ->Actions (F9) to open the Actions panel.
  - Select frame 1 in imageViewer.fla's main timeline.
  - Into the Actions panel, enter the following code:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

Using ImageViewer in a Movie (Contd…)

- Let's export our imageViewer.swf file and test it in Flash's Test Movie mode, as follows:
  - *Choose Control -> Test Movie. The .swf file should play, and your image should load and appear.*
  - *When you're finished marveling at your work, choose File -> Close to return to the imageViewer.fla file.*
- To change imageViewer.fla's export settings to support playback in Flash Player 6, follow these steps:
  - *Choose File -> Publish Settings.*
  - *On the Flash tab of the Publish Settings dialog box, select Flash Player 6 as the Version option.*
  - *Click OK.*

Preloading the ImageViewer Class

- Let's change our imageViewer.fla file so that the classes it uses aren't loaded until frame 10:
- *Choose File -> Publish Settings.*
- *In the Publish Settings dialog box, on the Flash tab, next to the ActionScript Version, click Settings.*
- *In the ActionScript Settings dialog box, for Export Frame for Classes, enter **10**.*
- *Click OK to confirm the ActionScript Settings.*
- *Click OK to confirm the Publish Settings.*
- Now let's add a very basic preloader to our imageViewer.fla file so load progress is reported while the *ImageViewer* class loads. When loading is complete, we'll advance the playhead to frame 15 where we'll instantiate *ImageViewer* (as we previously did on frame 1).

Preloading the ImageViewer Class (Contd..)

- First, we'll make the timeline 15 frames long, as follows:
  - *In the main timeline of imageViewer.fla, select frame 15 of the scripts layer.*
  - *Choose Insert -> Timeline ->Keyframe (F6).*
- Next, we'll add a *labels* layer with two frame labels, loading and main. The labels designate the application's loading state and startup point, respectively.
- *Choose Insert ->Timeline ->Layer.*
- *Double-click the new layer's name and change it to **labels**.*
- *At frames 4 and 15 of the labels layer, add a new keyframe (using Insert ->Timeline ->Keyframe).*
- *With frame 4 of the labels layer selected, in the Properties panel, under Frame, change <Frame Label> to **loading**.*
- *With frame 15 of the labels layer selected, in the Properties panel, under Frame, change <Frame Label> to **main**.*

Now add the preloader script to the *scripts* layer:

- At frame 5 of the *scripts* layer, add a new keyframe (using Insert ->timeline ->Keyframe).
- With frame 5 of the *scripts* layer selected, enter the following code into the Actions panel:

```
if (_framesloaded == _totalframes) {
gotoAndStop("main");}
else {  gotoAndPlay("loading");   }
```

- Next, we'll move the code that creates our *ImageViewer* instance from frame 1 to frame 15 of the *scripts* layer:
- Select frame 1 of the *scripts* layer.
- In the Actions panel, cut (delete using Ctrl-X or Cmd-X) the following code from frame:

```
var viewer:ImageViewer = new ImageViewer(this, 1);
viewer.loadImage("picture.jpg");
```

- With frame 15 of the *scripts* layer selected, paste (using Ctrl-V) the code you deleted in Step 2 into the Actions panel.
- Finally, we'll add a loading message that displays while the *ImageViewer* class loads:
- With frame 1 of the *scripts* layer selected, enter the following code into the Actions panel:

```
this.createTextField("loadmsg_txt", 0, 200, 200, 0, 0);
loadmsg_txt.autoSize = true;
loadmsg_txt.text = "Loading...Please wait.";
```

- With frame 15 of the *scripts* layer selected, enter the following code at the end of the Actions panel

```
loadmsg_txt.removeTextField( );
```

ImageViewer Implementation (Take 2)

- The third and fourth requirements: ***cropping the image and giving it a border***. We must apply a ***mask*** to it, which hides unwanted areas of the image from view.
- We'll add parameters to the *ImageViewer* constructor to specify the *size of the mask and the position of the cropped image.*
- To create border, we'll *create a movie clip, then draw a square outline in it.*

Splitting out the work into separate methods has the following benefits:

- It makes the code more intelligible.
- It simplifies the testing process (testing each method individually is easier than testing a large block of code that performs many operations).
- It allows assets to be created and re-created independently (e.g., the border on an image can change without reloading the image).
- It allows asset-creation operations to be modified or overridden independently (e.g., a new border-creation routine can be coded without disturbing other code).

Table The ImageViewer class's new instance methods that create movie clips

Method name	*Method description*
buildViewer( )	*Invokes individual methods to create the container, image, mask, and border clips*
createMainContainer( )	*Creates the* `container_mc` *clip*
createImageClip( )	*Creates the* `image_mc` *clip*
createImageClipMask( )	*Creates the* `mask_mc` *clip*
createBorder( )	*Creates the* `border_mc` *clip*

- **A reference to the target clip to which container_mc should be attached**
- **A list of depths indicating the visual stacking order of the container, image, mask, and border clips**
- **The style (line weight and color) of the border around the image**

Table The ImageViewer class's instance and class properties

Property name	Type	Property description
`container_mc`	*Instance*	*A reference to the main container clip, which contains all movie clips used by each ImageViewer instance*
`target_mc`	*Instance*	*A reference to the clip that will contain the* `container_mc` *clip, as specified by the ImageViewer constructor*
`containerDepth`	*Instance*	*The depth on which* `container_mc` *is created in* `target_mc`*, as specified by the ImageViewer constructor*
`imageDepth`	*Class*	*The depth on which* `image_mc` *is created in* `container_mc`
`maskDepth`	*Class*	*The depth on which* `mask_mc` *is created in* `container_mc`
`borderDepth`	*Class*	*The depth on which* `border_mc` *is created in* `container_mc`
`borderThickness`	*Instance*	*The thickness, in pixels, of the border around the* `image_mc` *clip*
`borderColor`	*Instance*	*The integer RGB color of the border around the* `image_mc` *clip*

The ImageViewer class, take 2

```
class ImageViewer { // Movie clip references
  private var container_mc:MovieClip;
  Private var target_mc:MovieClip;
  // Movie clip depths
  private var containerDepth:Number;
  private static var imageDepth:Number  = 0;
  private static var maskDepth:Number   = 1;
  private static var borderDepth:Number = 2;
  private var borderThickness:Number;
  private var borderColor:Number;
  public function ImageViewer (target:MovieClip,
          depth:Number,x:Number,y:Number,
          w:Number,h:Number,borderThickness:Number,
          borderColor:Number) {// Assign property values.
    target_mc = target;
    containerDepth = depth;
    this.borderThickness = borderThickness;
    this.borderColor = borderColor;
    // Set up the visual assets for this ImageViewer.
    buildViewer(x, y, w, h);  }
```

The ImageViewer class, take 2

```
private function buildViewer (x:Number,
        y:Number,w:Number,h:Number):Void {
        createMainContainer(x, y); createImageClip( );
        createImageClipMask(w, h); createBorder(w, h);
}
private function createMainContainer (x:Number,
y:Number):Void { container_mc =
        target_mc.createEmptyMovieClip("container_mc" ,
containerDepth, containerDepth);
container_mc._x = x; container_mc._y = y;
}
private function createImageClip ( ):Void {
         container_mc.createEmptyMovieClip("image_mc",
         imageDepth);   }
private function createImageClipMask
          (w:Number,h:Number):Void {
      if (!(w > 0 && h > 0)) {
        return;
      }
```

The ImageViewer class, take 2

```
container_mc.createEmptyMovieClip("mask_mc",
maskDepth);
    container_mc.mask_mc.moveTo(0, 0);
    container_mc.mask_mc.beginFill(0x0000FF);
    container_mc.mask_mc.lineTo(w, 0);
    container_mc.mask_mc.lineTo(w, h);
    container_mc.mask_mc.lineTo(0, h);
    container_mc.mask_mc.lineTo(0, 0);
    container_mc.mask_mc.endFill( );
    container_mc.mask_mc._visible = false;
    }
  private function createBorder (w:Number,
                  h:Number):Void {
    if (!(w > 0 && h > 0)) {
      return;    }
```

The ImageViewer class, take 2

```
container_mc.createEmptyMovieClip("border_mc",
                                   borderDepth);
container_mc.border_mc.lineStyle(borderThickness,
                                   borderColor);
      container_mc.border_mc.moveTo(0, 0);
      container_mc.border_mc.lineTo(w, 0);
    container_mc.border_mc.lineTo(w, h);
    container_mc.border_mc.lineTo(0, h);
    container_mc.border_mc.lineTo(0, 0);
  }
  public function loadImage (URL:String):Void {
    container_mc.image_mc.loadMovie(URL);
    container_mc.onEnterFrame = function ( ):Void {
      this.image_mc.setMask(this.mask_mc);
      delete this.onEnterFrame;
    }  }
}
```

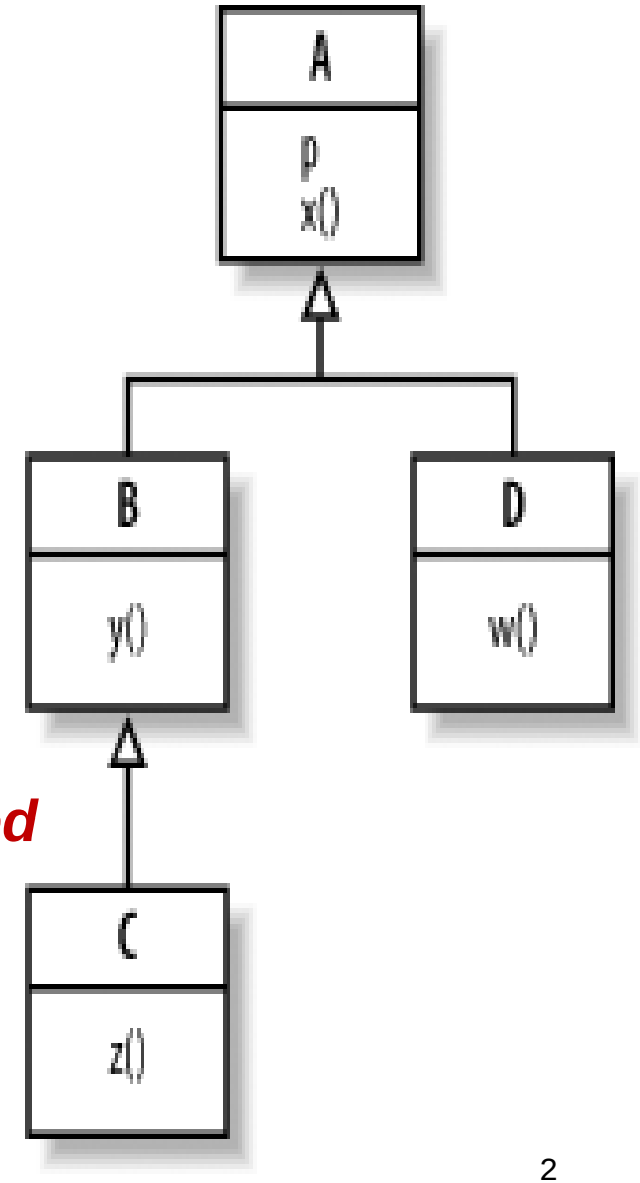
- Now that our *ImageViewer* class can crop an image and add a border to it.
- When we used the first version of the *ImageViewer* class, we placed the following code on frame 15 of imageViewer.fla:

Unit IV

Inheritance
Authoring an ActionScript 2.0 Subclass
Interfaces
Packages
Exceptions

INHERITANCE

- *A Primer on Inheritance*
- *Subclasses as Subtypes*
- *An OOP Chat Example*
- *Overriding Methods and Properties*
- *Constructor Functions in Subclasses*
- *Subclassing Built-in Classes*
- *Augmenting Built-in Classes & Objects*
- *The Theory of Inheritance*
- *Abstract and Final Classes Not Supported*



A Primer on Inheritance

- In OOP, inheritance is a formal ***relationship between two or more classes***, wherein *one class borrows (or inherits) the property and method definitions of another class*

```
class A {    public var p:Number = 10;
             public function x ( ):Void {
             trace("Method x( ) was called.");  }}
```

- We can create an instance of class *A*, invoke method *x( )*, and access property p like this:

```
var aInstance:A = new A( );
aInstance.x( );       // Displays: Method x( ) called.
trace(aInstance.p);  // Displays: 10
```

- Now let's add a second class, *B*, that inherits method *x( )* and property p from class *A*.

A Primer on Inheritance (Contd…)

- To set up the inheritance *relationship between A and B*, we use the ***extends*** keyword to indicate that class *B* inherits class *A*'s method and property definitions:

```
class B extends A {                    }
```

- Because class *B* extends (inherits from) class *A*, *instances of B can automatically use the method x( )* and the property p

```
var bInstance:B = new B( );
bInstance.x( );          // Displays: Method x( ) was called.
trace(bInstance.p);  // Displays: 10
```

- class *B* would define its *own methods and/or properties* in addition to inheriting *A*'s methods and properties.

```
class B extends A {
public function y ( ):Void {trace("Method y( ) was called.");  } }
```

A Primer on Inheritance (Contd…)

- Now instances of *B* can use all the methods and properties of both *B* and its superclass, *A*:

```
var bInstance:B = new B( );
// Invoke inherited method, defined by class A.
bInstance.x( );       // Displays: Method x( ) was called.
// Invoke method defined by class B.
bInstance.y( );       // Displays: Method y( ) was called.
// Access inherited property.
trace(bInstance.p);  // Displays: 10
```

- Inheritance relationship between two classes, the extended class (in our case, class *A*) is called the ***base class***, and the class that does the extending (in our case, class *B*) is called the ***derived class.***

A Primer on Inheritance (Contd…)

- However, the terms *"base class" and "derived class"* have several synonyms, including superclass and subclass, parent and child, and type and subtype.
- The following code shows *third class,C, that extends class B and also defines a new method, z( ).*
- Class *C* can use all the *methods and properties defined by itself, its* ***superclass (B),*** *or its superclass's* ***superclass (A)***
- Every class in ActionScript (***both built-in and custom***) inherits directly or indirectly from the root of the built-in hierarchy: *Object*.
- The *Object* class defines some very basic methods that ***all classes can use.***

A Primer on Inheritance (Contd…)

```
class C extends B {
        public function z ( ):Void {
        trace("Method z( ) was called.");  } }
var cInstance:C = new C( );
cInstance.x( );  // Displays: Method x( ) was called.
cInstance.y( );  // Displays: Method y( ) was called.
cInstance.z( );  // Displays: Method z( ) was called.
trace(cInstance.p);  // Displays: 10
```

- class *D* inherits directly from class *A*. Class *D* uses the methods and properties defined by itself and by its superclass, *A*.

```
class D extends A {
        public function w ( ):Void {
            trace("Method w( ) was called.");  }}
```

Subclasses as Subtypes

- A subclass is considered a ***subtype*** of its superclass. An instance of the subtype *can be used anywhere its superclass's type is expected.* ***var aInstance:A;***
- we can legally assign that variable an instance of any subclass of class *A*: ***aInstance = new B( );***
- The preceding assignment works because the compiler knows that *any instance of class B has (through inheritance) all the methods and properties defined by class A.*
- However, ***the reverse is not true***. We cannot assign an instance of *A* to a variable whose datatype is *B*:

  var bInstance:B = new A( ); // Type Mismatch Error.
- That assignment does not work because the *compiler cannot guarantee that an instance of class A will have the methods and properties defined by class B.*

An OOP Chat Example

- *Consider a chat application that defines a class, ChatRoom, which handles communication among a group of users.*
- *ChatRoom class defines methods for displaying & sending chat messages and for managing a list of users in the room*

Table 6-1. ChatRoom class's property and methods

Method or property name	Purpose
`userList`	Property storing a reference to the on-screen *List* component that displays users in the room.
displayMessage( )	Displays an incoming user message in a text field.
sendMessage( )	Sends an outgoing message to other users.
onAddUser( )	Adds the new user to the `userList`. Invoked when a user joins the room.
onRemoveUser( )	Removes the departed user from the `userList`. Invoked when a user leaves the room.
onUserChangeName( )	Changes the user's name in the `userList`. Invoked when a user's name changes.

AvatarChatRoom class's methods & properties

Method or property name	Inheritance relationship	Original behavior	Behavior added by *AvatarChatRoom* class
`userList`	Inherited property	Stores a reference to the on-screen *List* component that displays users in the room	None
`avatars`	Property added by subclass	Not applicable	Stores a list of *Avatar* instances displayed on screen
displayMessage( )	Overridden method	Displays an incoming user message in a text field	Also displays a message bubble next to the user's avatar
sendMessage( )	Inherited method	Sends an outgoing message to other users	None
onAddUser( )	Overridden method	Adds the new user to the `userList`	Also displays a new avatar on the screen
onRemoveUser( )	Overridden method	Removes the departed user from the `userList`	Also removes the user's avatar on the screen
onUserChangeName( )	Overridden method	Changes the user's name in the `userList`	Also changes the name displayed under an avatar
onUserChangePosition ( )	Method added by subclass	Not applicable	Positions the user's avatar on screen

Example An inheritance example using ChatRoom and AvatarChatRoom

```
class ChatRoom {
private var userList:mx.controls.List;
public function displayMessage(userID:String, msg:String):Void {
        trace("Displaying chat message in chat text field.");
}
public function sendMessage(msg:String):Void {
        trace("Sending chat message.");
}
public function onAddUser(userID:String):Void {
        trace("Adding user to userList.");
}
public function onRemoveUser(userID:String):Void {
        trace("Removing user from userList.");
}
public function onUserChangeName(userID:String, newName:String):Void {
        trace("Changing name in userList.");
        }
}
```

Example An inheritance example using ChatRoom and AvatarChatRoom

```
class AvatarChatRoom extends ChatRoom {
  private var avatars:Array;
  public function displayMessage (userID:String,
msg:String):Void {
    super.displayMessage(userID, msg);
    trace("Displaying message in avatar text bubble.");
  }
  public function onAddUser (userID:String):Void {
    super.onAddUser(userID);
    trace("Creating avatar for new user.");
  }
  public function onRemoveUser (userID:String):Void {
    super.onRemoveUser(userID)
    trace("Removing avatar for user.");
  }
  public function onUserChangeName (userID:String,
newName:String):Void {
    super.onUserChangeName(userID, newName);
    trace("Changing name on avatar.");
  }
  public function onUserChangePosition (userID:String,
                                        newX:Number,
newY:Number):Void {
    trace("Repositioning avatar.");
  }}
```

Constructor Function in Subclasses

- A constructor function initializes the instances of a class by:
  - *Calling methods that perform setup tasks*
  - *Setting properties on the object being created*
- When a class is extended, the subclass can define a constructor function of its own. A subclass constructor is expected to:
  - *Perform setup tasks related to the subclass*
  - *Set properties defined by the subclass*
  - *Invoke the superclass constructor (superconstructor)*
- In all inheritance relationships, setup and property initialization *relating to the superclass occur in the superclass constructor, not in the subclass constructor.*

Constructor Function in Subclasses (Contd..)

- A subclass constructor function, if specified, is formally required to invoke its *superclass constructor*, via the keyword ***super***, as the first statement in the function.

```
class A {  public function A ( ) {  }       }
class B extends A {   public function B ( ) {
    // Invoke superclass's constructor function explicitly
    super( );  }}

public function B ( ) {
  // Invoke superclass's constructor function explicitly.
  super( );}
public function B ( ) {
 // No constructor call. The compiler provides one implicitly. }
```

The Theory of Inheritance

- Inheritance lets us ***separate a core feature set*** from customized versions of that feature set. Code for the *core is stored in a superclass while code for the customizations is kept neatly in a subclass.*
- More than one subclass can extend the superclass, *allowing multiple customized versions of a particular feature set to exist simultaneously.*
- Inheritance also lets us express the *architecture* of an application in *hierarchical terms* that mirror the real world and the human psyche.
- In addition *to code reuse and logical* hierarchy, inheritance allows instances of different *subtypes to be used where a single type is expected*. Known as ***polymorphism***

Polymorphism and Dynamic Binding

- The word ***polymorphism*** itself means literally ***"many forms***"—*each single object can be treated as an instance of its own class or as an instance of any of its superclasses*.
- Polymorphism's partner is ***dynamic binding***, which guarantees that a *method invoked on an object will trigger the behavior defined by that object's actual class.*
- Dynamic binding, which occurs at ***runtime***, with *static type checking, which occurs at compile time*
- *Dynamic binding* is the runtime process by which each invocation of *draw( )* is associated with the appropriate implementation of that method.
- Dynamic binding is often called ***late binding:*** the method call is *bound* to a particular implementation "late" (i.e., at runtime).

Polymorphism and Dynamic Binding(Contd..)

```
class Shape {   public function draw ( ):Void {
    // No implementation.   }      }
class Circle extends Shape {
  public function draw ( ):Void {
  }}

class Rectangle extends Shape {
  public function draw ( ):Void {
  }}
class Triangle extends Shape {
  public function draw ( ):Void {
  }}
```

Abstract and Final Classes Not Supported

- An abstract class is any class that defines one or more *abstract* ***methods—methods that have a signature and a return type but no implementation***
- A class that wishes to extend an abstract class must implement ***all of the superclass's abstract methods***; otherwise, a compile-time error occurs.
- ActionScript 2.0 ***does not yet support abstract classes*** or abstract methods. Instead we simply define a method with no code in its body
- A ***final method*** is a method that *must not be implemented by a subclass.* Final methods are used to *prevent programmers from creating subclasses that accidentally introduce problems into the behavior of a superclass*

Authoring an ActionScript 2.0 Subclass Extending ImageViewer's Capabilities

- The list of possible functional requirements for our *ImageViewer* class:
  - Load an image , Display an image
  - Crop an image to a particular rectangular "view region"
  - Display a border around the image
  - Display image load progress
  - *Reposition the view region, Resize the view region*
  - Pan (reposition) the image within the view region
  - Zoom (resize) the image within the view region
- The *ImageViewer* class already implements the first five requirements, but it stops there.
- In the *ImageViewerDeluxe* class, we'll add the next two items on the list: *the abilities to reposition and resize the viewer.*

The ImageViewerDeluxe Skeleton

- The bare bones of any subclass requires:
  - *A class declaration that defines the inheritance relationship*
  - *A constructor that invokes the superclass constructor with the expected arguments*

```
class ImageViewerDeluxe extends ImageViewer {
  public function ImageViewerDeluxe (target:MovieClip,
    depth:Number,  x:Number, y:Number, w:Number, h:Number,
    borderThickness:Number, borderColor:Number)  {
    super(target, depth, x, y, w, h, borderThickness,
    borderColor);
} }
```

To create the ImageViewerDeluxe.as class file using Flash MX Professional 2004, follow these steps:

1. Choose File → New.
2. In the New Document dialog box, on the General tab, for Type, choose ActionScript File.
3. Click OK. The script editor launches with an empty file.
4. Copy the code into the script editor: in the next slide.
5. Choose File → Save As.
6. In the Save As dialog box, specify **ImageViewerDeluxe.as** as the filename (which is case sensitive), and save the file in the imageviewer folder that you created.
7. Instances of the *ImageViewerDeluxe* class have all the *ImageViewer* class's methods and can be used anywhere *ImageViewer* instances are used.

Adding setPosition( ) and setSize( ) Methods

- ***setPosition( )***
  - Changes the location of the viewer by assigning the (x, y) coordinates of the viewer's main container movie clip
- ***setSize( )***
  - Changes the size of the viewer by re-creating the mask and border over the image movie clip (i.e., by recropping the image)

```
public function setPosition (x:Number, y:Number):Void {
    container_mc._x = x; container_mc._y = y;          }
```

- The *setSize( )* method, which resizes the image viewer by recropping the image:

```
public function setSize (w:Number, h:Number):Void {
    createImageClipMask(w, h); createBorder(w, h);
    container_mc.image_mc.setMask(container_mc.mask_mc);
    }
```

The ImageViewerDeluxe class, final version

```
class ImageViewerDeluxe extends ImageViewer {
  public function ImageViewerDeluxe (target:MovieClip,
    depth:Number,  x:Number, y:Number, w:Number, h:Number,
    borderThickness:Number, borderColor:Number)  {
    super(target, depth, x, y, w, h, borderThickness,
    borderColor);  }    }
public function setPosition (x:Number, y:Number):Void {
        container_mc._x = x;
        container_mc._y = y;                                    }
public function setSize (w:Number, h:Number):Void     {
        createImageClipMask(w, h); createBorder(w, h);
    container_mc.image_mc.setMask(container_mc.mask_mc);
                                                                }
```

ImageViewerDeluxe class, final version (Contd..)

```
public function getImageWidth ( ):Number {
        return container_mc.image_mc._width;   }
public function getImageHeight ( ):Number {
        return container_mc.image_mc._height;  }
public function setShowFullImage(show:Boolean):Void {
        showFullImage = show;                  }
public function getShowFullImage( ):Boolean {
       return showFullImage;       }
public function scaleViewerToImage ( ):Void {
        setSize(getImageWidth( ), getImageHeight( ));    }
public function onLoadInit (target:MovieClip):Void {
       super.onLoadInit(target);
       if (showFullImage) {   scaleViewerToImage( ); } } }
```

INTERFACES

The Case for Interfaces

- An *interface* is an ActionScript 2.0 language construct used to ***define a new datatype***, much as a class defines a datatype.
- A class both defines a ***datatype and provides the implementation*** for it, an interface *defines a datatype in* ***abstract terms*** only; an interface provides *no implementation for the datatype.*
- An interface, instead of providing its own implementation, is ***adopted by one or more classes*** that agree to provide the implementation.
- A class that provides an ***implementation for an interface*** belongs both to its own datatype and to the datatype defined by the interface.

The Case for Interfaces (Contd..)

```
class OrderListener {
  public function onOrderError ( ):Void {
    // Generic implementation of onOrderError( ), not shown…  }}
class StatsTracker extends OrderListener {
  // Override OrderListener.onOrderError( ).
  public function onOrderError ( ) {
    // Send problem report to database. Code not shown…  }}
class OrderProcessor {
  public function addListener (ol:OrderListener):Boolean {
    // Code here should register ol to receive OrderProcessor
    events,
    // and return a Boolean value indicating whether registration  }}
class OrderUI extends MovieClip {
  public function onOrderError ( ) {
    // Display problem report on screen, not shown…  }}
```

Interfaces and Multidatatype Classes

- An interface is simply a list of methods.

```
interface OrderListener {
  function onOrderError( );
}
```

- Classes use the *implements* keyword to enter into an agreement with an interface, promising to define the methods it contains.

```
class OrderUI extends MovieClip implements OrderListener {
    public function onOrderError ( ) {
      // Display problem report on screen, not shown...
    }}
```

Interface Syntax and Use

- To create an interface in ActionScript 2.0, we use the *interface* keyword, using the following syntax:

```
interface  SomeName  {
  function method1  (param1 :datatype ,...paramn :datatype ):returnType ;
  function method2  (param1 :datatype ,...paramn :datatype ):returnType ;
     ...
  function methodn  (param1 :datatype ,...paramn :datatype ):returnType ;
}
```

- In interfaces, method declarations do not (and must not) include curly braces.
- All methods declared in an interface must be *public*; hence, by convention, the attribute *public* (which is the default) is omitted from method definitions in an interface.

Interface Syntax and Use (Contd..)

- ***Class files***
  - Contain the method and property definitions for each class. They start with the *class* keyword.
- ***Intrinsic files***
  - Contain method signatures strictly for the purpose of satisfying the compiler's type checking. They start with the *intrinsic* keyword.
- ***Interface files***
  - Contain a list of methods to be implemented but not the implementations themselves. They start with the *interface* keyword.
- A class that wishes to adopt an interface's datatype *must agree to implement that interface's methods.*

Interface Syntax and Use (Contd..)

- The class uses the *implements* keyword, which has the following syntax:

  class SomeName implements SomeInterface { }

- The implementing class's method definitions must match the interface's method definitions exactly, including number of parameters, parameter types, and return type.
- If differs between the interface and the implementing class, the compiler generates an error.
- A class can legally implement more than one interface by separating interface names with commas, as follows:

```
class SomeName implements SomeInterface,
                          SomeOtherInterface {
}
```

P A C K A G E S

- Larger applications, creates dozens of different ***classes and packages grouping them logically together.***
- ***To group classes and interfaces together, we use packages.***
- *namespace collision* or *naming conflict*.
- A package is a unique place to ***put a group of classes, much as a directory on your hard drive is a unique place*** to put a group of files.
- Packages allow multiple classes of the same name to ***coexist without namespace collisions***, because each package constitutes a *namespace* within which the class name is unique.
- A namespace is a set of names that contains *no duplicates*.

Package Syntax

- The ***Domain Name System (DNS)*** defines a namespace in which registrants can *reserve a domain nam*e, such as moock.org
- Every package has a name that *describes the classes it contains.*
- By convention, package names ***do not use initial caps*** but might use mixed case, such as greatUtils.
- To refer to a specific class within a package, we use the dot operator:

  packagename.ClassName

- game.Player Packages can be nested to form a hierarchy; that is, a package can contain another package.

  game.vehicles.SpaceShip

- By nesting packages, we can organize our classes into discrete subcategories within a larger category, which again prevents naming conflicts.

Import statement

- An ***unqualified reference*** includes just the class name.

  var ship:SpaceShip = new SpaceShip( );

- If a class is in a package, we must use a fully qualified reference when creating instances of the class and specifying the ***datatypes of variables, parameters, and return types.***

  var ship:game.vehicles.SpaceShip = new game.vehicles.SpaceShip( );// Fully qualified

- The ***import*** statement lets us use an *unqualified class* reference as an ***alias*** for a fully qualified class reference. The basic syntax of *import* is:

  import packagename.ClassName;

  import game.vehicles.SpaceShip;

Importing an Entire Package

- To import all the classes in a package, use the wildcard character, *:

  import packagename.*;

- This wildcard syntax lets us refer to any class in *packagename* by its unqualified name rather than its fully qualified name.

  import geometry.Circle;
  import geometry.Triangle;
  import geometry.Rectangle;

  import geometry.*;

- If we use package *wildcards to import two classes with the same name but from different packages*, we again encounter a ***namespace collision***.

Package Naming Conventions

```
import game.vehicles.*;
import ordering.*;
var s:Ship = new Ship( );  // new game.vehicles.Ship
var s:ordering.Ship = new ordering.Ship( );
```

- To avoid confusion, we should use ***fully qualified names*** for all references to *Ship*.
- *Just as two class names can conflict, two package names can conflict.*
- To avoid package or class naming conflicts you should use the convention of ***placing all your classes and packages in a package named after your organization's domain name.***
- com.acme com precedes acme (the package name is the reverse of the domain name).

Defining Packages

- Defining a package for a class requires *two* general steps:
  - *Create a directory structure whose name identically matches the package hierarchy.*
  - *Specify the package name when defining the class.*
- There is no such thing as a package file. A package is a concept based on placing a given class's .as file in the appropriate folder.
- To place the class *Player* in package com.yourdomain.game

  Create a new directory named com.

  Create a new directory inside com named yourdomain.

  Create a new directory inside yourdomain named game.

  Create a text file, Player.as, in the directory /com/yourdomain/game/.

  In Player.as, class com.yourdomain.game.Player

  { // Class body not shown... }

Package Access and the Classpath

- In order for a .fla file to access a class in a package, *the root directory of the package must reside within the classpath.*
- The classpath is simply the ***list of directories (paths)*** on your hard drive in which the compiler looks for classes.
- Entries in the classpath can be *added globally* for all documents (the *global classpath*)
- The global classpath includes the following directories:
  - *The **current directory**, usually represented by a period (.)—i.e., the directory in which the .fla resides*
  - *The Classes directory, **$(LocalData)/**Classes—*
- Select Edit -> Preferences -> ActionScript -> Language ->ActionScript 2.0 Settings.
- Under Classpath, click the plus sign (+) to add a new classpath entry. A blank entry appears.
- In the classpath entry, type **C:\data\actionscript**. (Or use the crosshair button to browse to that directory.)

EXCEPTIONS
The Exception-Handling Cycle

- we can write code that generates standardized errors via the ***throw*** statement.
- We handle those errors via the ***try/catch/finally*** statement.
- To generate an exception (i.e., signal an error) in our code,

  throw expression

- The ***Error*** class, introduced as a ***built-in class*** in Flash Player is a standard class for representing ***error conditions***.

```
class Box {
  private var width:Number;
  public function setWidth (w:Number):Boolean {
    if ((isNaN(w) || w == null) || (w <= 0 || w >
                                    Number.MAX_VALUE)) {
      // Invalid data, so quit.
      return false;    }
    width = w;
    return true;  }}
```

The Exception-Handling Cycle (Contd..)

- The *Error* class defines two ***public properties, name and message*** , used to describe the error.
- When the ***throw*** statement executes, program control is immediately ***transferred to a special section of code*** that knows how to deal with the problem.
- The code that deals with the problem is said to ***handle the exception.***

```
public function setWidth (w:Number):Void {
  if ((isNaN(w) || w == null) || (w <= 0 || w >
Number.MAX_VALUE)) {
        -
    throw new Error("Invalid width specified.");
  } width = w; }
```

The Exception-Handling Cycle (Contd..)

```
var b:Box = new Box( );
var someWidth:Number = -10;
try {
  b.setWidth(someWidth);
  trace("Width set successfully.");
} catch (e:Error) {
  // ERROR! Invalid data. Display a warning.
  trace("An error occurred: " + e.message); }
```

- The ***catch*** block handles exceptions generated by the **try block.**
- The code in the *catch* block executes if, and only if, code in the **try block generates an exception**

The Exception-Handling Cycle (Contd..)

- The ***try*** keyword tells the interpreter that we're about to ***execute some code*** that might *generate an exception.*
- The **catch** block *handles exceptions generated* by the *try* block.
- *The code in the catch block executes if, and only if, code in the try block generates an exception.*

```
var b:Box = new Box( );
var someWidth:Number = -10;
try {
  b.setWidth(someWidth);
  trace("Width set successfully.");
} catch (e:Error) {
  trace("An error occurred: " + e.message);}
```

```
try {  // Code here might cause an exception.     }

catch (e:Error) {
  trace("An error occurred: " + e.message);}
```

The Exception-Handling Cycle (Contd..)

- Code in a *try* clause *invokes a function that might throw an exception.*
- *Code in the invoked function throws an exception* using the *throw* statement if an error occurs.
- The *catch* block *deals with any errors that occur* in the *try* block.

```
var x:Number = 0;
var y:Number = 0;
var e:Error;
try {
  if (isNaN(x/y)) {
    throw new Error("Quotient is NaN.");
  } else {
    trace ("Result is " + String(x/y));
  }
} catch (e:Error) {
  trace("Error: " + e.message);
}
```

The Exception-Handling Cycle (Contd..)

- Control returns *either to the try block or the catch block*
- When the *catch* block is executed, *it receives the expression of the throw statement as a parameter*.
- Within a *try* block, if a statement executes, you can *safely trust that all preceding statements have executed* successfully.
- If code in a *try* block (or a method invoked in the *try* block) throws an error, *the remaining statements in the try block are skipped and the statements in the catch block are executed*.
- If no exception is thrown, *the try block completes and execution resumes with the statements immediately following the try/catch/finally statement.*

Handling Multiple Types of Exceptions

- Each *try* block can have *any number of supporting catch blocks.*
- When an exception is thrown in a *try* block that ***has multiple catch blocks***, the interpreter executes the *catch* block whose *parameter's datatype matches the datatype of the value originally thrown.*
- The datatypes of the *catch* parameter and thrown value are considered a ***match if they are identical*** or if the *catch* parameter type is a *superclass* of the thrown value's type.
- The syntax of a *try* statement with multiple *catch* blocks:

```
try {  // Code that might generate an exception.       }
catch (e:ErrorType1) {  // Error-handling code for ErrorType1.}
catch (e:ErrorType2) {  // Error-handling code for ErrorType2.}
catch (e:ErrorTypen) {  // Error-handling code for ErrorTypen.}
```

Handling Multiple Types of Exceptions (Contd..)

```
try {
  b.setWidth(someWidth);
  trace("Width set successfully.");
}
catch (e:Error) {
  switch (e.message) {
    case "Illegal Box dimension specified.":
    trace("An error occurred: " + e.message);
    trace("Please specify a valid dimension.");
    break;

    case "Box dimensions must be greater than 0.":
    trace("An error occurred: " + e.message);
    trace("Please specify a larger value.");
    break;

    case "Box dimensions must be < Number.MAX_VALUE.":
    trace("An error occurred: " + e.message);
    trace("Please specify a smaller value.");
    break;
  }}
```

Exception Bubbling

- The exceptions in a *try* block can be ***thrown either directly or as the result of a method call***.
- An exception can be ***thrown*** anywhere in an ActionScript program, even on a frame in a ***timeline.***
- ***if no catch block exists?*** These mysteries are resolved through the magic of ***exception bubbling***.
- When a *throw* statement executes, the interpreter immediately *stops normal program flow and looks for an enclosing try block*.

```
throw new Error("Something went wrong");
```

- If the *throw* statement is enclosed in a *try* block, the interpreter next tries to ***find a catch block whose parameter's datatype matches*** the datatype of the value thrown

```
try {  throw new Error("Something went wrong"); }
```

Exception Bubbling (Contd…)

- *If a matching catch block is found*, the interpreter transfers program control to that block:

```
try {  throw new Error("Something went wrong");
} catch (e:Error) { //Handle problems… }
```

- But if a ***matching catch block cannot be found*** or if the *throw* statement did not appear within a *try* block in the first place, ***then the interpreter checks whether the throw statement occurred within a method or function.***

```
public function doSomething ( ):Void {
throw new Error("Something went wrong");   }
class ErrorDemo {  public function doSomething ( ):Void {
    trace("About to throw an exception from doSomething( ) ");
    throw new Error("Something went wrong");  }
```

Exception Bubbling (Contd…)

```
public static function startApp ( ):Void {
    try {
      var demo:ErrorDemo = new ErrorDemo( );
      demo.doSomething( );
    } catch (e:Error) {
      trace("Exception caught in startApp( ), thrown by
doSomething( ).");    }  }}

class ErrorDemo {
  public function doSomething ( ):Void {
    trace("About to throw an exception from doSomething( )");
    throw new Error("Something went wrong");  }
  public static function startApp ( ):Void {
    var demo:ErrorDemo = new ErrorDemo( );
    demo.doSomething( );   } }
try {
  ErrorDemo.startApp( );
} catch (e:Error) {
  trace("Exception caught Error.startApp( ) was invoked.");}
```

Uncaught Exceptions

- If the interpreter never finds a *catch* block that can handle the thrown exception
- Sends the value of the thrown *Error* object's message property to the Output panel
- Aborts execution of all code currently remaining in the call stack

```
class ErrorDemo {
  public function doSomething ( ):Void {
    throw new Error("Something went wrong");  }
  public static function startApp ( ):Void {
    doSomething( );  }}
ErrorDemo.startApp( );
```

The finally Block

- A *try* block contains code that might throw an exception, and a *catch* block executes code in response to a thrown exception.
- The *finally* block, by comparison, always executes, whether or not code in the *try* block throws an exception.
- The *finally* block is placed once (and only once) as the last block in a *try/catch/finally* statement.

```
try {
 // Statements
} catch (e:ErrorType1) {   // Handle ErrorType1 exceptions.
} catch (e:ErrorTypen) {   // Handle ErrorTypen exceptions.
} finally {// code always executes, how the try block exits.
}
```

- Misplacing the *finally* block causes a compile-time error.

The *finally* block executes in one of the 4 circumstances

- Immediately after the **try block completes without errors**
- Immediately after a **catch block handles an exception generated in the try block**
- Immediately ***before an uncaught exception bubbles up***
- Immediately before a **return, continue, or break statement transfers control** out of the **try or catch blocks**
- The *finally* block of a *try/catch/finally* statement typically ***contains cleanup code that must execute whether or not an exception occurs in the corresponding try block.***

```
public function attackEnemy (enemy:SpaceShip):Void {
   try {
setCurrentTarget(enemy);     fireOnCurretTarget( );
   } finally {    setCurrentTarget(null);   }}
```

Limitations of Exception Handling in ActionScript 2.0

No Checked Exceptions

- In ActionScript 2.0, if a method throws an exception, it's up to the programmer to know about it in advance and handle it appropriately.
- The compiler will make no complaints if an exception is not handled.

No Built-in Exceptions

- Flash doesn't throw an exception when a developer attempts an illegal operation, such as dividing by zero. Neither the class library built into the Flash Player nor the Flash MX 2004 v2 components throw exceptions.

Exception Performance Issues

- The only exceptions generated in Flash are those thrown by ***custom-written*** programs.
- First, Flash must relay all error information to the programmer using unwieldy ***error codes or return values***.
- Second Flash often *fails silently when a runtime error occurs*.

Exception Performance Issues

- If a method or function encounters an error, it returns a *message or code describing the problem* and *expects the caller to know how to interpret that code*
- Both *exception handling and error codes* are useful and can be used together in different parts of the *same program*, depending upon the performance needs of different parts of your code.

Exception handling benchmark test

```
// The class file, in ExceptionPerformanceTest.as.
class ExceptionPerformanceTest {
  public function test1 ( ):Void {
     throw new CustomException( );  }
  public function test2 ( ):Boolean {
     return false;  }}

// The exception class in CustomException.as.
class CustomException extends Error {
  public var message:String = "This is the error message."; }

// The test code, in ExceptionPerformanceTest.fla.
var ept:ExceptionPerformanceTest = new ExceptionPerformanceTest(
);
var count1:Number = 0;
var count2:Number = 0;
var start1:Number = getTimer( );
```

Exception handling benchmark test

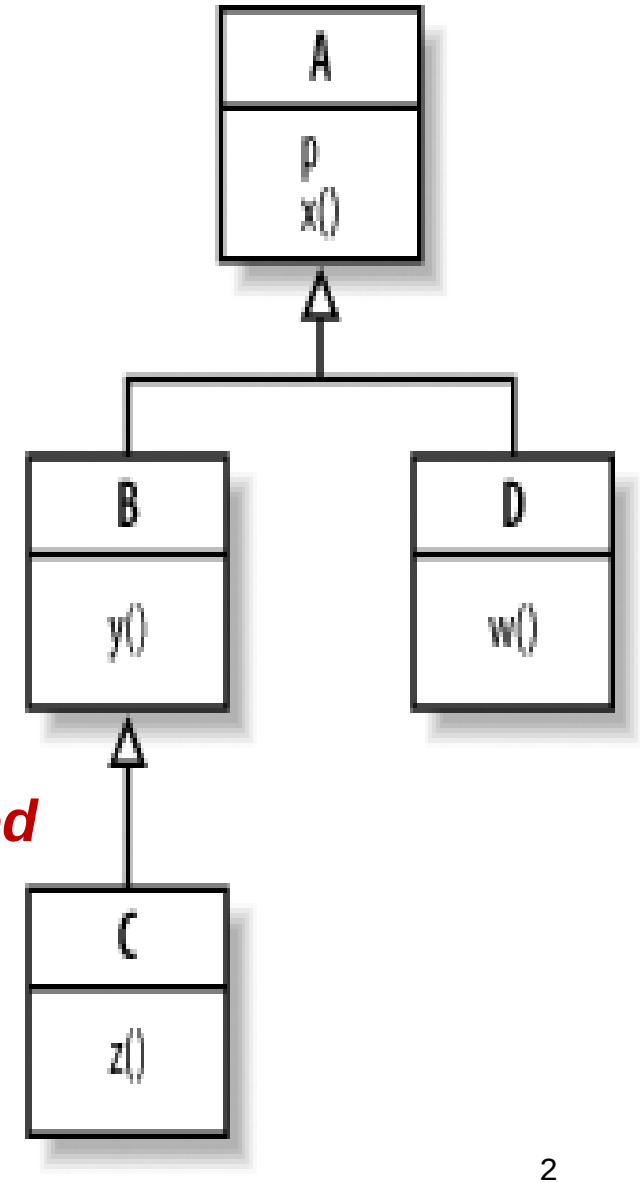
```
for (var i:Number = 0; i < 1000; i++) {
  try {
    ept.test1( );
  } catch (e:Error) {
    count1++;
  }}

var elapsed1 = getTimer( ) - start1;
var start2:Number = getTimer( );
for (var i:Number = 0; i < 1000; i++) {
  if (!ept.test2( )) {
    count2++;
  }}
var elapsed2 = getTimer( ) - start2;
trace(elapsed1);  // On my Pentium III 700, displays: 187
trace(elapsed2); // On my Pentium III 700, displays: 57
```

Unit IV

Inheritance
Authoring an ActionScript 2.0 Subclass
Interfaces
Packages
Exceptions

INHERITANCE

- *A Primer on Inheritance*
- *Subclasses as Subtypes*
- *An OOP Chat Example*
- *Overriding Methods and Properties*
- *Constructor Functions in Subclasses*
- *Subclassing Built-in Classes*
- *Augmenting Built-in Classes & Objects*
- *The Theory of Inheritance*
- *Abstract and Final Classes Not Supported*



A Primer on Inheritance

- In OOP, inheritance is a formal ***relationship between two or more classes***, wherein *one class borrows (or inherits) the property and method definitions of another class*

```
class A {    public var p:Number = 10;
             public function x ( ):Void {
             trace("Method x( ) was called.");  }}
```

- We can create an instance of class *A*, invoke method *x( )*, and access property p like this:

```
var aInstance:A = new A( );
aInstance.x( );       // Displays: Method x( ) called.
trace(aInstance.p);  // Displays: 10
```

- Now let's add a second class, *B*, that inherits method *x( )* and property p from class *A*.

A Primer on Inheritance (Contd…)

- To set up the inheritance *relationship between A and B*, we use the ***extends*** keyword to indicate that class *B* inherits class *A*'s method and property definitions:

```
class B extends A {                    }
```

- Because class *B* extends (inherits from) class *A*, *instances of B can automatically use the method x( )* and the property p

```
var bInstance:B = new B( );
bInstance.x( );          // Displays: Method x( ) was called.
trace(bInstance.p);  // Displays: 10
```

- class *B* would define its *own methods and/or properties* in addition to inheriting *A*'s methods and properties.

```
class B extends A {
public function y ( ):Void {trace("Method y( ) was called.");  } }
```

A Primer on Inheritance (Contd…)

- Now instances of *B* can use all the methods and properties of both *B* and its superclass, *A*:

```
var bInstance:B = new B( );
// Invoke inherited method, defined by class A.
bInstance.x( );       // Displays: Method x( ) was called.
// Invoke method defined by class B.
bInstance.y( );       // Displays: Method y( ) was called.
// Access inherited property.
trace(bInstance.p);  // Displays: 10
```

- Inheritance relationship between two classes, the extended class (in our case, class *A*) is called the ***base class***, and the class that does the extending (in our case, class *B*) is called the ***derived class.***

A Primer on Inheritance (Contd…)

- However, the terms *"base class" and "derived class"* have several synonyms, including superclass and subclass, parent and child, and type and subtype.
- The following code shows *third class,C, that extends class B and also defines a new method, z( ).*
- Class *C* can use all the *methods and properties defined by itself, its* ***superclass (B),*** *or its superclass's* ***superclass (A)***
- Every class in ActionScript (***both built-in and custom***) inherits directly or indirectly from the root of the built-in hierarchy: *Object*.
- The *Object* class defines some very basic methods that ***all classes can use.***

A Primer on Inheritance (Contd…)

```
class C extends B {
        public function z ( ):Void {
        trace("Method z( ) was called.");  } }
var cInstance:C = new C( );
cInstance.x( );  // Displays: Method x( ) was called.
cInstance.y( );  // Displays: Method y( ) was called.
cInstance.z( );  // Displays: Method z( ) was called.
trace(cInstance.p);  // Displays: 10
```

- class *D* inherits directly from class *A*. Class *D* uses the methods and properties defined by itself and by its superclass, *A*.

```
class D extends A {
        public function w ( ):Void {
            trace("Method w( ) was called.");  }}
```

Subclasses as Subtypes

- A subclass is considered a ***subtype*** of its superclass. An instance of the subtype *can be used anywhere its superclass's type is expected.* ***var aInstance:A;***
- we can legally assign that variable an instance of any subclass of class *A*: ***aInstance = new B( );***
- The preceding assignment works because the compiler knows that *any instance of class B has (through inheritance) all the methods and properties defined by class A.*
- However, ***the reverse is not true***. We cannot assign an instance of *A* to a variable whose datatype is *B*:

  var bInstance:B = new A( ); // Type Mismatch Error.
- That assignment does not work because the *compiler cannot guarantee that an instance of class A will have the methods and properties defined by class B.*

An OOP Chat Example

- *Consider a chat application that defines a class, ChatRoom, which handles communication among a group of users.*
- *ChatRoom class defines methods for displaying & sending chat messages and for managing a list of users in the room*

Table 6-1. ChatRoom class's property and methods

Method or property name	Purpose
`userList`	Property storing a reference to the on-screen *List* component that displays users in the room.
displayMessage( )	Displays an incoming user message in a text field.
sendMessage( )	Sends an outgoing message to other users.
onAddUser( )	Adds the new user to the `userList`. Invoked when a user joins the room.
onRemoveUser( )	Removes the departed user from the `userList`. Invoked when a user leaves the room.
onUserChangeName( )	Changes the user's name in the `userList`. Invoked when a user's name changes.

AvatarChatRoom class's methods & properties

Method or property name	Inheritance relationship	Original behavior	Behavior added by *AvatarChatRoom* class
`userList`	Inherited property	Stores a reference to the on-screen *List* component that displays users in the room	None
`avatars`	Property added by subclass	Not applicable	Stores a list of *Avatar* instances displayed on screen
displayMessage( )	Overridden method	Displays an incoming user message in a text field	Also displays a message bubble next to the user's avatar
sendMessage( )	Inherited method	Sends an outgoing message to other users	None
onAddUser( )	Overridden method	Adds the new user to the `userList`	Also displays a new avatar on the screen
onRemoveUser( )	Overridden method	Removes the departed user from the `userList`	Also removes the user's avatar on the screen
onUserChangeName( )	Overridden method	Changes the user's name in the `userList`	Also changes the name displayed under an avatar
onUserChangePosition ( )	Method added by subclass	Not applicable	Positions the user's avatar on screen

Example An inheritance example using ChatRoom and AvatarChatRoom

```
class ChatRoom {
private var userList:mx.controls.List;
public function displayMessage(userID:String, msg:String):Void {
        trace("Displaying chat message in chat text field.");
}
public function sendMessage(msg:String):Void {
        trace("Sending chat message.");
}
public function onAddUser(userID:String):Void {
        trace("Adding user to userList.");
}
public function onRemoveUser(userID:String):Void {
        trace("Removing user from userList.");
}
public function onUserChangeName(userID:String, newName:String):Void {
        trace("Changing name in userList.");
        }
}
```

Example An inheritance example using ChatRoom and AvatarChatRoom

```
class AvatarChatRoom extends ChatRoom {
  private var avatars:Array;
  public function displayMessage (userID:String,
msg:String):Void {
    super.displayMessage(userID, msg);
    trace("Displaying message in avatar text bubble.");
  }
  public function onAddUser (userID:String):Void {
    super.onAddUser(userID);
    trace("Creating avatar for new user.");
  }
  public function onRemoveUser (userID:String):Void {
    super.onRemoveUser(userID)
    trace("Removing avatar for user.");
  }
  public function onUserChangeName (userID:String,
newName:String):Void {
    super.onUserChangeName(userID, newName);
    trace("Changing name on avatar.");
  }
  public function onUserChangePosition (userID:String,
                                        newX:Number,
newY:Number):Void {
    trace("Repositioning avatar.");
  }}
```

Constructor Function in Subclasses

- A constructor function initializes the instances of a class by:
  - *Calling methods that perform setup tasks*
  - *Setting properties on the object being created*
- When a class is extended, the subclass can define a constructor function of its own. A subclass constructor is expected to:
  - *Perform setup tasks related to the subclass*
  - *Set properties defined by the subclass*
  - *Invoke the superclass constructor (superconstructor)*
- In all inheritance relationships, setup and property initialization *relating to the superclass occur in the superclass constructor, not in the subclass constructor.*

Constructor Function in Subclasses (Contd..)

- A subclass constructor function, if specified, is formally required to invoke its *superclass constructor*, via the keyword ***super***, as the first statement in the function.

```
class A {  public function A ( ) {  }       }
class B extends A {   public function B ( ) {
    // Invoke superclass's constructor function explicitly
    super( );  }}

public function B ( ) {
  // Invoke superclass's constructor function explicitly.
  super( );}
public function B ( ) {
 // No constructor call. The compiler provides one implicitly. }
```

The Theory of Inheritance

- Inheritance lets us ***separate a core feature set*** from customized versions of that feature set. Code for the *core is stored in a superclass while code for the customizations is kept neatly in a subclass.*
- More than one subclass can extend the superclass, *allowing multiple customized versions of a particular feature set to exist simultaneously.*
- Inheritance also lets us express the *architecture* of an application in *hierarchical terms* that mirror the real world and the human psyche.
- In addition *to code reuse and logical* hierarchy, inheritance allows instances of different *subtypes to be used where a single type is expected*. Known as ***polymorphism***

Polymorphism and Dynamic Binding

- The word ***polymorphism*** itself means literally ***"many forms***"—*each single object can be treated as an instance of its own class or as an instance of any of its superclasses*.
- Polymorphism's partner is ***dynamic binding***, which guarantees that a *method invoked on an object will trigger the behavior defined by that object's actual class.*
- Dynamic binding, which occurs at ***runtime***, with *static type checking, which occurs at compile time*
- *Dynamic binding* is the runtime process by which each invocation of *draw( )* is associated with the appropriate implementation of that method.
- Dynamic binding is often called ***late binding:*** the method call is *bound* to a particular implementation "late" (i.e., at runtime).

Polymorphism and Dynamic Binding(Contd..)

```
class Shape {   public function draw ( ):Void {
    // No implementation.   }      }
class Circle extends Shape {
  public function draw ( ):Void {
  }}

class Rectangle extends Shape {
  public function draw ( ):Void {
  }}
class Triangle extends Shape {
  public function draw ( ):Void {
  }}
```

Abstract and Final Classes Not Supported

- An abstract class is any class that defines one or more *abstract* ***methods—methods that have a signature and a return type but no implementation***
- A class that wishes to extend an abstract class must implement ***all of the superclass's abstract methods***; otherwise, a compile-time error occurs.
- ActionScript 2.0 ***does not yet support abstract classes*** or abstract methods. Instead we simply define a method with no code in its body
- A ***final method*** is a method that *must not be implemented by a subclass.* Final methods are used to *prevent programmers from creating subclasses that accidentally introduce problems into the behavior of a superclass*

Authoring an ActionScript 2.0 Subclass Extending ImageViewer's Capabilities

- The list of possible functional requirements for our *ImageViewer* class:
  - Load an image , Display an image
  - Crop an image to a particular rectangular "view region"
  - Display a border around the image
  - Display image load progress
  - *Reposition the view region, Resize the view region*
  - Pan (reposition) the image within the view region
  - Zoom (resize) the image within the view region
- The *ImageViewer* class already implements the first five requirements, but it stops there.
- In the *ImageViewerDeluxe* class, we'll add the next two items on the list: *the abilities to reposition and resize the viewer.*

The ImageViewerDeluxe Skeleton

- The bare bones of any subclass requires:
  - *A class declaration that defines the inheritance relationship*
  - *A constructor that invokes the superclass constructor with the expected arguments*

```
class ImageViewerDeluxe extends ImageViewer {
  public function ImageViewerDeluxe (target:MovieClip,
    depth:Number,  x:Number, y:Number, w:Number, h:Number,
    borderThickness:Number, borderColor:Number)  {
    super(target, depth, x, y, w, h, borderThickness,
    borderColor);
} }
```

To create the ImageViewerDeluxe.as class file using Flash MX Professional 2004, follow these steps:

1. Choose File → New.
2. In the New Document dialog box, on the General tab, for Type, choose ActionScript File.
3. Click OK. The script editor launches with an empty file.
4. Copy the code into the script editor: in the next slide.
5. Choose File → Save As.
6. In the Save As dialog box, specify **ImageViewerDeluxe.as** as the filename (which is case sensitive), and save the file in the imageviewer folder that you created.
7. Instances of the *ImageViewerDeluxe* class have all the *ImageViewer* class's methods and can be used anywhere *ImageViewer* instances are used.

Adding setPosition( ) and setSize( ) Methods

- ***setPosition( )***
  - Changes the location of the viewer by assigning the (x, y) coordinates of the viewer's main container movie clip
- ***setSize( )***
  - Changes the size of the viewer by re-creating the mask and border over the image movie clip (i.e., by recropping the image)

```
public function setPosition (x:Number, y:Number):Void {
    container_mc._x = x; container_mc._y = y;          }
```

- The *setSize( )* method, which resizes the image viewer by recropping the image:

```
public function setSize (w:Number, h:Number):Void {
    createImageClipMask(w, h); createBorder(w, h);
    container_mc.image_mc.setMask(container_mc.mask_mc);
    }
```

The ImageViewerDeluxe class, final version

```
class ImageViewerDeluxe extends ImageViewer {
  public function ImageViewerDeluxe (target:MovieClip,
    depth:Number,  x:Number, y:Number, w:Number, h:Number,
    borderThickness:Number, borderColor:Number)  {
    super(target, depth, x, y, w, h, borderThickness,
    borderColor);  }    }
public function setPosition (x:Number, y:Number):Void {
        container_mc._x = x;
        container_mc._y = y;                                    }
public function setSize (w:Number, h:Number):Void     {
        createImageClipMask(w, h); createBorder(w, h);
    container_mc.image_mc.setMask(container_mc.mask_mc);
                                                                }
```

ImageViewerDeluxe class, final version (Contd..)

```
public function getImageWidth ( ):Number {
        return container_mc.image_mc._width;   }
public function getImageHeight ( ):Number {
        return container_mc.image_mc._height;  }
public function setShowFullImage(show:Boolean):Void {
        showFullImage = show;                  }
public function getShowFullImage( ):Boolean {
       return showFullImage;       }
public function scaleViewerToImage ( ):Void {
        setSize(getImageWidth( ), getImageHeight( ));    }
public function onLoadInit (target:MovieClip):Void {
       super.onLoadInit(target);
       if (showFullImage) {   scaleViewerToImage( ); } } }
```

INTERFACES

The Case for Interfaces

- An *interface* is an ActionScript 2.0 language construct used to ***define a new datatype***, much as a class defines a datatype.
- A class both defines a ***datatype and provides the implementation*** for it, an interface *defines a datatype in* ***abstract terms*** only; an interface provides *no implementation for the datatype.*
- An interface, instead of providing its own implementation, is ***adopted by one or more classes*** that agree to provide the implementation.
- A class that provides an ***implementation for an interface*** belongs both to its own datatype and to the datatype defined by the interface.

The Case for Interfaces (Contd..)

```
class OrderListener {
  public function onOrderError ( ):Void {
    // Generic implementation of onOrderError( ), not shown…  }}
class StatsTracker extends OrderListener {
  // Override OrderListener.onOrderError( ).
  public function onOrderError ( ) {
    // Send problem report to database. Code not shown…  }}
class OrderProcessor {
  public function addListener (ol:OrderListener):Boolean {
    // Code here should register ol to receive OrderProcessor
    events,
    // and return a Boolean value indicating whether registration  }}
class OrderUI extends MovieClip {
  public function onOrderError ( ) {
    // Display problem report on screen, not shown…  }}
```

Interfaces and Multidatatype Classes

- An interface is simply a list of methods.

```
interface OrderListener {
  function onOrderError( );
}
```

- Classes use the *implements* keyword to enter into an agreement with an interface, promising to define the methods it contains.

```
class OrderUI extends MovieClip implements OrderListener {
    public function onOrderError ( ) {
      // Display problem report on screen, not shown...
    }}
```

Interface Syntax and Use

- To create an interface in ActionScript 2.0, we use the *interface* keyword, using the following syntax:

```
interface  SomeName  {
  function method1  (param1 :datatype ,...paramn :datatype ):returnType ;
  function method2  (param1 :datatype ,...paramn :datatype ):returnType ;
     ...
  function methodn  (param1 :datatype ,...paramn :datatype ):returnType ;
}
```

- In interfaces, method declarations do not (and must not) include curly braces.
- All methods declared in an interface must be *public*; hence, by convention, the attribute *public* (which is the default) is omitted from method definitions in an interface.

Interface Syntax and Use (Contd..)

- ***Class files***
  - Contain the method and property definitions for each class. They start with the *class* keyword.
- ***Intrinsic files***
  - Contain method signatures strictly for the purpose of satisfying the compiler's type checking. They start with the *intrinsic* keyword.
- ***Interface files***
  - Contain a list of methods to be implemented but not the implementations themselves. They start with the *interface* keyword.
- A class that wishes to adopt an interface's datatype *must agree to implement that interface's methods.*

Interface Syntax and Use (Contd..)

- The class uses the *implements* keyword, which has the following syntax:

  class SomeName implements SomeInterface { }

- The implementing class's method definitions must match the interface's method definitions exactly, including number of parameters, parameter types, and return type.
- If differs between the interface and the implementing class, the compiler generates an error.
- A class can legally implement more than one interface by separating interface names with commas, as follows:

```
class SomeName implements SomeInterface,
                          SomeOtherInterface {
}
```

P A C K A G E S

- Larger applications, creates dozens of different ***classes and packages grouping them logically together.***
- ***To group classes and interfaces together, we use packages.***
- *namespace collision* or *naming conflict*.
- A package is a unique place to ***put a group of classes, much as a directory on your hard drive is a unique place*** to put a group of files.
- Packages allow multiple classes of the same name to ***coexist without namespace collisions***, because each package constitutes a *namespace* within which the class name is unique.
- A namespace is a set of names that contains *no duplicates*.

Package Syntax

- The ***Domain Name System (DNS)*** defines a namespace in which registrants can *reserve a domain nam*e, such as moock.org
- Every package has a name that *describes the classes it contains.*
- By convention, package names ***do not use initial caps*** but might use mixed case, such as greatUtils.
- To refer to a specific class within a package, we use the dot operator:

  packagename.ClassName

- game.Player Packages can be nested to form a hierarchy; that is, a package can contain another package.

  game.vehicles.SpaceShip

- By nesting packages, we can organize our classes into discrete subcategories within a larger category, which again prevents naming conflicts.

Import statement

- An ***unqualified reference*** includes just the class name.

  var ship:SpaceShip = new SpaceShip( );

- If a class is in a package, we must use a fully qualified reference when creating instances of the class and specifying the ***datatypes of variables, parameters, and return types.***

  var ship:game.vehicles.SpaceShip = new game.vehicles.SpaceShip( );// Fully qualified

- The ***import*** statement lets us use an *unqualified class* reference as an ***alias*** for a fully qualified class reference. The basic syntax of *import* is:

  import packagename.ClassName;

  import game.vehicles.SpaceShip;

Importing an Entire Package

- To import all the classes in a package, use the wildcard character, *:

  import packagename.*;

- This wildcard syntax lets us refer to any class in *packagename* by its unqualified name rather than its fully qualified name.

  import geometry.Circle;
  import geometry.Triangle;
  import geometry.Rectangle;

  import geometry.*;

- If we use package *wildcards to import two classes with the same name but from different packages*, we again encounter a ***namespace collision***.

Package Naming Conventions

```
import game.vehicles.*;
import ordering.*;
var s:Ship = new Ship( );  // new game.vehicles.Ship
var s:ordering.Ship = new ordering.Ship( );
```

- To avoid confusion, we should use ***fully qualified names*** for all references to *Ship*.
- *Just as two class names can conflict, two package names can conflict.*
- To avoid package or class naming conflicts you should use the convention of ***placing all your classes and packages in a package named after your organization's domain name.***
- com.acme com precedes acme (the package name is the reverse of the domain name).

Defining Packages

- Defining a package for a class requires *two* general steps:
  - *Create a directory structure whose name identically matches the package hierarchy.*
  - *Specify the package name when defining the class.*
- There is no such thing as a package file. A package is a concept based on placing a given class's .as file in the appropriate folder.
- To place the class *Player* in package com.yourdomain.game

  Create a new directory named com.

  Create a new directory inside com named yourdomain.

  Create a new directory inside yourdomain named game.

  Create a text file, Player.as, in the directory /com/yourdomain/game/.

  In Player.as, class com.yourdomain.game.Player

  { // Class body not shown... }

Package Access and the Classpath

- In order for a .fla file to access a class in a package, *the root directory of the package must reside within the classpath.*
- The classpath is simply the ***list of directories (paths)*** on your hard drive in which the compiler looks for classes.
- Entries in the classpath can be *added globally* for all documents (the *global classpath*)
- The global classpath includes the following directories:
  - *The **current directory**, usually represented by a period (.)—i.e., the directory in which the .fla resides*
  - *The Classes directory, **$(LocalData)/**Classes—*
- Select Edit -> Preferences -> ActionScript -> Language ->ActionScript 2.0 Settings.
- Under Classpath, click the plus sign (+) to add a new classpath entry. A blank entry appears.
- In the classpath entry, type **C:\data\actionscript**. (Or use the crosshair button to browse to that directory.)

EXCEPTIONS
The Exception-Handling Cycle

- we can write code that generates standardized errors via the ***throw*** statement.
- We handle those errors via the ***try/catch/finally*** statement.
- To generate an exception (i.e., signal an error) in our code,

  throw expression

- The ***Error*** class, introduced as a ***built-in class*** in Flash Player is a standard class for representing ***error conditions***.

```
class Box {
  private var width:Number;
  public function setWidth (w:Number):Boolean {
    if ((isNaN(w) || w == null) || (w <= 0 || w >
                                    Number.MAX_VALUE)) {
      // Invalid data, so quit.
      return false;    }
    width = w;
    return true;  }}
```

The Exception-Handling Cycle (Contd..)

- The *Error* class defines two ***public properties, name and message*** , used to describe the error.
- When the ***throw*** statement executes, program control is immediately ***transferred to a special section of code*** that knows how to deal with the problem.
- The code that deals with the problem is said to ***handle the exception.***

```
public function setWidth (w:Number):Void {
  if ((isNaN(w) || w == null) || (w <= 0 || w >
Number.MAX_VALUE)) {
        -
    throw new Error("Invalid width specified.");
  } width = w; }
```

The Exception-Handling Cycle (Contd..)

```
var b:Box = new Box( );
var someWidth:Number = -10;
try {
  b.setWidth(someWidth);
  trace("Width set successfully.");
} catch (e:Error) {
  // ERROR! Invalid data. Display a warning.
  trace("An error occurred: " + e.message); }
```

- The ***catch*** block handles exceptions generated by the **try block.**
- The code in the *catch* block executes if, and only if, code in the **try block generates an exception**

The Exception-Handling Cycle (Contd..)

- The ***try*** keyword tells the interpreter that we're about to ***execute some code*** that might *generate an exception.*
- The **catch** block *handles exceptions generated* by the *try* block.
- *The code in the catch block executes if, and only if, code in the try block generates an exception.*

```
var b:Box = new Box( );
var someWidth:Number = -10;
try {
  b.setWidth(someWidth);
  trace("Width set successfully.");
} catch (e:Error) {
  trace("An error occurred: " + e.message);}
```

```
try {  // Code here might cause an exception.     }

catch (e:Error) {
  trace("An error occurred: " + e.message);}
```

The Exception-Handling Cycle (Contd..)

- Code in a *try* clause *invokes a function that might throw an exception.*
- *Code in the invoked function throws an exception* using the *throw* statement if an error occurs.
- The *catch* block *deals with any errors that occur* in the *try* block.

```
var x:Number = 0;
var y:Number = 0;
var e:Error;
try {
  if (isNaN(x/y)) {
    throw new Error("Quotient is NaN.");
  } else {
    trace ("Result is " + String(x/y));
  }
} catch (e:Error) {
  trace("Error: " + e.message);
}
```

The Exception-Handling Cycle (Contd..)

- Control returns *either to the try block or the catch block*
- When the *catch* block is executed, *it receives the expression of the throw statement as a parameter*.
- Within a *try* block, if a statement executes, you can *safely trust that all preceding statements have executed* successfully.
- If code in a *try* block (or a method invoked in the *try* block) throws an error, *the remaining statements in the try block are skipped and the statements in the catch block are executed*.
- If no exception is thrown, *the try block completes and execution resumes with the statements immediately following the try/catch/finally statement.*

Handling Multiple Types of Exceptions

- Each *try* block can have *any number of supporting catch blocks.*
- When an exception is thrown in a *try* block that ***has multiple catch blocks***, the interpreter executes the *catch* block whose *parameter's datatype matches the datatype of the value originally thrown.*
- The datatypes of the *catch* parameter and thrown value are considered a ***match if they are identical*** or if the *catch* parameter type is a *superclass* of the thrown value's type.
- The syntax of a *try* statement with multiple *catch* blocks:

```
try {  // Code that might generate an exception.       }
catch (e:ErrorType1) {  // Error-handling code for ErrorType1.}
catch (e:ErrorType2) {  // Error-handling code for ErrorType2.}
catch (e:ErrorTypen) {  // Error-handling code for ErrorTypen.}
```

Handling Multiple Types of Exceptions (Contd..)

```
try {
  b.setWidth(someWidth);
  trace("Width set successfully.");
}
catch (e:Error) {
  switch (e.message) {
    case "Illegal Box dimension specified.":
    trace("An error occurred: " + e.message);
    trace("Please specify a valid dimension.");
    break;

    case "Box dimensions must be greater than 0.":
    trace("An error occurred: " + e.message);
    trace("Please specify a larger value.");
    break;

    case "Box dimensions must be < Number.MAX_VALUE.":
    trace("An error occurred: " + e.message);
    trace("Please specify a smaller value.");
    break;
  }}
```

Exception Bubbling

- The exceptions in a *try* block can be ***thrown either directly or as the result of a method call***.
- An exception can be ***thrown*** anywhere in an ActionScript program, even on a frame in a ***timeline.***
- ***if no catch block exists?*** These mysteries are resolved through the magic of ***exception bubbling***.
- When a *throw* statement executes, the interpreter immediately *stops normal program flow and looks for an enclosing try block*.

```
throw new Error("Something went wrong");
```

- If the *throw* statement is enclosed in a *try* block, the interpreter next tries to ***find a catch block whose parameter's datatype matches*** the datatype of the value thrown

```
try {  throw new Error("Something went wrong"); }
```

Exception Bubbling (Contd…)

- *If a matching catch block is found*, the interpreter transfers program control to that block:

```
try {  throw new Error("Something went wrong");
} catch (e:Error) { //Handle problems… }
```

- But if a ***matching catch block cannot be found*** or if the *throw* statement did not appear within a *try* block in the first place, ***then the interpreter checks whether the throw statement occurred within a method or function.***

```
public function doSomething ( ):Void {
throw new Error("Something went wrong");   }
class ErrorDemo {  public function doSomething ( ):Void {
    trace("About to throw an exception from doSomething( ) ");
    throw new Error("Something went wrong");  }
```

Exception Bubbling (Contd…)

```
public static function startApp ( ):Void {
    try {
      var demo:ErrorDemo = new ErrorDemo( );
      demo.doSomething( );
    } catch (e:Error) {
      trace("Exception caught in startApp( ), thrown by
doSomething( ).");    }  }}

class ErrorDemo {
  public function doSomething ( ):Void {
    trace("About to throw an exception from doSomething( )");
    throw new Error("Something went wrong");  }
  public static function startApp ( ):Void {
    var demo:ErrorDemo = new ErrorDemo( );
    demo.doSomething( );   } }
try {
  ErrorDemo.startApp( );
} catch (e:Error) {
  trace("Exception caught Error.startApp( ) was invoked.");}
```

Uncaught Exceptions

- If the interpreter never finds a *catch* block that can handle the thrown exception
- Sends the value of the thrown *Error* object's message property to the Output panel
- Aborts execution of all code currently remaining in the call stack

```
class ErrorDemo {
  public function doSomething ( ):Void {
    throw new Error("Something went wrong");  }
  public static function startApp ( ):Void {
    doSomething( );  }}
ErrorDemo.startApp( );
```

The finally Block

- A *try* block contains code that might throw an exception, and a *catch* block executes code in response to a thrown exception.
- The *finally* block, by comparison, always executes, whether or not code in the *try* block throws an exception.
- The *finally* block is placed once (and only once) as the last block in a *try/catch/finally* statement.

```
try {
 // Statements
} catch (e:ErrorType1) {   // Handle ErrorType1 exceptions.
} catch (e:ErrorTypen) {   // Handle ErrorTypen exceptions.
} finally {// code always executes, how the try block exits.
}
```

- Misplacing the *finally* block causes a compile-time error.

The *finally* block executes in one of the 4 circumstances

- Immediately after the **try block completes without errors**
- Immediately after a **catch block handles an exception generated in the try block**
- Immediately ***before an uncaught exception bubbles up***
- Immediately before a **return, continue, or break statement transfers control** out of the **try or catch blocks**
- The *finally* block of a *try/catch/finally* statement typically ***contains cleanup code that must execute whether or not an exception occurs in the corresponding try block.***

```
public function attackEnemy (enemy:SpaceShip):Void {
   try {
setCurrentTarget(enemy);     fireOnCurretTarget( );
   } finally {    setCurrentTarget(null);   }}
```

Limitations of Exception Handling in ActionScript 2.0

No Checked Exceptions

- In ActionScript 2.0, if a method throws an exception, it's up to the programmer to know about it in advance and handle it appropriately.
- The compiler will make no complaints if an exception is not handled.

No Built-in Exceptions

- Flash doesn't throw an exception when a developer attempts an illegal operation, such as dividing by zero. Neither the class library built into the Flash Player nor the Flash MX 2004 v2 components throw exceptions.

Exception Performance Issues

- The only exceptions generated in Flash are those thrown by ***custom-written*** programs.
- First, Flash must relay all error information to the programmer using unwieldy ***error codes or return values***.
- Second Flash often *fails silently when a runtime error occurs*.

Exception Performance Issues

- If a method or function encounters an error, it returns a *message or code describing the problem* and *expects the caller to know how to interpret that code*
- Both *exception handling and error codes* are useful and can be used together in different parts of the *same program*, depending upon the performance needs of different parts of your code.

Exception handling benchmark test

```
// The class file, in ExceptionPerformanceTest.as.
class ExceptionPerformanceTest {
  public function test1 ( ):Void {
     throw new CustomException( );  }
  public function test2 ( ):Boolean {
     return false;  }}

// The exception class in CustomException.as.
class CustomException extends Error {
  public var message:String = "This is the error message."; }

// The test code, in ExceptionPerformanceTest.fla.
var ept:ExceptionPerformanceTest = new ExceptionPerformanceTest(
);
var count1:Number = 0;
var count2:Number = 0;
var start1:Number = getTimer( );
```

Exception handling benchmark test

```
for (var i:Number = 0; i < 1000; i++) {
  try {
    ept.test1( );
  } catch (e:Error) {
    count1++;
  }}

var elapsed1 = getTimer( ) - start1;
var start2:Number = getTimer( );
for (var i:Number = 0; i < 1000; i++) {
  if (!ept.test2( )) {
    count2++;
  }}
var elapsed2 = getTimer( ) - start2;
trace(elapsed1);  // On my Pentium III 700, displays: 187
trace(elapsed2); // On my Pentium III 700, displays: 57
```

Unit V

An OOP Application Framework
Using Components with ActionScript 2.0
MovieClip Subclasses

The Basic Directory Structure

- Create a ***directory named AppName*** on your hard drive. The AppName directory will contain *everything in our project*, including source code and final output
- In the AppName directory, ***create a subdirectory named deploy***. The deploy directory will contain the ***final, compiled application***, ready for posting to a web site or other distribution medium.
- In the AppName directory, create a ***subdirectory named source***. The source directory will contain all ***source code for the application***, including classes *(.as files)* and Flash documents *(.fla files).*

AppName/

deploy/

source/

The Flash Document (.fla file)

- Every Flash application must include at least one Flash document ***(.fla file)***.
- The *Flash document is the source file* from which a Flash movie ***(.swf file)*** is exported.
- The *Flash movie* is what's actually displayed in the *Flash Player* (i.e., the Flash runtime environment).
- In the Flash authoring tool, choose *File ->New.*
- In the *New Document dialog box*, on the General tab, for Type, choose *Flash Document*, then click OK.
- Use *File ->Save As* to save the Flash document as *AppName.fla* in the AppName/source directory.

The Classes

- In Flash, most applications are *visual and include graphical user interfaces*. Hence, the classes of an OOP Flash application typically ***create and manage user interface components.***
- An OOP application can have ***any number of classes***, but only one of them is used to ***start the application in motion***.
- We'll store our classes in the ***package com.somedomain.***
- To create the directories for the ***com.somedomain package***, follow these steps:
  - In the AppName/source directory, *create a subdirectory named com.*
  - In the AppName/source/com directory, *create a subdirectory named somedomain.*

The Classes (Contd..)

- *To create class A*, follow these steps if you're using Flash MX Professional 2004:
- In Flash MX Professional 2004, *choose File ->New.*
- In the *New Document dialog box*, on the General tab, for Type, *choose ActionScript File.*
- Click *OK*. The script editor *launches with an empty file*.
- Enter the following code into the script editor:

```
import com.somedomain.B;
class com.somedomain.A {
  private static var bInstance:B;
  public function A ( ) {
    // In this example, class A's constructor is not used.
  }
```

The Classes (Contd..)

public static function main ( ):Void {

trace("Starting application.");

bInstance = new B( ); } }

Choose File -> Save As.

- In the Save As dialog box, specify **A.as, B.as** as the filename, and save the file in the directory:

class com.somedomain.B {

public function B ( ) {

trace("An instance of class B was constructed."); } }

AppName/source/com/somedomain.

- As long as the directory in which they reside is added to the *global or document classpath*, the classes will be *accessible to timeline code and other ActionScript classes.*

The Document Timeline

- The *fundamental metaphor* of a Flash document is the *timeline*, which can be used to ***create animations like a filmstrip***.
- When used for animation, ***the frames in the timeline are displayed in rapid linear succession*** by the Flash Player.
- The timeline can also be used to *create a series of application states*, in which *specific frames correspond to specific states* and frames are *displayed according to the application's logic*
- To load our *A* and *B* classes, we'll follow these steps:
  - *Specify the export frame for classes in the movie.*
  - *Add the labeled state frames loading and main to AppName.fla's timeline.*
  - *Add code that displays a loading message while the movie loads.*

To specify the export frame for classes in the movie, follow these steps:

- Open AppName.fla in the Flash authoring tool.
- Choose File ->Publish Settings.
- In the Publish Settings dialog box, on the Flash tab, next to the ActionScript Version, click Settings.
- In the ActionScript Settings dialog box, for the Export Frame for Classes option, enter **10**.
- Click OK to confirm the ActionScript settings.
- Click OK to confirm the publish settings.

To add the labeled state frames loading and main to AppName.fla's timeline, follow these steps:

- In AppName.fla's main timeline, double-click *Layer 1* and rename it to **scripts**. We'll place all our code on the *scripts* layer.

The Document Timeline (Contd..)

- In the main timeline of AppName.fla, select frame 15 of the *scripts* layer.
- Choose Insert ->Timeline ->Keyframe (F6).
- Choose Insert ->Timeline ->Layer.
- Double-click the new layer's name and change it to **labels**.
- At frames 4 and 15 of the *labels* layer, add a new keyframe (Insert Timeline Keyframe or F6). Just as the *scripts* layer holds all our scripts, the *labels* layer is used exclusively to hold frame labels.
- With frame 4 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **loading**.
- With frame 15 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **main**.

steps to add code that displays a loading message while the movie loads:

- At frame 5 of the *scripts* layer, add a new keyframe (Insert ->Timeline ->Keyframe or F6).
- With frame 5 of the *scripts* layer selected, enter the following code into the Actions panel (F9):

```
if (_framesloaded == _totalframes) {
  gotoAndStop("main");
} else {   gotoAndPlay("loading");  }
```

- With frame 1 of the *scripts* layer selected, enter the following code into the Actions panel:

```
this.createTextField("loadmsg_txt", 0, 200, 200, 0, 0);
loadmsg_txt.autoSize = true;
loadmsg_txt.text = "Loading...Please wait.";
```

The Document Timeline (Contd..)

- With frame 15 of the *scripts* layer selected, enter the following code into the Actions panel:

  loadmsg_txt.removeTextField( );

- We've now provided the basic timeline structure that loads our application's classes.
- All that's left is to start the application by invoking *A.main( )*.
- We do that on the frame labeled *main* in AppName.fla.
- Add the following code to the end of the script on frame 15 (i.e., just below loadmsg_txt.removeTextField( );):

  import com.somedomain.A;

  A.main( );

- To test, we need to export a .swf file and run it in the Flash Player

The Exported Flash Movie (.swf file)

- Our application is now ready for testing and—assuming all goes well—deployment.
- To specify the directory in which to create AppName.swf, follow these steps:
  - *With AppName.fla open, choose File ->Publish Settings-> Formats.*
  - *Under the File heading, for Flash (.swf), enter **../deploy/AppName.swf**.*
  - *Click OK.*
- To test our application in the Flash authoring tool's Test Movie mode, select Control Test Movie.
- Testing a movie actually creates AppName.swf in the *AppName*/deploy directory and immediately loads it into a debugging version of the Flash Player.

publish an HTML page that includes the movie, ready for posting to a web site as follows:

With AppName.fla open, choose File Publish Settings Formats.

*Under the File heading, for HTML (.html), enter **../deploy/AppName.html**.*

Click Publish. Click OK.

- Test locally by opening *AppName.html* in your *web browser.*
- When the local testing proves successful, upload the *.html and .swf files to your web server.*
- *Packages and classpaths matter only at compile time*.
- We you can upload the .html and .swf wherever you like on your server, but in our example*, the two files must reside in the same web server folder.*

Projects in Flash MX Professional 2004

- To help manage the files in a large application, Flash MX Professional 2004 supports the concept of projects.
- A project is a group of related files that can be managed via the Project panel in the Flash MX Professional 2004 authoring tool.
- The Project panel resembles a file explorer and offers the following features:
  - *Authoring tool integration with source control applications such as Microsoft Visual SourceSafe*
  - *Easy access to related source files*
  - *One-click application publishing, even while editing class files*

Using Components with ActionScript 2.0

Currency Converter Application Overview

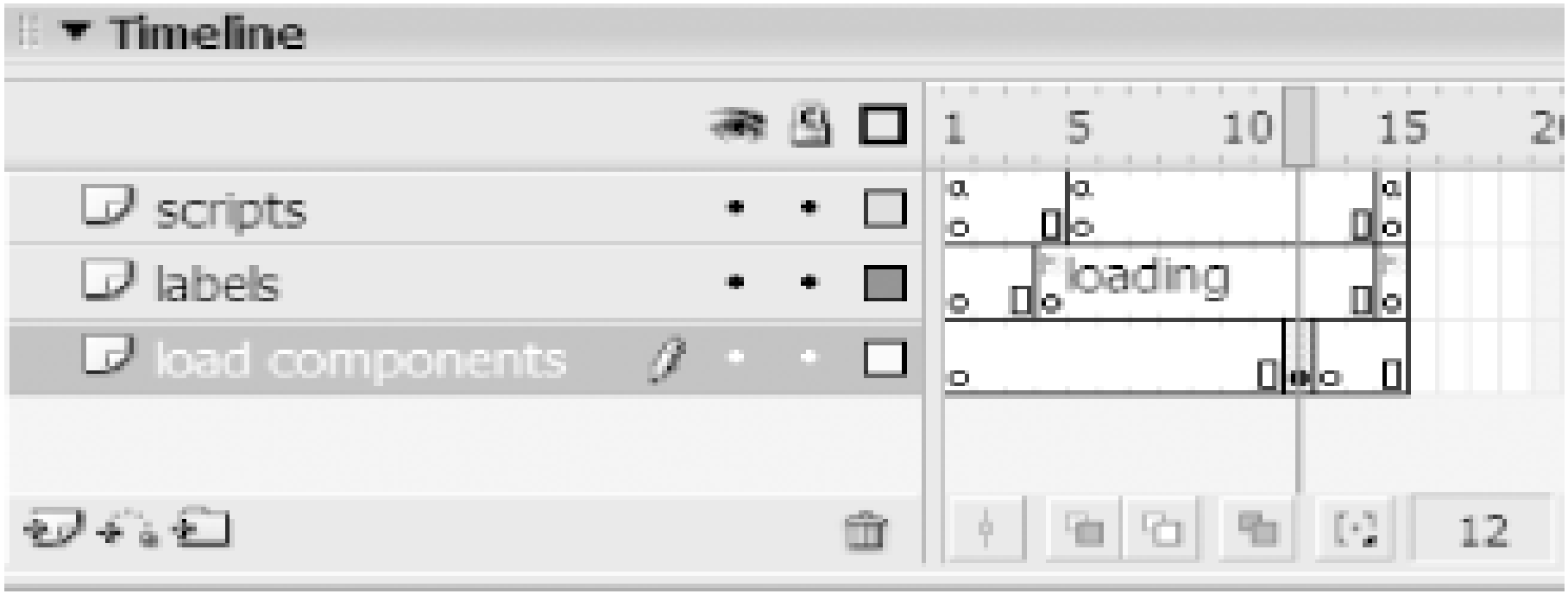
- Our example GUI application is a simple currency converter. The components used in our currency converter interface *(Button, ComboBox, Label, TextArea, and TextInput).*
- The user must specify an amount in Canadian dollars, select a *currency type from the drop-down list,* and click the Convert button to determine the equivalent amount in the selected currency. The result is displayed on screen.
- Our application's main Flash document is named *CurrencyConverter.fla.*
- It resides in CurrencyConverter/source.
- Our application's only class is *CurrencyConverter*.
- It is stored in an external .as class file named *CurrencyConverter.as*

The currency converter application

Preparing the Flash Document

- **Adding Components to the Document**
  - With CurrencyConverter.fla open in the Flash authoring tool, *select Window Development Panels > Components.*
  - In the Components panel, in the folder named *UI Components, click and drag the Button component to the Stage.* The Button component appears in the Library.
  - Select and delete the Button instance from the Stage.
- Select *Insert ->Timeline ->Layer.*
- Double-click the new layer and rename it ***load components***.
- Select *frame 12 in the load components layer*.
- Select *Insert ->Timeline ->Keyframe.*
- Select frame *13* in the load components layer.
- Select *Insert ->Timeline ->Keyframe.*

CurrencyConverter.fla's timeline and Stage



Starting the Application
The CurrencyConverter Class

- The main( ) call comes on frame 15 of the *scripts* layer in CurrencyConverter.fla's timeline, after our classes and components have been preloaded.

  import org.moock.tools.CurrencyConverter;
  CurrencyConverter.main(this, 0, 150, 100);

- The *CurrencyConverter* class's three main duties are to:
  - *Provide a method that starts the application (main( ))*
  - *Create the application interface*
  - *Respond to user input*

The currency converter application (Contd..)

```
// Import the package containing the Flash UI components we're
using.
import mx.controls.*;
// Define the CurrencyConverter class, and include the package
path.
class org.moock.tools.CurrencyConverter {
  // Hardcode the exchange rates for this example.
  private static var rateUS:Number = 1.3205;  // Rate for US
dollar
  private static var rateUK:Number = 2.1996;  // Rate for pound
sterling
  private static var rateEU:Number = 1.5600;  // Rate for euro
  // The container for all UI elements
  private var converter_mc:MovieClip;
  // The user interface components
  private var input:TextInput;                // Text field for
original amount
  private var currencyPicker:ComboBox;    // Currency selection
menu
  private var result:TextArea;                // Text field for
conversion output

  public function CurrencyConverter (target:MovieClip,
depth:Number,
                                     x:Number, y:Number) {
    buildConverter(target, depth, x, y);
  }
```

The currency converter application (contd..)

```
public function buildConverter (target:MovieClip, depth:Number,
                                x:Number, y:Number):Void {
    // Store a reference to the current object for use by nested
functions.
    var thisConverter:CurrencyConverter = this;
    // Make a container movie clip to hold the converter's UI.
    converter_mc = target.createEmptyMovieClip("converter",
depth);
    converter_mc._x = x;
    converter_mc._y = y;
    // Create the title.
    var title:Label = converter_mc.createClassObject(Label,
"title", 0);
    title.autoSize = "left";
    title.text = "Canadian Currency Converter";
    title.setStyle("color", 0x770000);
    title.setStyle("fontSize", 16);
```

The currency converter application (Contd..)

```
    // Create the instructions.
     var instructions:Label =
converter_mc.createClassObject(Label,

"instructions", 1);
     instructions.autoSize = "left";
     instructions.text = "Enter Amount in Canadian Dollars";
  instructions.move(instructions.x, title.y + title.height + 5);

     // Create an input text field to receive the amount to
convert.
  input = converter_mc.createClassObject(TextInput, "input", 2);
     input.setSize(200, 25);
     input.move(input.x, instructions.y + instructions.height);
     input.restrict = "0-9.";

     // Handle this component's enter event using a generic
listener object.
     var enterHandler:Object = new Object( );
     enterHandler.enter = function (e:Object):Void {
       thisConverter.convert( );
     }
     input.addEventListener("enter", enterHandler);
```

The currency converter application (Contd..)

```
// Create the currency selector ComboBox.
    currencyPicker = converter_mc.createClassObject(ComboBox,
"picker", 3);
    currencyPicker.setSize(200, currencyPicker.height);
currencyPicker.move(currencyPicker.x, input.y + input.height +
10);
    currencyPicker.dataProvider = [
              {label:"Select Target Currency", data:null},
              {label:"Canadian to U.S. Dollar", data:"US"},
              {label:"Canadian to UK Pound Sterling", data:"UK"},
              {label:"Canadian to EURO", data:"EU"}];

    // Create the Convert button.
    var convertButton:Button =
converter_mc.createClassObject(Button,"convertButton",4);
 convertButton.move(currencyPicker.x + currencyPicker.width + 5,
                        currencyPicker.y);
    convertButton.label = "Convert!";
    // Handle this component's events using a handler function.
    convertButton.clickHandler = function (e:Object):Void {
      thisConverter.convert( );
    };
```

The currency converter application (Contd..)

```
// Create the result output field.
 result = converter_mc.createClassObject(TextArea, "result", 5);
    result.setSize(200, 25);
    result.move(result.x, currencyPicker.y +
currencyPicker.height + 10);
    result.editable = false;
  }
public function convert ( ):Void {
    var convertedAmount:Number;
    var origAmount:Number = parseFloat(input.text);
    if (!isNaN(origAmount)) {
      if (currencyPicker.selectedItem.data != null) {
        switch (currencyPicker.selectedItem.data) {
        case "US":
        convertedAmount = origAmount / CurrencyConverter.rateUS;
            break;
        case "UK":
        convertedAmount = origAmount / CurrencyConverter.rateUK;
            break;
        case "EU":
        convertedAmount = origAmount / CurrencyConverter.rateEU;
            break;
        }
```

The currency converter application (Contd..)

```
result.text = "Result: " + convertedAmount;
        } else {
          result.text = "Please select a currency.";
        }
      } else {
        result.text = "Original amount is not valid.";
      }
    }
    // Program point of entry
    public static function main (target:MovieClip, depth:Number,
                                 x:Number, y:Number):Void {
      var converter:CurrencyConverter = new
CurrencyConverter(target, depth, x, y);
    }
}
```

The currency converter application (Contd..)

- Importing the Components' Package
- CurrencyConverter Properties
- The main( ) method
- The Class Constructor
- Creating the User Interface
- The interface container
- The title Label component
- The instructions Label component
- The input TextInput component

The currency converter application (Contd..)

- The currencyPicker ComboBox component
- The convertButton Button component
- The result TextArea component
- Converting Currency Based on User Input
- **Exporting the Final Application**
  - *With CurrencyConverter.fla open, choose File Publish Settings Formats.*
  - *Under the File heading, for Flash (.swf), enter **../deploy/CurrencyConverter.swf**.*
  - *Click OK.*
  - *To test our application in the Flash authoring tool's Test Movie mode, select Control Test Movie.*

Handling Component Events

- We handled component events in two different ways:
- With a generic listener object (in the case of the TextInput component):

```
var enterHandler:Object = new Object( );
enterHandler.enter = function (e:Object):Void {
  thisConverter.convert( );}
input.addEventListener("enter", enterHandler);
```

- With an event handler function (in the case of the Button component):

```
convertButton.clickHandler = function (e:Object):Void {
  thisConverter.convert( );}
```

Component-event-handling techniques

Technique	Example	Notes
Generic listener object	*var convertClickHandler:Object = new Object( );* *convertClickHandler.click = function (e:Object):Void { thisConverter.convert( );}* *convertButton.addEventListener("click", convertClickHandler);*	Generally, the preferred means of handling component events
Typed listener object	*convertButton.addEventListener("click", new ConvertButtonHandler(this));*	Same as generic but exposes event-handling code more explicitly
Event Proxy class	*convertButton.addEventListener("click", new EventProxy(this, "convert"));* *or* *convertButton.addEventListener("click", new EventProxy(this, convert));*	Functionally the same as generic listener object, but more convenient and easier to read

Component-event-handling techniques

Technique	Example	Notes
Listener function	convertButton.addEventListener("click", function (e:Object):Void { thisConverter.convert( ); });	The lesser evil of the two function-only event-handling mechanisms
Event handler function	convertButton.clickHandler = function (e:Object):Void { thisConverter.convert( ); }	The least-desirable means of handling component events; discouraged by Macromedia

MovieClip Subclasses

The Duality of MovieClip Subclasses

- A *movie clip* is a self-contained multimedia object with a timeline for changing state. Movie clips can contain graphics, video, and audio. They are perfect for creating the *audio/visual elements* of an application
- Every *MovieClip* subclass has two parts: *a movie clip symbol and a corresponding ActionScript 2.0 class*. A *MovieClip* subclass uses the ***extends*** keyword to inherit from *MovieClip*:

  class SomeClass extends MovieClip { }

- A *MovieClip* subclass must also be *represented physically in a Flash document Library* by a movie clip symbol.
- The movie clip symbol in the Library specifies the class that represents it, thus *coupling the symbol and MovieClip subclass together.*

Avatar: A MovieClip Subclass Example

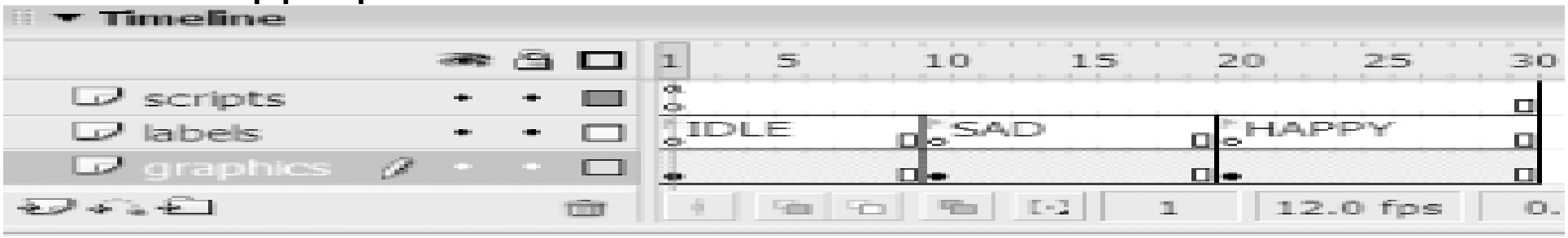
- To create instances of a *MovieClip* subclass, ***we do not use the new operator*** as we would with a typical class.
- Instances of the movie clip subclass's symbol are created either manually in the authoring tool or programmatically via **attachMovie( ) or duplicateMovieClip( ).**
- Our example *MovieClip* subclass, named *Avatar*, is an on-screen representation of a user in a chat room or a game.
- The *Avatar* class could quite appropriately be implemented using ***composition.***

The AvatarSymbol Movie Clip

- Every *MovieClip* subclass has a corresponding movie clip symbol

The AvatarSymbol Movie Clip

- Our *Avatar* class's movie clip symbol is called ***AvatarSymbol.***
- It contains a graphical depiction of the user in three different states: *Idle, Sad, and Happy.*
- The three states correspond to three labeled frames in *AvatarSymbol*'s timeline.
- To change the state of the avatar, we position *AvatarSymbol*'s at the appropriate frame in its *timeline.*



Creation of AvatarSymbol Movie Clip

1. Create a *new Flash document named AvatarDemo.fla*.
2. Select Insert ->New Symbol.
3. For the symbol Name, enter ***AvatarSymbol***. The name is arbitrary, but by convention, you should name the symbol *Class*Symbol where *Class* is the name of the class with which you expect to associate the symbol.
4. For the symbol's Behavior type, select *Movie Clip.*
5. Click OK. Flash automatically enters *Edit mode for the AvatarSymbol symbol.*

Steps to build *AvatarSymbol*'s timeline and contents:

1. Double-click *Layer 1* and rename it to *graphics*.
2. Select frame 30 of the graphics layer.

Steps to build *AvatarSymbol*'s timeline and contents:

3. Choose Insert ->Timeline ->Frame (F5).
4. Using Insert ->Timeline ->Layer, create *two new layers*, and name them *labels and scripts.*
5. With frame 1 of the *scripts* layer selected, enter the following code into the Actions panel (F9): ***stop( );***
6. Using Insert Timeline *Blank Keyframe (F7),* create blank keyframes on the *labels* and *graphics* layers at keyframes *10 and 20.*
7. With frame 1 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to ***IDLE***.
8. With frame 10 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **SAD**.
9. With frame 20 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **HAPPY**.
10. On the graphics layer, use Flash's drawing tools to *draw an idle face on frame 1, a sad face on frame 10, and a happy face on frame 20.*

- Notice that we add a stop( ) statement on frame 1 but not on frames 10 &20.

The Avatar Class

- To subclass the *MovieClip* class, we merely use the *extends keyword*, as we would for any other class. The basic code is:

  class Avatar extends MovieClip { }

- The *Avatar* class defines two instance methods:
- *init( ) : Initializes each Avatar instance*
- *setState( ) : Sets the avatar's state using one of three constants (class properties)—IDLE, SAD, and HAPPY.*
- The class has *no constructor function*. Instances of *MovieClip* subclasses are not created using the *standard constructor call syntax (using the new operator).*
- *init( )* method fulfills the traditional *role of the constructor* and should be called on each Avatar instance directly after it's created.

Avatar, a MovieClip subclass

```
class Avatar extends MovieClip {
public static var HAPPY:Number = 0;
public static var SAD:Number = 1;
public static var IDLE:Number = 2;
public function init ( ):Void {
  setState(Avatar.HAPPY);   }
public function setState(newState:Number):Void {
  switch (newState) {
    case Avatar.HAPPY:
      this.gotoAndStop("HAPPY");        break;
    case Avatar.SAD:
      this.gotoAndStop("SAD");          break;
    case Avatar.IDLE:
      this.gotoAndStop("IDLE");         break;
  }  }}
```

Linking AvatarSymbol to the Avatar Class

- To associate the *AvatarSymbol* movie clip with the *Avatar* class, we must set the *AvatarSymbol*'s so-called "AS 2.0 Class" in the Flash authoring tool, as follows:
  - Select the *AvatarSymbol* movie clip in AvatarDemo.fla's Library.
  - Select the pop-up Options menu in the top-right corner of the Library panel, and choose the *Linkage option.*
  - In the Linkage Properties dialog box, for Linkage, *select Export for ActionScript.*
  - In the Linkage Properties dialog box, ***for Identifier, enter AvatarSymbol.***
  - In the Linkage Properties dialog box, for AS 2.0 ***Class, enter Avatar.***
  - Click OK.

Creating Avatar Instances

var av:Avatar = Avatar(someMovieClip.attachMovie("AvatarSymbol", "avatar", 0));

- The instance of the library symbol (*AvatarSymbol*) is *physically added to the Stage* and also *reflected back* to ActionScript in the form of the object returned by *attachMovie( )*

Initializing Avatar Instances

- To *initialize our new Avatar instance, we invoke init( )* on it, as follows:

av.init( );

public static function createAvatar (name:String,
target:MovieClip, depth:Number):Avatar {
var av:Avatar = Avatar(target.attachMovie("AvatarSymbol", name, depth));
av.init( ); return av; }

***Avatar* class, showing the new *createAvatar( )* method and updated *init( )* method in context with the rest of the class:**

```
class Avatar extends MovieClip {
   public static var HAPPY:Number = 0;
   public static var SAD:Number = 1;
   public static var IDLE:Number = 2;
   public static function createAvatar
(name:String, target:MovieClip, depth:Number,
x:Number,y:Number):Avatar {
      var av:Avatar =
Avatar(target.attachMovie("AvatarSymbol", name,
depth));
      av.init(x, y);
      return av;   }
   public function init (x:Number, y:Number):Void
{
      setState(Avatar.HAPPY);
      this._x = x;
      this._y = y;   }
```

Avatar class, showing the new _createAvatar( )_ method and updated _init( )_ method in context with the rest of the class: (Contd..)

```
 public function
setState(newState:Number):Void {
    switch (newState) {
      case Avatar.HAPPY:
        this.gotoAndStop("HAPPY");
        break;
      case Avatar.SAD:
        this.gotoAndStop("SAD");
        break;
      case Avatar.IDLE:
        this.gotoAndStop("IDLE");
        break;     }   } }
```

Avatar: The Composition Version

- The composition version of the *Avatar* class is *cleaner and more flexible than its MovieClip subclass*
- When *Avatar* instances need to belong to the *MovieClip* datatype (e.g., for the sake of *polymorphism*)
- When the *Avatar* class and the *AvatarSymbol* movie clip *symbol need to be packaged together as a component*
- When *Avatar* instances need most of the methods of the *MovieClip* class *("Is-A" relationship is legitimate*)

```
class Avatar {
  public static var HAPPY:Number = 0;
  public static var SAD:Number = 1;
  public static var IDLE:Number = 2;
  private var av_mc:MovieClip;
```

```
public function Avatar (name:String, target:MovieClip, depth:Number,
    x:Number, y:Number) {
   av_mc = target.attachMovie("AvatarSymbol", name, depth);
   setState(Avatar.HAPPY);     setPosition(x, y);  }

 public function setState(newState:Number):Void {
   switch (newState) {
     case Avatar.HAPPY:
       av_mc.gotoAndStop("HAPPY");        break;
     case Avatar.SAD:
       av_mc.gotoAndStop("SAD");          break;
     case Avatar.IDLE:
       av_mc.gotoAndStop("IDLE");         break;    }  }

 public function setPosition (x:Number, y:Number):Void {
   av_mc._x = x;     av_mc._y = y;  }}
```

Issues with Nested Assets:: Properties and Methods of Nested Assets Initially Undefined

- When initializing a movie clip hierarchy in a movie, the *Flash Player starts with the outermost clip in the hierarchy*, defines its *custom properties and methods*, then moves on to its *child clips, defines their custom properties and methods*, and so on.
- A movie clip symbol, ***ChatRoomSymbol***, that represents a chat room. It contains the following component instances:
- outgoing :
  - *A TextInput component for outgoing messages*
- incoming :
  - *A TextArea component for incoming messages*
- userList :
  - *A List component for the list of users in the room*

ChatRoomSymbol instance via composition in a class, *ChatRoom*

```
class ChatRoom {
  private static var chatroomID:Number = 0;
  private var chat_mc:MovieClip;
  public function ChatRoom (target:MovieClip, depth:Number) {
    // Create clip instance.
    chat_mc = target.attachMovie("ChatRoomSymbol",
    "chatroom" + ChatRoom.chatroomID++,depth);
    chat_mc.userList.dataProvider =
                [{label:"Colin", data:"User1"},
                {label:"Derek", data:"User2"},
                {label:"James", data:"User3"}];  }}
```

Nested Assets Not Automatically Recognized by Compiler

- Assets nested in a movie clip symbol are not *automatically available to a corresponding MovieClip subclass* as instance properties.
- A *MovieClip* subclass cannot *refer to assets nested inside its associated movie clip symbol unless the subclass explicitly declares* those assets as instance properties.

```
class Login extends MovieClip {
public var userName:TextField; // Declare userName
public var password:TextField; // Declare password
public function Login ( ) {
userName.text = "Enter your name.";
password.text = "Enter your password."; } }
```

A Note on MovieClip Sub-subclasses

- Nothing officially prevents a *MovieClip* subclass from being *subclassed.*
- If we wanted to create an *Avatar* whose *mood changed randomly, we could create a RandomAvatar class* that inherits from *Avatar*.
- However, the *RandomAvatar* class would need to be *associated with its own library movie clip symbol*.
- The *RandomAvatar* class would *inherit the behavior of the Avatar class*, it would *not inherit Avatar's physical movie clip symbol.*
- To create the *RandomAvatar* class, we'd have to create a *completely separate symbol, RandomAvatarSymbol*, that is a *duplicate of AvatarSymbol*.

A Note on MovieClip Sub-subclasses (Contd..)

```
class RandomAvatar extends Avatar {
  private var randomInt:Number;
  public function RandomAvatar (name:String, target:MovieClip,
    depth:Number, x:Number, y:Number) {
    super(name, target, depth, x, y);
    startRandom( );  }

  public function startRandom ( ):Void {
    randomInt = setInterval (function (av:RandomAvatar):Void {
      var r:Number = Math.floor(Math.random( ) * 3);
      av.setState(r);     } , 500, this); }
  public function stopRandom ( ):Void {
    clearInterval(randomInt);  }}
```

Content in AvatarDemo.fla file

- *var av:Avatar = new Avatar("avatar", this, 0, 300, 200);*
- *var rav:RandomAvatar = new RandomAvatar("randomavatar", this, 1, 100, 100);*

Unit V

An OOP Application Framework
Using Components with ActionScript 2.0
MovieClip Subclasses

The Basic Directory Structure

- Create a ***directory named AppName*** on your hard drive. The AppName directory will contain *everything in our project*, including source code and final output
- In the AppName directory, ***create a subdirectory named deploy***. The deploy directory will contain the ***final, compiled application***, ready for posting to a web site or other distribution medium.
- In the AppName directory, create a ***subdirectory named source***. The source directory will contain all ***source code for the application***, including classes *(.as files)* and Flash documents *(.fla files).*

AppName/

deploy/

source/

The Flash Document (.fla file)

- Every Flash application must include at least one Flash document ***(.fla file)***.
- The *Flash document is the source file* from which a Flash movie ***(.swf file)*** is exported.
- The *Flash movie* is what's actually displayed in the *Flash Player* (i.e., the Flash runtime environment).
- In the Flash authoring tool, choose *File ->New.*
- In the *New Document dialog box*, on the General tab, for Type, choose *Flash Document*, then click OK.
- Use *File ->Save As* to save the Flash document as *AppName.fla* in the AppName/source directory.

The Classes

- In Flash, most applications are *visual and include graphical user interfaces*. Hence, the classes of an OOP Flash application typically ***create and manage user interface components.***
- An OOP application can have ***any number of classes***, but only one of them is used to ***start the application in motion***.
- We'll store our classes in the ***package com.somedomain.***
- To create the directories for the ***com.somedomain package***, follow these steps:
  - In the AppName/source directory, *create a subdirectory named com.*
  - In the AppName/source/com directory, *create a subdirectory named somedomain.*

The Classes (Contd..)

- *To create class A*, follow these steps if you're using Flash MX Professional 2004:
- In Flash MX Professional 2004, *choose File ->New.*
- In the *New Document dialog box*, on the General tab, for Type, *choose ActionScript File.*
- Click *OK*. The script editor *launches with an empty file*.
- Enter the following code into the script editor:

```
import com.somedomain.B;
class com.somedomain.A {
  private static var bInstance:B;
  public function A ( ) {
    // In this example, class A's constructor is not used.
  }
```

The Classes (Contd..)

public static function main ( ):Void {

trace("Starting application.");

bInstance = new B( ); } }

Choose File -> Save As.

- In the Save As dialog box, specify **A.as, B.as** as the filename, and save the file in the directory:

class com.somedomain.B {

public function B ( ) {

trace("An instance of class B was constructed."); } }

AppName/source/com/somedomain.

- As long as the directory in which they reside is added to the *global or document classpath*, the classes will be *accessible to timeline code and other ActionScript classes.*

The Document Timeline

- The *fundamental metaphor* of a Flash document is the *timeline*, which can be used to ***create animations like a filmstrip***.
- When used for animation, ***the frames in the timeline are displayed in rapid linear succession*** by the Flash Player.
- The timeline can also be used to *create a series of application states*, in which *specific frames correspond to specific states* and frames are *displayed according to the application's logic*
- To load our *A* and *B* classes, we'll follow these steps:
  - *Specify the export frame for classes in the movie.*
  - *Add the labeled state frames loading and main to AppName.fla's timeline.*
  - *Add code that displays a loading message while the movie loads.*

To specify the export frame for classes in the movie, follow these steps:

- Open AppName.fla in the Flash authoring tool.
- Choose File ->Publish Settings.
- In the Publish Settings dialog box, on the Flash tab, next to the ActionScript Version, click Settings.
- In the ActionScript Settings dialog box, for the Export Frame for Classes option, enter **10**.
- Click OK to confirm the ActionScript settings.
- Click OK to confirm the publish settings.

To add the labeled state frames loading and main to AppName.fla's timeline, follow these steps:

- In AppName.fla's main timeline, double-click *Layer 1* and rename it to **scripts**. We'll place all our code on the *scripts* layer.

The Document Timeline (Contd..)

- In the main timeline of AppName.fla, select frame 15 of the *scripts* layer.
- Choose Insert ->Timeline ->Keyframe (F6).
- Choose Insert ->Timeline ->Layer.
- Double-click the new layer's name and change it to **labels**.
- At frames 4 and 15 of the *labels* layer, add a new keyframe (Insert Timeline Keyframe or F6). Just as the *scripts* layer holds all our scripts, the *labels* layer is used exclusively to hold frame labels.
- With frame 4 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **loading**.
- With frame 15 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **main**.

steps to add code that displays a loading message while the movie loads:

- At frame 5 of the *scripts* layer, add a new keyframe (Insert ->Timeline ->Keyframe or F6).
- With frame 5 of the *scripts* layer selected, enter the following code into the Actions panel (F9):

```
if (_framesloaded == _totalframes) {
  gotoAndStop("main");
} else {   gotoAndPlay("loading");  }
```

- With frame 1 of the *scripts* layer selected, enter the following code into the Actions panel:

```
this.createTextField("loadmsg_txt", 0, 200, 200, 0, 0);
loadmsg_txt.autoSize = true;
loadmsg_txt.text = "Loading...Please wait.";
```

The Document Timeline (Contd..)

- With frame 15 of the *scripts* layer selected, enter the following code into the Actions panel:

  loadmsg_txt.removeTextField( );

- We've now provided the basic timeline structure that loads our application's classes.
- All that's left is to start the application by invoking *A.main( )*.
- We do that on the frame labeled *main* in AppName.fla.
- Add the following code to the end of the script on frame 15 (i.e., just below loadmsg_txt.removeTextField( );):

  import com.somedomain.A;

  A.main( );

- To test, we need to export a .swf file and run it in the Flash Player

The Exported Flash Movie (.swf file)

- Our application is now ready for testing and—assuming all goes well—deployment.
- To specify the directory in which to create AppName.swf, follow these steps:
  - *With AppName.fla open, choose File ->Publish Settings-> Formats.*
  - *Under the File heading, for Flash (.swf), enter **../deploy/AppName.swf**.*
  - *Click OK.*
- To test our application in the Flash authoring tool's Test Movie mode, select Control Test Movie.
- Testing a movie actually creates AppName.swf in the *AppName*/deploy directory and immediately loads it into a debugging version of the Flash Player.

publish an HTML page that includes the movie, ready for posting to a web site as follows:

With AppName.fla open, choose File Publish Settings Formats.

*Under the File heading, for HTML (.html), enter **../deploy/AppName.html**.*

Click Publish. Click OK.

- Test locally by opening *AppName.html* in your *web browser.*
- When the local testing proves successful, upload the *.html and .swf files to your web server.*
- *Packages and classpaths matter only at compile time*.
- We you can upload the .html and .swf wherever you like on your server, but in our example*, the two files must reside in the same web server folder.*

Projects in Flash MX Professional 2004

- To help manage the files in a large application, Flash MX Professional 2004 supports the concept of projects.
- A project is a group of related files that can be managed via the Project panel in the Flash MX Professional 2004 authoring tool.
- The Project panel resembles a file explorer and offers the following features:
  - *Authoring tool integration with source control applications such as Microsoft Visual SourceSafe*
  - *Easy access to related source files*
  - *One-click application publishing, even while editing class files*

Using Components with ActionScript 2.0

Currency Converter Application Overview

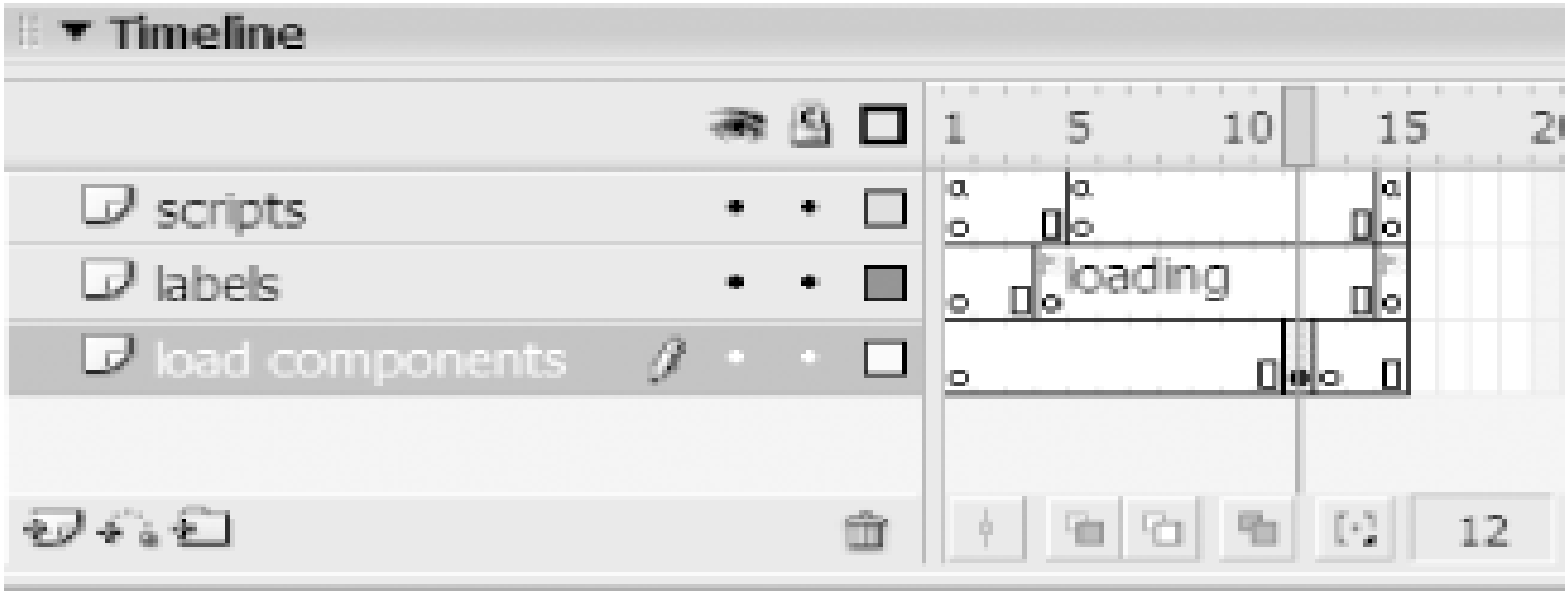
- Our example GUI application is a simple currency converter. The components used in our currency converter interface *(Button, ComboBox, Label, TextArea, and TextInput).*
- The user must specify an amount in Canadian dollars, select a *currency type from the drop-down list,* and click the Convert button to determine the equivalent amount in the selected currency. The result is displayed on screen.
- Our application's main Flash document is named *CurrencyConverter.fla.*
- It resides in CurrencyConverter/source.
- Our application's only class is *CurrencyConverter*.
- It is stored in an external .as class file named *CurrencyConverter.as*

The currency converter application

Preparing the Flash Document

- **Adding Components to the Document**
  - With CurrencyConverter.fla open in the Flash authoring tool, *select Window Development Panels > Components.*
  - In the Components panel, in the folder named *UI Components, click and drag the Button component to the Stage.* The Button component appears in the Library.
  - Select and delete the Button instance from the Stage.
- Select *Insert ->Timeline ->Layer.*
- Double-click the new layer and rename it ***load components***.
- Select *frame 12 in the load components layer*.
- Select *Insert ->Timeline ->Keyframe.*
- Select frame *13* in the load components layer.
- Select *Insert ->Timeline ->Keyframe.*

CurrencyConverter.fla's timeline and Stage



Starting the Application
The CurrencyConverter Class

- The main( ) call comes on frame 15 of the *scripts* layer in CurrencyConverter.fla's timeline, after our classes and components have been preloaded.

  import org.moock.tools.CurrencyConverter;
  CurrencyConverter.main(this, 0, 150, 100);

- The *CurrencyConverter* class's three main duties are to:
  - *Provide a method that starts the application (main( ))*
  - *Create the application interface*
  - *Respond to user input*

The currency converter application (Contd..)

```
// Import the package containing the Flash UI components we're
using.
import mx.controls.*;
// Define the CurrencyConverter class, and include the package
path.
class org.moock.tools.CurrencyConverter {
  // Hardcode the exchange rates for this example.
  private static var rateUS:Number = 1.3205;  // Rate for US
dollar
  private static var rateUK:Number = 2.1996;  // Rate for pound
sterling
  private static var rateEU:Number = 1.5600;  // Rate for euro
  // The container for all UI elements
  private var converter_mc:MovieClip;
  // The user interface components
  private var input:TextInput;                // Text field for
original amount
  private var currencyPicker:ComboBox;    // Currency selection
menu
  private var result:TextArea;                // Text field for
conversion output

  public function CurrencyConverter (target:MovieClip,
depth:Number,
                                     x:Number, y:Number) {
    buildConverter(target, depth, x, y);
  }
```

The currency converter application (contd..)

```
public function buildConverter (target:MovieClip, depth:Number,
                                x:Number, y:Number):Void {
    // Store a reference to the current object for use by nested
functions.
    var thisConverter:CurrencyConverter = this;
    // Make a container movie clip to hold the converter's UI.
    converter_mc = target.createEmptyMovieClip("converter",
depth);
    converter_mc._x = x;
    converter_mc._y = y;
    // Create the title.
    var title:Label = converter_mc.createClassObject(Label,
"title", 0);
    title.autoSize = "left";
    title.text = "Canadian Currency Converter";
    title.setStyle("color", 0x770000);
    title.setStyle("fontSize", 16);
```

The currency converter application (Contd..)

```
    // Create the instructions.
     var instructions:Label =
converter_mc.createClassObject(Label,

"instructions", 1);
     instructions.autoSize = "left";
     instructions.text = "Enter Amount in Canadian Dollars";
  instructions.move(instructions.x, title.y + title.height + 5);

     // Create an input text field to receive the amount to
convert.
  input = converter_mc.createClassObject(TextInput, "input", 2);
     input.setSize(200, 25);
     input.move(input.x, instructions.y + instructions.height);
     input.restrict = "0-9.";

     // Handle this component's enter event using a generic
listener object.
     var enterHandler:Object = new Object( );
     enterHandler.enter = function (e:Object):Void {
       thisConverter.convert( );
     }
     input.addEventListener("enter", enterHandler);
```

The currency converter application (Contd..)

```
// Create the currency selector ComboBox.
    currencyPicker = converter_mc.createClassObject(ComboBox,
"picker", 3);
    currencyPicker.setSize(200, currencyPicker.height);
currencyPicker.move(currencyPicker.x, input.y + input.height +
10);
    currencyPicker.dataProvider = [
              {label:"Select Target Currency", data:null},
              {label:"Canadian to U.S. Dollar", data:"US"},
              {label:"Canadian to UK Pound Sterling", data:"UK"},
              {label:"Canadian to EURO", data:"EU"}];

    // Create the Convert button.
    var convertButton:Button =
converter_mc.createClassObject(Button,"convertButton",4);
 convertButton.move(currencyPicker.x + currencyPicker.width + 5,
                        currencyPicker.y);
    convertButton.label = "Convert!";
    // Handle this component's events using a handler function.
    convertButton.clickHandler = function (e:Object):Void {
      thisConverter.convert( );
    };
```

The currency converter application (Contd..)

```
// Create the result output field.
 result = converter_mc.createClassObject(TextArea, "result", 5);
    result.setSize(200, 25);
    result.move(result.x, currencyPicker.y +
currencyPicker.height + 10);
    result.editable = false;
  }
public function convert ( ):Void {
    var convertedAmount:Number;
    var origAmount:Number = parseFloat(input.text);
    if (!isNaN(origAmount)) {
      if (currencyPicker.selectedItem.data != null) {
        switch (currencyPicker.selectedItem.data) {
        case "US":
        convertedAmount = origAmount / CurrencyConverter.rateUS;
            break;
        case "UK":
        convertedAmount = origAmount / CurrencyConverter.rateUK;
            break;
        case "EU":
        convertedAmount = origAmount / CurrencyConverter.rateEU;
            break;
        }
```

The currency converter application (Contd..)

```
result.text = "Result: " + convertedAmount;
        } else {
          result.text = "Please select a currency.";
        }
      } else {
        result.text = "Original amount is not valid.";
      }
    }
    // Program point of entry
    public static function main (target:MovieClip, depth:Number,
                                 x:Number, y:Number):Void {
      var converter:CurrencyConverter = new
CurrencyConverter(target, depth, x, y);
    }
}
```

The currency converter application (Contd..)

- Importing the Components' Package
- CurrencyConverter Properties
- The main( ) method
- The Class Constructor
- Creating the User Interface
- The interface container
- The title Label component
- The instructions Label component
- The input TextInput component

The currency converter application (Contd..)

- The currencyPicker ComboBox component
- The convertButton Button component
- The result TextArea component
- Converting Currency Based on User Input
- **Exporting the Final Application**
  - *With CurrencyConverter.fla open, choose File Publish Settings Formats.*
  - *Under the File heading, for Flash (.swf), enter **../deploy/CurrencyConverter.swf**.*
  - *Click OK.*
  - *To test our application in the Flash authoring tool's Test Movie mode, select Control Test Movie.*

Handling Component Events

- We handled component events in two different ways:
- With a generic listener object (in the case of the TextInput component):

```
var enterHandler:Object = new Object( );
enterHandler.enter = function (e:Object):Void {
  thisConverter.convert( );}
input.addEventListener("enter", enterHandler);
```

- With an event handler function (in the case of the Button component):

```
convertButton.clickHandler = function (e:Object):Void {
  thisConverter.convert( );}
```

Component-event-handling techniques

Technique	Example	Notes
Generic listener object	*var convertClickHandler:Object = new Object( );* *convertClickHandler.click = function (e:Object):Void { thisConverter.convert( );}* *convertButton.addEventListener("click", convertClickHandler);*	Generally, the preferred means of handling component events
Typed listener object	*convertButton.addEventListener("click", new ConvertButtonHandler(this));*	Same as generic but exposes event-handling code more explicitly
Event Proxy class	*convertButton.addEventListener("click", new EventProxy(this, "convert"));* *or* *convertButton.addEventListener("click", new EventProxy(this, convert));*	Functionally the same as generic listener object, but more convenient and easier to read

Component-event-handling techniques

Technique	Example	Notes
Listener function	convertButton.addEventListener("click", function (e:Object):Void { thisConverter.convert( ); });	The lesser evil of the two function-only event-handling mechanisms
Event handler function	convertButton.clickHandler = function (e:Object):Void { thisConverter.convert( ); }	The least-desirable means of handling component events; discouraged by Macromedia

MovieClip Subclasses

The Duality of MovieClip Subclasses

- A *movie clip* is a self-contained multimedia object with a timeline for changing state. Movie clips can contain graphics, video, and audio. They are perfect for creating the *audio/visual elements* of an application
- Every *MovieClip* subclass has two parts: *a movie clip symbol and a corresponding ActionScript 2.0 class*. A *MovieClip* subclass uses the ***extends*** keyword to inherit from *MovieClip*:

  class SomeClass extends MovieClip { }

- A *MovieClip* subclass must also be *represented physically in a Flash document Library* by a movie clip symbol.
- The movie clip symbol in the Library specifies the class that represents it, thus *coupling the symbol and MovieClip subclass together.*

Avatar: A MovieClip Subclass Example

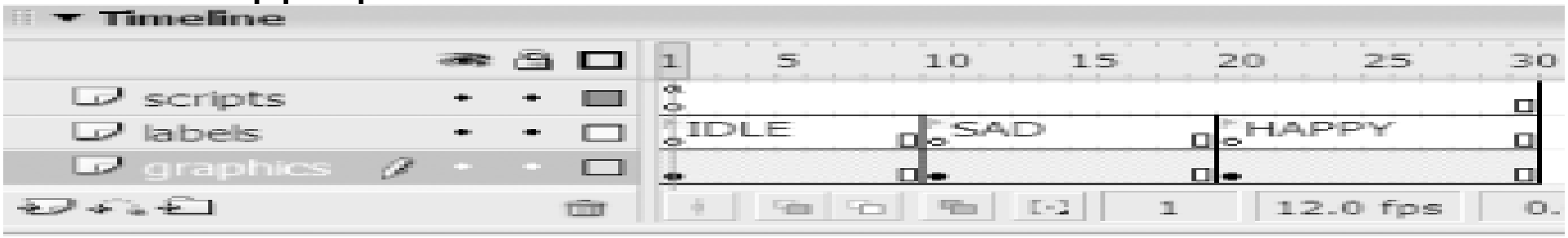
- To create instances of a *MovieClip* subclass, ***we do not use the new operator*** as we would with a typical class.
- Instances of the movie clip subclass's symbol are created either manually in the authoring tool or programmatically via **attachMovie( ) or duplicateMovieClip( ).**
- Our example *MovieClip* subclass, named *Avatar*, is an on-screen representation of a user in a chat room or a game.
- The *Avatar* class could quite appropriately be implemented using ***composition.***

The AvatarSymbol Movie Clip

- Every *MovieClip* subclass has a corresponding movie clip symbol

The AvatarSymbol Movie Clip

- Our *Avatar* class's movie clip symbol is called ***AvatarSymbol.***
- It contains a graphical depiction of the user in three different states: *Idle, Sad, and Happy.*
- The three states correspond to three labeled frames in *AvatarSymbol*'s timeline.
- To change the state of the avatar, we position *AvatarSymbol*'s at the appropriate frame in its *timeline.*



Creation of AvatarSymbol Movie Clip

1. Create a *new Flash document named AvatarDemo.fla*.
2. Select Insert ->New Symbol.
3. For the symbol Name, enter ***AvatarSymbol***. The name is arbitrary, but by convention, you should name the symbol *Class*Symbol where *Class* is the name of the class with which you expect to associate the symbol.
4. For the symbol's Behavior type, select *Movie Clip.*
5. Click OK. Flash automatically enters *Edit mode for the AvatarSymbol symbol.*

Steps to build *AvatarSymbol*'s timeline and contents:

1. Double-click *Layer 1* and rename it to *graphics*.
2. Select frame 30 of the graphics layer.

Steps to build *AvatarSymbol*'s timeline and contents:

3. Choose Insert ->Timeline ->Frame (F5).
4. Using Insert ->Timeline ->Layer, create *two new layers*, and name them *labels and scripts.*
5. With frame 1 of the *scripts* layer selected, enter the following code into the Actions panel (F9): ***stop( );***
6. Using Insert Timeline *Blank Keyframe (F7),* create blank keyframes on the *labels* and *graphics* layers at keyframes *10 and 20.*
7. With frame 1 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to ***IDLE***.
8. With frame 10 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **SAD**.
9. With frame 20 of the *labels* layer selected, in the Properties panel, under Frame, change <Frame Label> to **HAPPY**.
10. On the graphics layer, use Flash's drawing tools to *draw an idle face on frame 1, a sad face on frame 10, and a happy face on frame 20.*

- Notice that we add a stop( ) statement on frame 1 but not on frames 10 &20.

The Avatar Class

- To subclass the *MovieClip* class, we merely use the *extends keyword*, as we would for any other class. The basic code is:

  class Avatar extends MovieClip { }

- The *Avatar* class defines two instance methods:
- *init( ) : Initializes each Avatar instance*
- *setState( ) : Sets the avatar's state using one of three constants (class properties)—IDLE, SAD, and HAPPY.*
- The class has *no constructor function*. Instances of *MovieClip* subclasses are not created using the *standard constructor call syntax (using the new operator).*
- *init( )* method fulfills the traditional *role of the constructor* and should be called on each Avatar instance directly after it's created.

Avatar, a MovieClip subclass

```
class Avatar extends MovieClip {
public static var HAPPY:Number = 0;
public static var SAD:Number = 1;
public static var IDLE:Number = 2;
public function init ( ):Void {
  setState(Avatar.HAPPY);   }
public function setState(newState:Number):Void {
  switch (newState) {
    case Avatar.HAPPY:
      this.gotoAndStop("HAPPY");        break;
    case Avatar.SAD:
      this.gotoAndStop("SAD");          break;
    case Avatar.IDLE:
      this.gotoAndStop("IDLE");         break;
  }  }}
```

Linking AvatarSymbol to the Avatar Class

- To associate the *AvatarSymbol* movie clip with the *Avatar* class, we must set the *AvatarSymbol*'s so-called "AS 2.0 Class" in the Flash authoring tool, as follows:
  - Select the *AvatarSymbol* movie clip in AvatarDemo.fla's Library.
  - Select the pop-up Options menu in the top-right corner of the Library panel, and choose the *Linkage option.*
  - In the Linkage Properties dialog box, for Linkage, *select Export for ActionScript.*
  - In the Linkage Properties dialog box, ***for Identifier, enter AvatarSymbol.***
  - In the Linkage Properties dialog box, for AS 2.0 ***Class, enter Avatar.***
  - Click OK.

Creating Avatar Instances

var av:Avatar = Avatar(someMovieClip.attachMovie("AvatarSymbol", "avatar", 0));

- The instance of the library symbol (*AvatarSymbol*) is *physically added to the Stage* and also *reflected back* to ActionScript in the form of the object returned by *attachMovie( )*

Initializing Avatar Instances

- To *initialize our new Avatar instance, we invoke init( )* on it, as follows:

av.init( );

public static function createAvatar (name:String,
target:MovieClip, depth:Number):Avatar {
var av:Avatar = Avatar(target.attachMovie("AvatarSymbol", name, depth));
av.init( ); return av; }

***Avatar* class, showing the new *createAvatar( )* method and updated *init( )* method in context with the rest of the class:**

```
class Avatar extends MovieClip {
   public static var HAPPY:Number = 0;
   public static var SAD:Number = 1;
   public static var IDLE:Number = 2;
   public static function createAvatar
(name:String, target:MovieClip, depth:Number,
x:Number,y:Number):Avatar {
      var av:Avatar =
Avatar(target.attachMovie("AvatarSymbol", name,
depth));
      av.init(x, y);
      return av;   }
   public function init (x:Number, y:Number):Void
{
      setState(Avatar.HAPPY);
      this._x = x;
      this._y = y;   }
```

Avatar class, showing the new _createAvatar( )_ method and updated _init( )_ method in context with the rest of the class: (Contd..)

```
 public function
setState(newState:Number):Void {
    switch (newState) {
      case Avatar.HAPPY:
        this.gotoAndStop("HAPPY");
        break;
      case Avatar.SAD:
        this.gotoAndStop("SAD");
        break;
      case Avatar.IDLE:
        this.gotoAndStop("IDLE");
        break;     }   } }
```

Avatar: The Composition Version

- The composition version of the *Avatar* class is *cleaner and more flexible than its MovieClip subclass*
- When *Avatar* instances need to belong to the *MovieClip* datatype (e.g., for the sake of *polymorphism*)
- When the *Avatar* class and the *AvatarSymbol* movie clip *symbol need to be packaged together as a component*
- When *Avatar* instances need most of the methods of the *MovieClip* class *("Is-A" relationship is legitimate*)

```
class Avatar {
  public static var HAPPY:Number = 0;
  public static var SAD:Number = 1;
  public static var IDLE:Number = 2;
  private var av_mc:MovieClip;
```

```
public function Avatar (name:String, target:MovieClip, depth:Number,
    x:Number, y:Number) {
   av_mc = target.attachMovie("AvatarSymbol", name, depth);
   setState(Avatar.HAPPY);     setPosition(x, y);  }

 public function setState(newState:Number):Void {
   switch (newState) {
     case Avatar.HAPPY:
       av_mc.gotoAndStop("HAPPY");        break;
     case Avatar.SAD:
       av_mc.gotoAndStop("SAD");          break;
     case Avatar.IDLE:
       av_mc.gotoAndStop("IDLE");         break;    }  }

 public function setPosition (x:Number, y:Number):Void {
   av_mc._x = x;     av_mc._y = y;  }}
```

Issues with Nested Assets:: Properties and Methods of Nested Assets Initially Undefined

- When initializing a movie clip hierarchy in a movie, the *Flash Player starts with the outermost clip in the hierarchy*, defines its *custom properties and methods*, then moves on to its *child clips, defines their custom properties and methods*, and so on.
- A movie clip symbol, ***ChatRoomSymbol***, that represents a chat room. It contains the following component instances:
- outgoing :
  - *A TextInput component for outgoing messages*
- incoming :
  - *A TextArea component for incoming messages*
- userList :
  - *A List component for the list of users in the room*

ChatRoomSymbol instance via composition in a class, *ChatRoom*

```
class ChatRoom {
  private static var chatroomID:Number = 0;
  private var chat_mc:MovieClip;
  public function ChatRoom (target:MovieClip, depth:Number) {
    // Create clip instance.
    chat_mc = target.attachMovie("ChatRoomSymbol",
    "chatroom" + ChatRoom.chatroomID++,depth);
    chat_mc.userList.dataProvider =
                [{label:"Colin", data:"User1"},
                {label:"Derek", data:"User2"},
                {label:"James", data:"User3"}];  }}
```

Nested Assets Not Automatically Recognized by Compiler

- Assets nested in a movie clip symbol are not *automatically available to a corresponding MovieClip subclass* as instance properties.
- A *MovieClip* subclass cannot *refer to assets nested inside its associated movie clip symbol unless the subclass explicitly declares* those assets as instance properties.

```
class Login extends MovieClip {
public var userName:TextField; // Declare userName
public var password:TextField; // Declare password
public function Login ( ) {
userName.text = "Enter your name.";
password.text = "Enter your password."; } }
```

A Note on MovieClip Sub-subclasses

- Nothing officially prevents a *MovieClip* subclass from being *subclassed.*
- If we wanted to create an *Avatar* whose *mood changed randomly, we could create a RandomAvatar class* that inherits from *Avatar*.
- However, the *RandomAvatar* class would need to be *associated with its own library movie clip symbol*.
- The *RandomAvatar* class would *inherit the behavior of the Avatar class*, it would *not inherit Avatar's physical movie clip symbol.*
- To create the *RandomAvatar* class, we'd have to create a *completely separate symbol, RandomAvatarSymbol*, that is a *duplicate of AvatarSymbol*.

A Note on MovieClip Sub-subclasses (Contd..)

```
class RandomAvatar extends Avatar {
  private var randomInt:Number;
  public function RandomAvatar (name:String, target:MovieClip,
    depth:Number, x:Number, y:Number) {
    super(name, target, depth, x, y);
    startRandom( );  }

  public function startRandom ( ):Void {
    randomInt = setInterval (function (av:RandomAvatar):Void {
      var r:Number = Math.floor(Math.random( ) * 3);
      av.setState(r);     } , 500, this); }
  public function stopRandom ( ):Void {
    clearInterval(randomInt);  }}
```

Content in AvatarDemo.fla file

- *var av:Avatar = new Avatar("avatar", this, 0, 300, 200);*
- *var rav:RandomAvatar = new RandomAvatar("randomavatar", this, 1, 100, 100);*

Chapter 7

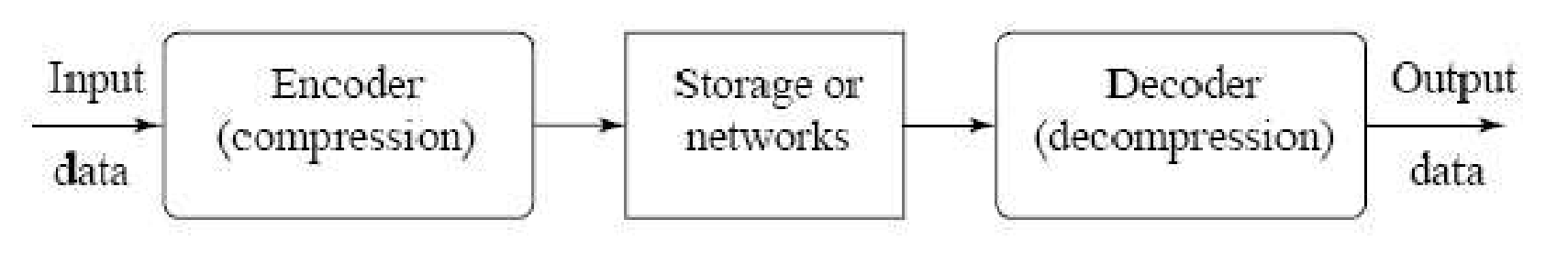
Lossless Compression Algorithms

7.1 Introduction
7.2 Basics of Information Theory
7.3 Run-Length Coding
7.4 Variable-Length Coding (VLC)
7.5 Dictionary-based Coding
7.6 Arithmetic Coding
7.7 Lossless Image Compression

7.1 Introduction

- **Compression:** the process of coding that will effectively reduce the total number of bits needed to represent certain information.

A General Data Compression Scheme.



- If the compression and decompression processes induce no information loss, then the compression scheme is **lossless**;
- Otherwise, it is **lossy**.

Compression ratio:

$$\textit{compression ratio} = B0 \,/\, B1$$

$B0$: number of bits before compression
$B1$: number of bits after compression

7.2 Basics of Information Theory

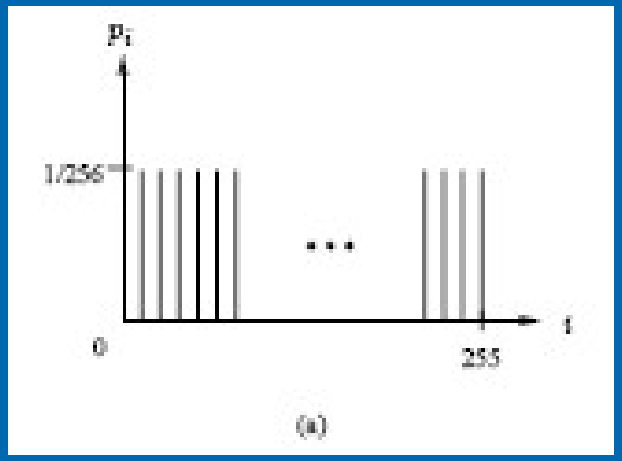
The *entropy* η *(eta)* of an information *source* with alphabet $S = \{s_1, s_2, \ldots s_n\}$ is:

$$\eta = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

p_i : probability that symbol s_i will occur in S.

- It can be interpreted as the average shortest message length, in bits, that can be sent to communicate the true value of the random variable to a recipient.
- This represents a fundamental mathematical limit on the best possible lossless data compression of any communication.

Distribution of Gray-Level Intensities



Histograms for a Gray-level Image.

Figure shows the histogram of an image with *uniform* distribution of gray-level intensities,

i.e., *For all i,* $p_i = 1/256$.

Hence, the entropy of this image is: $\log_2 256 = 8$

Entropy and Code Length

The entropy η is a weighted-sum of terms $\log_2 1/p_i$;

Hence it represents the *average* amount of information contained per symbol in the source *S*.

The entropy η specifies the lower bound for the average number of bits to code each symbol in *S*,

i.e., $\eta \leq \acute{\imath}$

$\acute{\imath}$ - the average length (measured in bits) of the code-words produced by the encoder.

Entropy and Code Length

- Alphabet={a, b, c, d} with probability {4/8, 2/8, 1/8, 1/8}
- $\eta = 4/8*\log_2 2 + 2/8*\log_2 4 + 1/8*\log_2 8 + 1/8*\log_2 8$
- $\eta = 1/2 + 1/2 + 3/8 + 3/8 = 1.75$ average length
- a => 0 b => 10 c => 110 d => 111
- Message: {abcdabaa} => {0 10 110 111 0 10 0 0}
- 14 bits / 8 chars = 1.75 average length

7.3 Run-Length Coding

Run-Length Coding

- **Rationale for RLC:** if the information source has the property that symbols tend to form continuous groups, then such symbol and the length of the group can be coded.
- **Memoryless Source:** Namely, the value of the current symbol does not depend on the values of the previously appeared symbols.
- Instead of assuming memoryless source, *Run-Length Coding (RLC)* exploits memory present in the information source.

- **Run-length encoding (RLE) is a very simple form of** data compression in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run.

WWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWW

If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:

6WB12W3B14W

7.4 Variable-Length Coding (VLC)

Variable-Length Coding (VLC)

Shannon-Fano Algorithm - a top-down approach

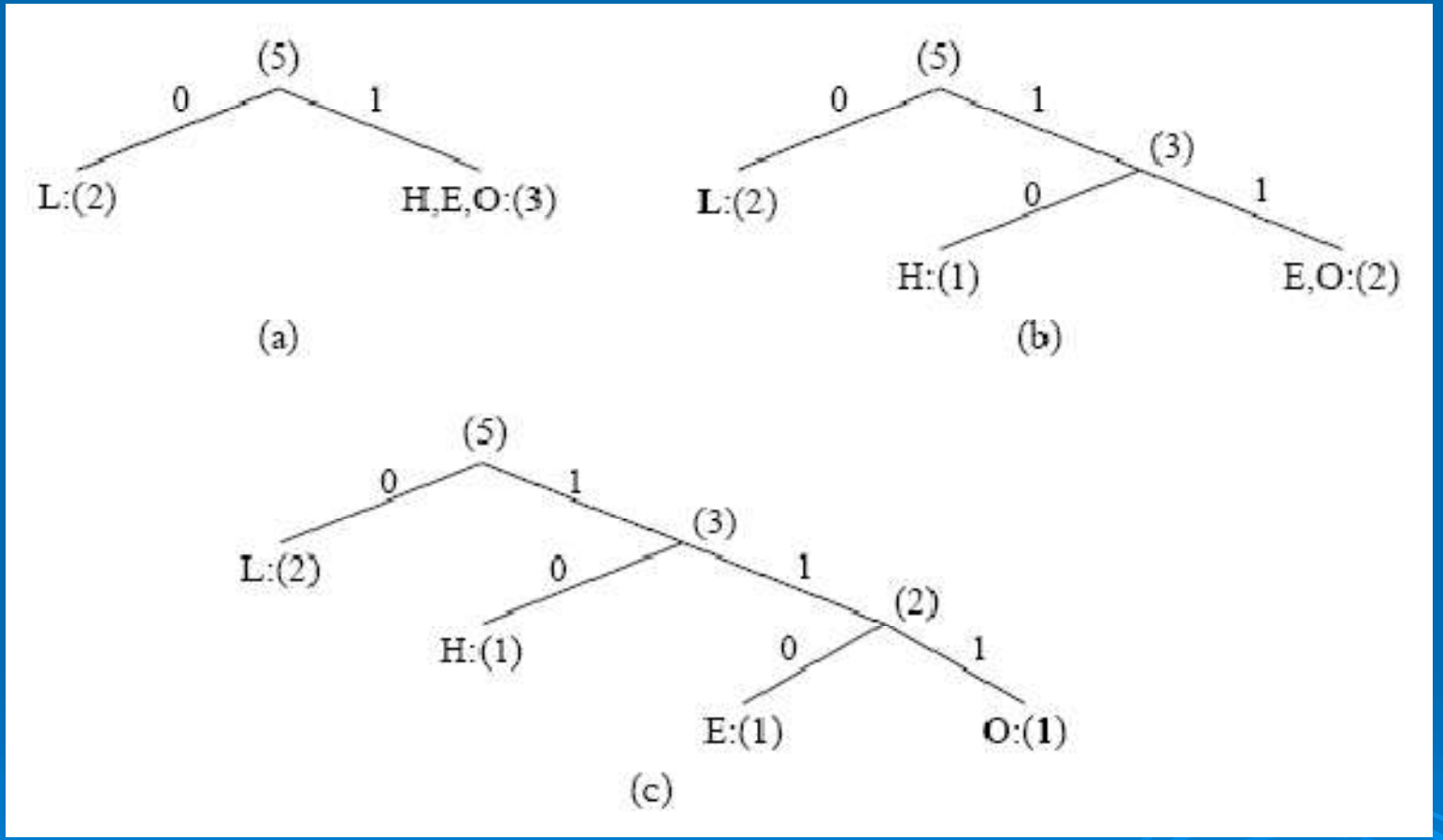
1. Sort the symbols according to the frequency count of their occurrences.
2. Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.

An Example: coding of “HELLO”

Symbol	H	E	L	O
Count	1	1	2	1

Frequency count of the symbols in "HELLO"

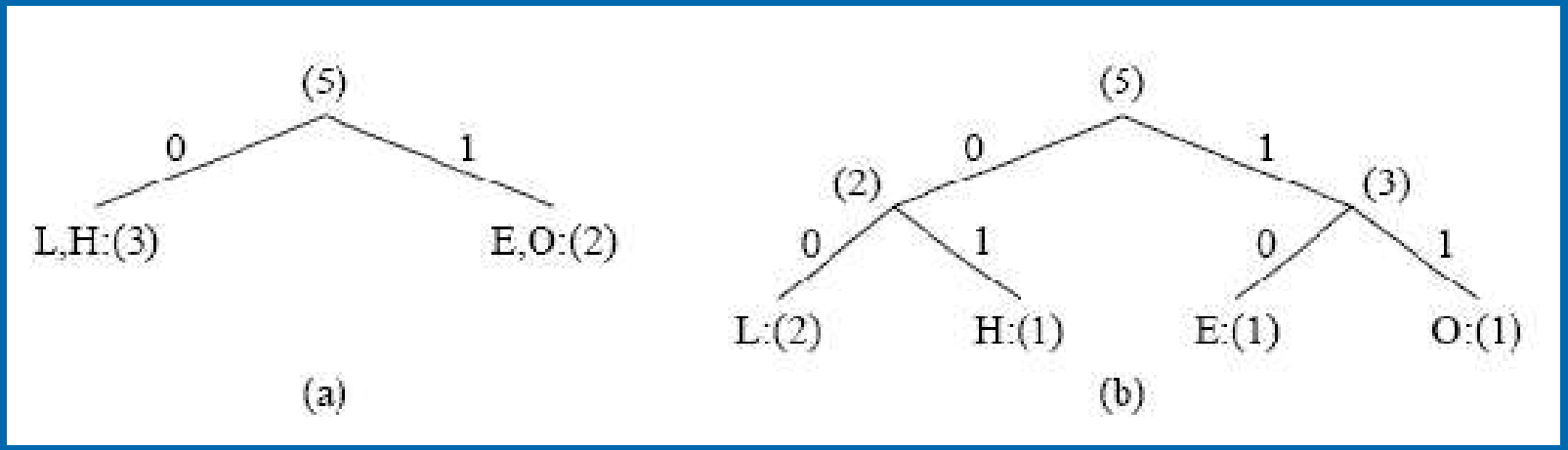
Coding Tree for HELLO by Shannon-Fano.



Result of Performing Shannon-Fano on HELLO

Symbol	Count	$\text{Log}_2\ 1/p_i$	Code	# of bits
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
Total number of bits				10

Another coding tree for HELLO by Shannon-Fano.



Symbol	Count	$\text{Log}_2\ 1/p_i$	Code	# of bits
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
Total number of bits				10

Huffman Coding

Huffman Coding Algorithm | a bottom-up approach

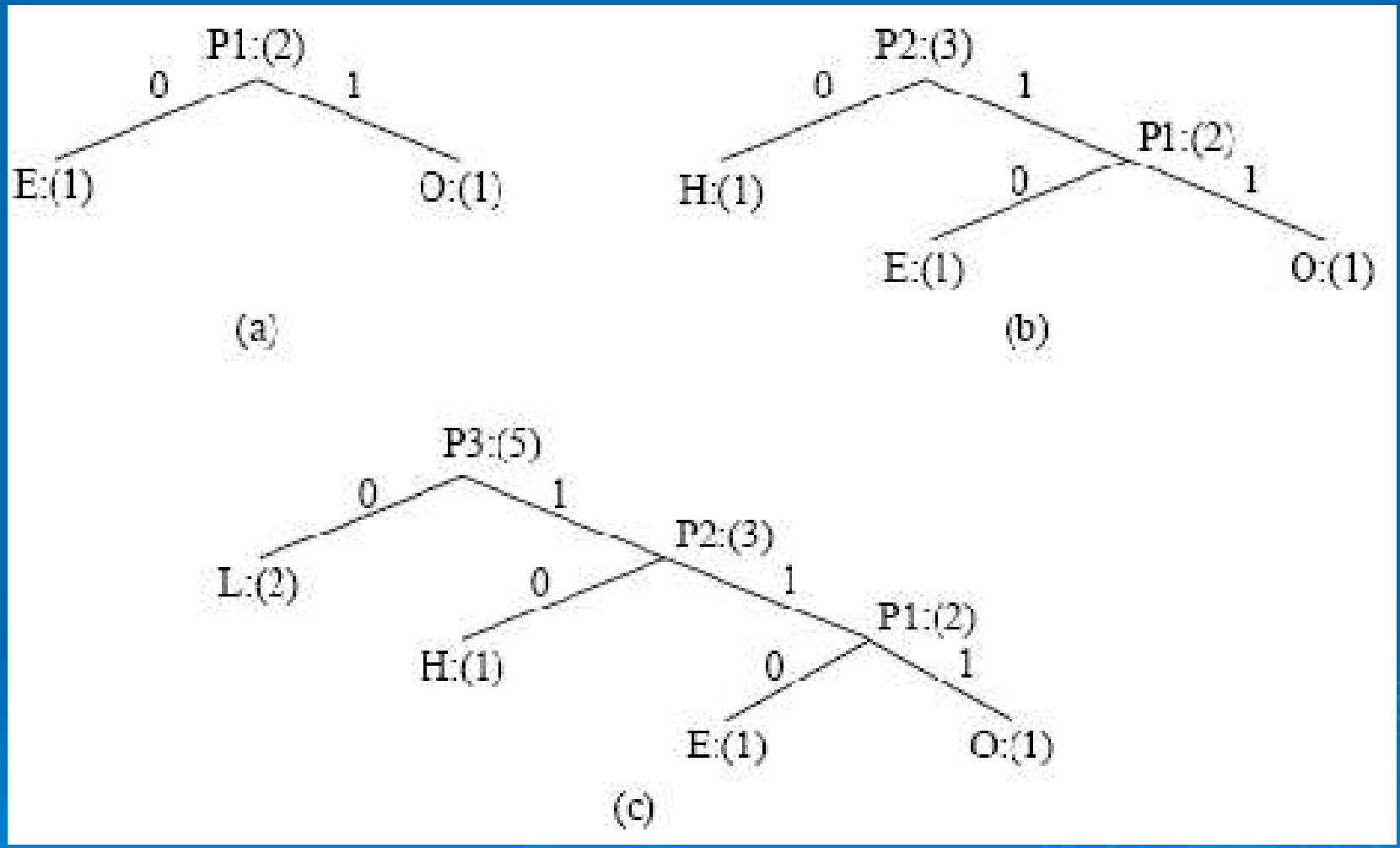
1. Initialization: Put all symbols on a list sorted according to their frequency counts.
2. Repeat until the list has only one symbol left:
    (1) From the list pick two symbols with the lowest frequency counts. Form a Huffman sub-tree that has these two symbols as child nodes and create a parent node.
    (2) Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
    (3) Delete the children from the list.
3. Assign a codeword for each leaf based on the path from the root.

Huffman Coding

- New symbols P1, P2, P3 are created to refer to the parent nodes in the Huffman coding tree. The contents in the list are illustrated below:

After initialization: L H E O
After iteration (a): L H P1
After iteration (b): L P2
After iteration (c): P3

Coding Tree for “HELLO” using the Huffman Algorithm.



Properties of Huffman Coding

1. **Unique Prefix Property**: No Huffman code is a prefix of any other Huffman code - prevents any ambiguity in decoding.
2. **Optimality**: *minimum redundancy code.*
   - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
   - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
   - The average code length for an information source *S* is strictly less than $\eta + 1$. We have:

$$\acute{\imath} < \eta + 1$$

Shannon-Fano vs. Huffman Coding

- Example: In a message, the codes and their frequencies are A(15), B(7), C(6), D(6), E(5). Encode this message with Shannon-fano and Huffman coding.

- Try yourself!

- Shannon-fano: 89 bits
- Huffman : 87 bits

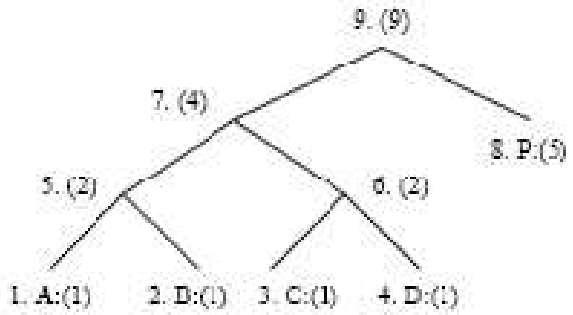
Adaptive Huffman Coding

- Statistics are gathered and updated dynamically as the data stream arrives.
- Symbols are assigned with some initially agreed upon codes, without any prior knowledge of the frequency counts.
- *Then, tree* construct is updated dynamically. Update basically does two things:
  - increments the frequency counts for the symbols (including any new ones).
  - updates the configuration of the tree.
- The *encoder* and *decoder* must use exactly the same *initial code* and *update tree* routines.

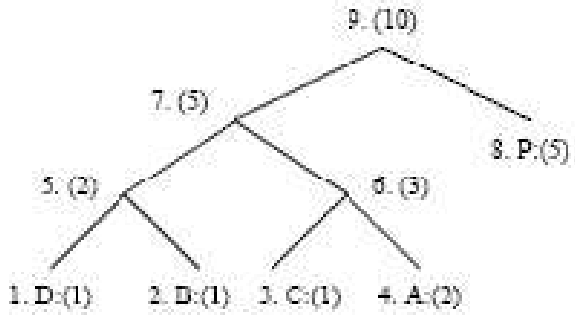
Notes on Adaptive Huffman Tree Updating

- Nodes are numbered in order from left to right, bottom to top. The numbers in parentheses indicates the count.
- The tree must always maintain its *sibling property*, i.e., all nodes (internal and leaf) are arranged in the order of increasing counts.
- If the sibling property is about to be violated, a *swap* procedure is invoked to update the tree by rearranging the nodes.
- When a swap is necessary, the farthest node with count *N* is swapped with the node whose count has just been increased to *N* +1.

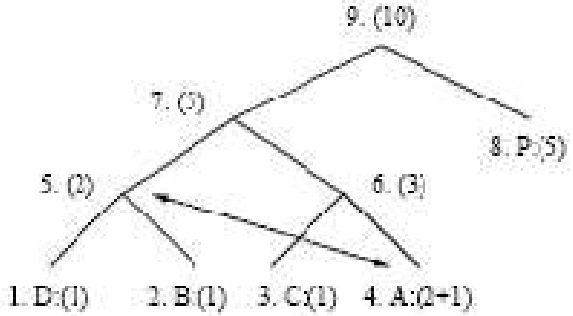
Node Swapping for Updating an Adaptive Huffman Tree



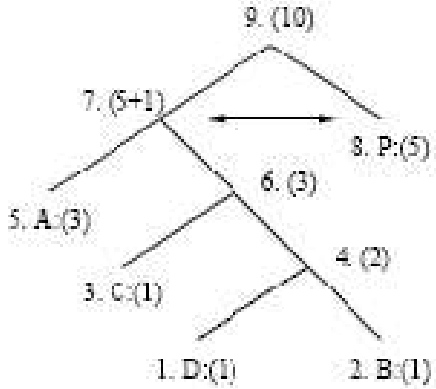
(a) A Huffman tree



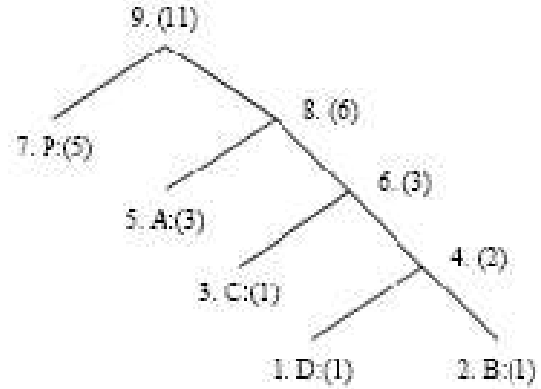
(b) Receiving 2nd 'A' triggered a swap



(c−1) A swap is needed after receiving 3rd 'A'



(c−2) Another swap is needed



(c−3) The Huffman tree after receiving 3rd 'A'

Another Example: Adaptive Huffman Coding

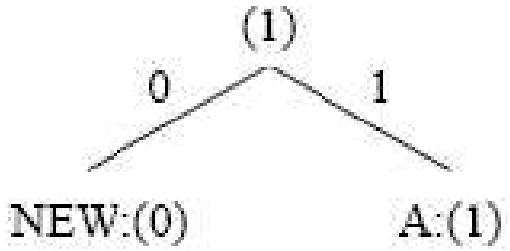
- This is to clearly illustrate more implementation details. We show exactly what *bits* are sent, as opposed to simply stating how the tree is updated.

- An additional rule: if any character/symbol is to be sent the first time, it must be preceded by a special symbol, NEW. The initial code for NEW is 0. The *count* for NEW is always kept as 0 (the count is never increased); hence it is always denoted as NEW:(0)

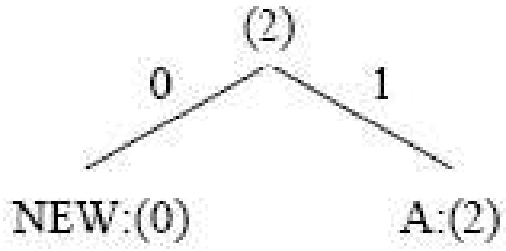
Initial code assignment for AADCCDD using adaptive Huffman coding.

Initial Code
NEW: 0
A: 00001
B: 00010
C: 00011
D: 00100
. .
. .
. .

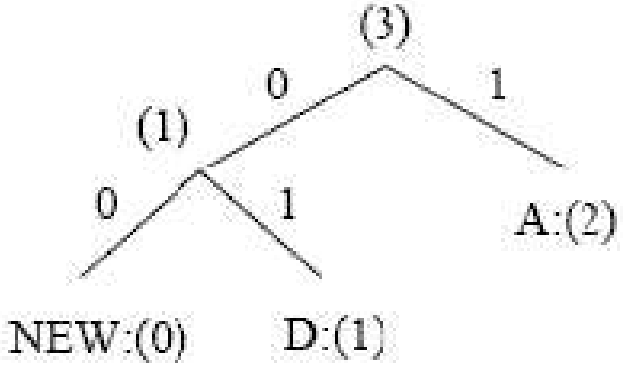
Adaptive Huffman tree for AADCCDD



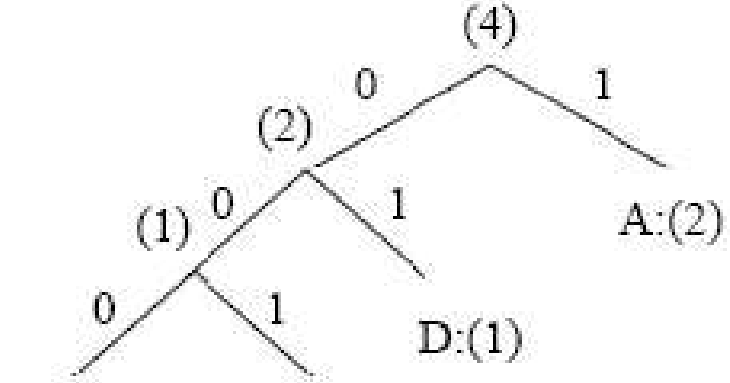
"A"



"AA"

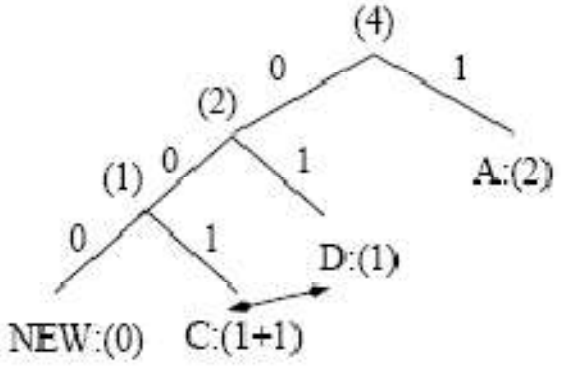


"AAD"

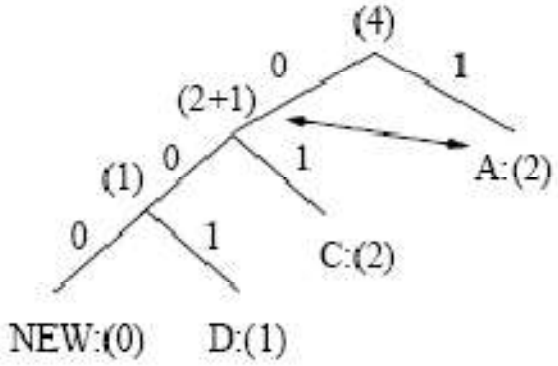


"AADC"

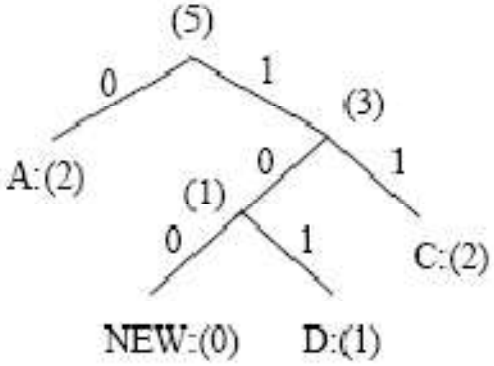
Adaptive Huffman tree for AADCCDD



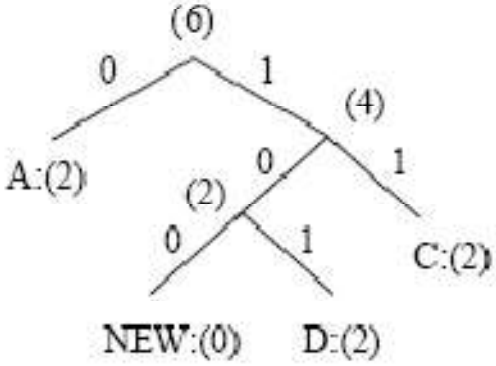
"AADCC" Step 1



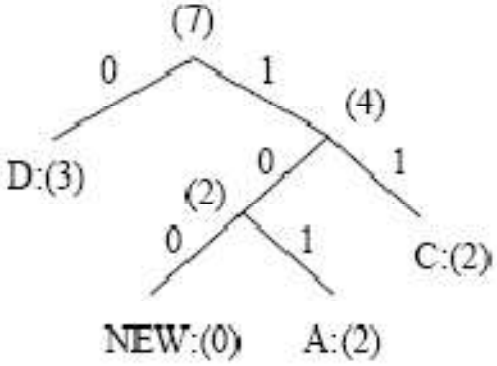
"AADCC" Step 2



"AADCC" Step 3



"AADCCD"



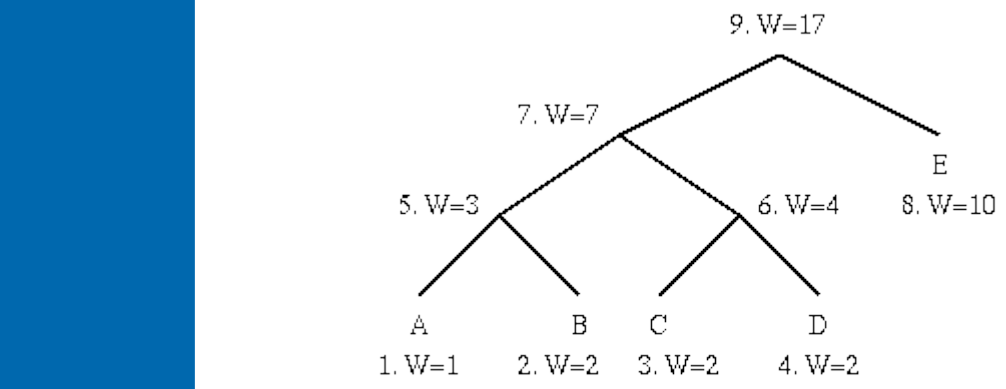
"AADCCDD"

Sequence of symbols and codes sent to the decoder

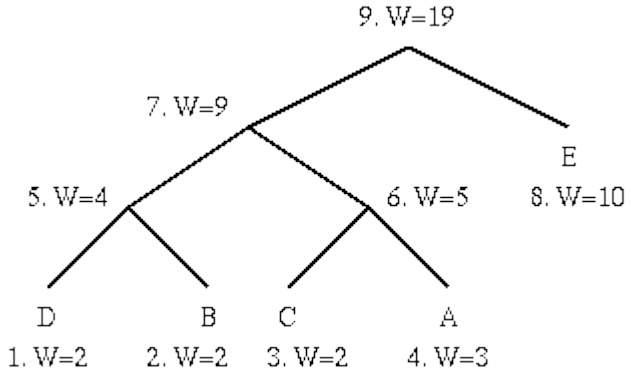
Symbol	NEW	A	A	NEW	D	NEW	C	C	D	D
Code	0	00001	1	0	00100	00	00011	001	101	101

- It is important to emphasize that the code for a particular symbol changes during the adaptive Huffman coding process.
- For example, after AADCCDD, when the character D over-takes A as the most frequent symbol, its code changes from 101 to 0.

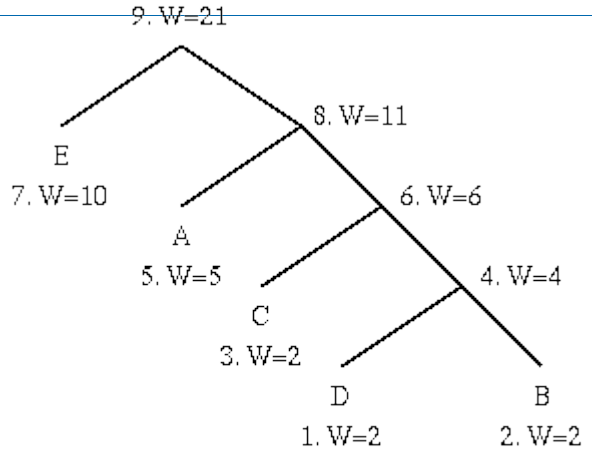
Example :



A Huffman Tree



After a node switch (A was incremented twice)



After A was incremented two more times

7.5 Dictionary-based Coding

7.5 Dictionary-based Coding

- LZW uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together, e.g., words in English text.
- The LZW encoder and decoder build up the same dictionary dynamically while receiving the data.
- LZW places longer and longer repeated entries into a dictionary, and then emits the *code* for an element, rather than the string itself, if the element has already been placed in the dictionary.

LZW compression for string “ABABBABCABABBA”

- Let's start with a very simple dictionary (also referred to as a “string table”), initially containing only 3 characters, with codes as follows:

code	string
1	A
2	B
3	C

- Now if the input string is “ABABBABCABABBA”, the LZW compression algorithm works as follows:

LZW Compression Algorithm

```
BEGIN
s = next input character;
while not EOF
   {   c = next input character;
       if s + c exists in the dictionary
          s = s + c;
      else
       { output the code for s;
       add string s + c to the dictionary with a new code;
         s = c;
       }
    }
output the code for s;
END
```

“ABABBABCABABBA”

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	A	4	10	ABA
A	B			
AB	B			
ABB	A	6	11	ABBA
A	EOF	1		

The output codes are: 1 2 4 5 2 3 4 6 1. Instead of sending 14 characters, only 9 codes need to be sent (compression ratio = 14/9 = 1.56).

LZW Decompression (simple version)

```
BEGIN
s = NIL;
while not EOF
   {
        k = next input code;
        entry = dictionary entry for k;
        output entry;
        if (s != NIL)
        add string s + entry[0] to dictionary with a new code;
        s = entry;
    }
END
```

1 2 4 5 2 3 4 6 1

The LZW decompression algorithm then works as follows:

S	k	entry/output	code	string
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	4	AB	9	CA
AB	6	ABB	10	ABA
ABB	1	A	11	ABBA
A	EOF			

Apparently, the output string is “ABABBABCABABBA”, a truly lossless result!

7.6 Arithmetic Coding

Arithmetic Coding

- Arithmetic coding is a more modern coding method that
- usually out-performs Huffman coding.

- Huffman coding assigns each symbol a codeword which has an integral bit length. Arithmetic coding can treat the whole message as one unit.

- A message is represented by a half-open interval [*a; b*) where *a* and *b* are real numbers between 0 and 1. Initially, the interval is [0*;* 1). When the message becomes longer, the length of the interval shortens and the number of bits needed to represent the interval increases.

Arithmetic Coding Encoder Algorithm

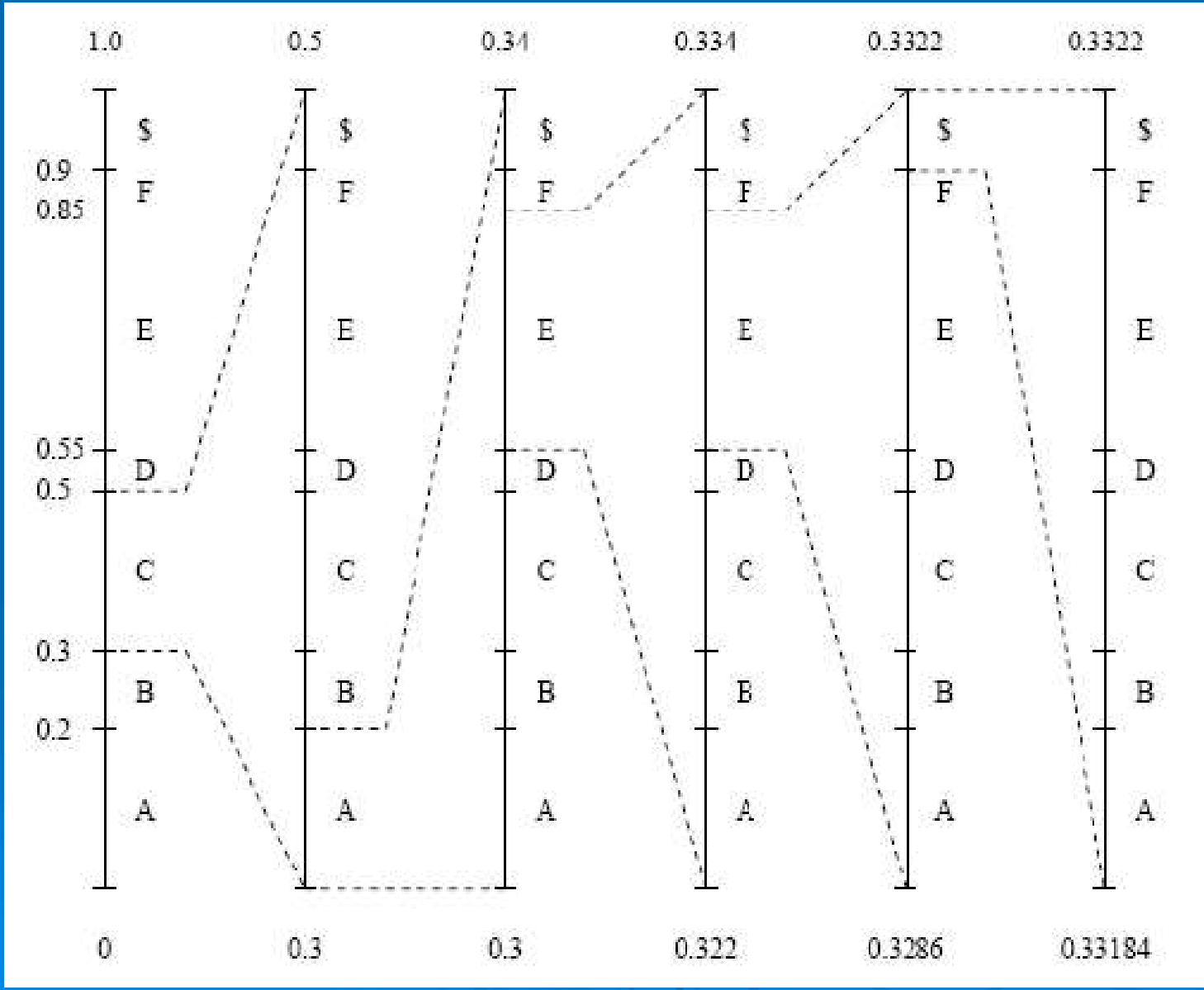
```
BEGIN
low = 0.0; high = 1.0; range = 1.0;
while (symbol != terminator)
  {
     get (symbol);
     low = low + range * Range_low(symbol);
     high = low + range * Range_high(symbol);
     range = high - low;
   }
output a code so that low <= code < high;
END
```

Example: Encoding in Arithmetic Coding

Symbol	Probability	Range
A	0.2	[0, 0.2)
B	0.1	[0.2, 0.3)
C	0.2	[0.3, 0.5)
D	0.05	[0.5, 0.55)
E	0.3	[0.55, 0.85)
F	0.05	[0.85, 0.9)
$	0.1	[0.9, 1.0)

Encode Symbols “CAEE$”

Graphical display of shrinking ranges



New *low*, *high*, and *range* generated.

Symbol	low	high	range
	0	1.0	1.0
C	0.3	0.5	0.2
A	0.30	0.34	0.04
E	0.322	0.334	0.012
E	0.3286	0.3322	0.0036
$	0.33184	0.33220	0.00036

The algorithm for extracting the ranges is:

Loop. For all the symbols.

Range = high_range of the symbol - low_range of the symbol

Number = number - low_range of the symbol

Number = number / range

Arithmetic coding: decode symbols "CAEE$"

value	Output Symbol	low	high	range
0.33203125	C	0.3	0.5	0.2
0.16015625	A	0.0	0.2	0.2
0.80078125	E	0.55	0.85	0.3
0.8359375	E	0.55	0.85	0.3
0.953125	$	0.9	1.0	0.1

6.7 Lossless Image Compression

Lossless Image Compression

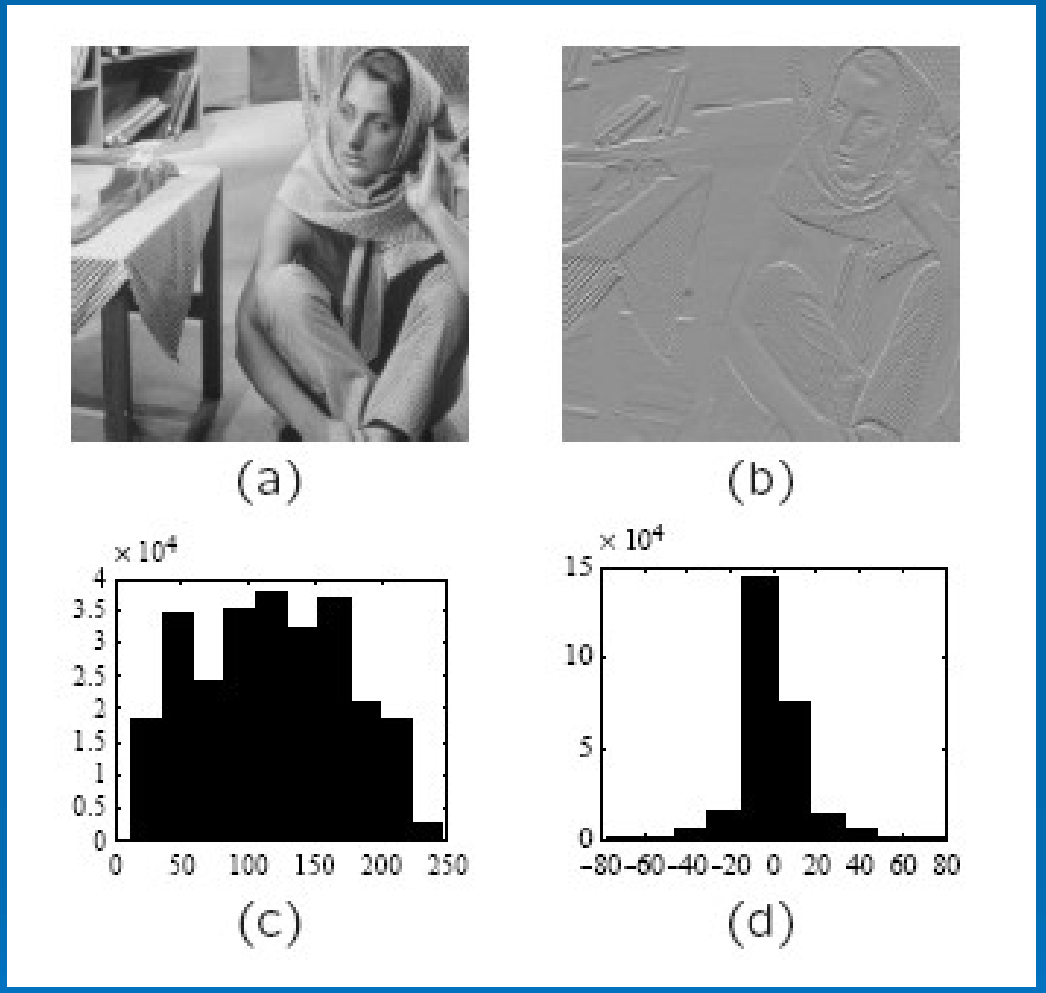
- **Approaches of Differential Coding of Images:**
  - Given an original image $I(x, y)$, using a simple difference operator we can define a difference image $d(x, y)$ as follows:

$$d(x, y) = I(x, y) - I(x - 1, y)$$

or use the discrete version of the 2-D Laplacian operator to define a difference image $d(x; y)$ as

$$d(x, y) = 4I(x, y) - I(x, y - 1) - I(x, y + 1) - I(x+1, y) - I(x-1, y)$$

- Due to *spatial redundancy* existed in normal images I, the difference image d will have a narrower histogram and hence a smaller entropy.



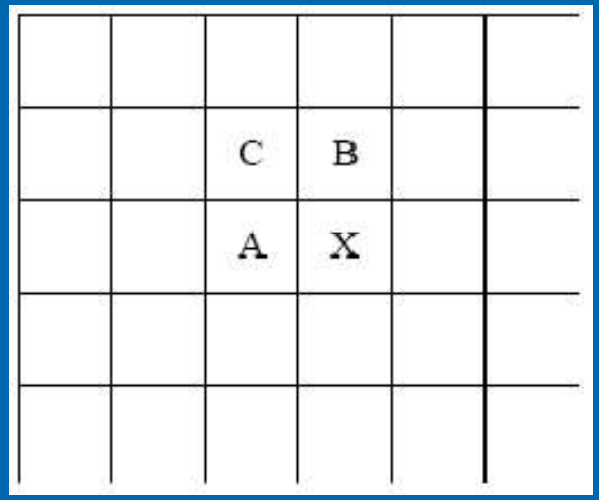
Distributions for Original versus Derivative Images.
(a,b): Original gray-level image and its partial derivative image;
(c,d): Histograms for original and derivative images.

Lossless JPEG

- **Lossless JPEG:** A special case of the JPEG image compression.
- **The Predictive method**
  1. **Forming a differential prediction:** A predictor combines the values of up to three neighboring pixels as the predicted value for the current pixel. The predictor can use any one of the seven schemes.

  2. **Encoding:** The encoder compares the prediction with the actual pixel value at the position `X' and encodes the difference using one of the lossless compression techniques we have discussed, e.g., the Huffman coding scheme.

Neighboring Pixels for Predictors in Lossless JPEG.



Note: Any of A, B, or C has already been decoded before it is used in the predictor, on the decoder side of an encode-decode cycle.

Predictors for Lossless JPEG

Predictor	Prediction
P1	A
P2	B
P3	C
P4	A + B - C
P5	A + (B - C) / 2
P6	B + (A - C) / 2
P7	(A + B) / 2

Comparison with other lossless compression programs

Compression Program	Compression Ratio			
	Lena	football	F-18	flowers
Lossless JPEG	1.45	1.54	2.29	1.26
Optimal lossless JPEG	1.49	1.67	2.71	1.33
compress (LZW)	0.86	1.24	2.21	0.87
gzip (LZ77)	1.08	1.36	3.10	1.05
gzip -9 (optimal LZ77)	1.08	1.36	3.13	1.05
pack (Huffman coding)	1.02	1.12	1.19	1.00

Lossless compression tools

- Entropy coding
  - *Huffman, Arithmetic, LZW, run-length*
- Predictive coding
  - *reduce the dynamic range to code*
- Transform
  - *enhance energy compaction*

Fundamentals of Multimedia

Chapter 12

MPEG Video Coding II

MPEG-4, 7

Ze-Nian Li & Mark S. Drew

Outline

12.1 Overview of MPEG-4

12.5 MPEG-4 Part10/H.264

12.6 MPEG-7

12.7 MPEG-21

12.1 Overview of MPEG-4

- MPEG-4: a newer standard. Besides compression, pays great attention to issues about user interactivities.
- MPEG-4 departs from its predecessors in adopting a new object-based coding:
  - Offering higher compression ratio, also beneficial for digital video composition, manipulation, indexing, and retrieval.
  - Figure 12.1 illustrates how MPEG-4 videos can be composed and manipulated by simple operations on the visual objects.

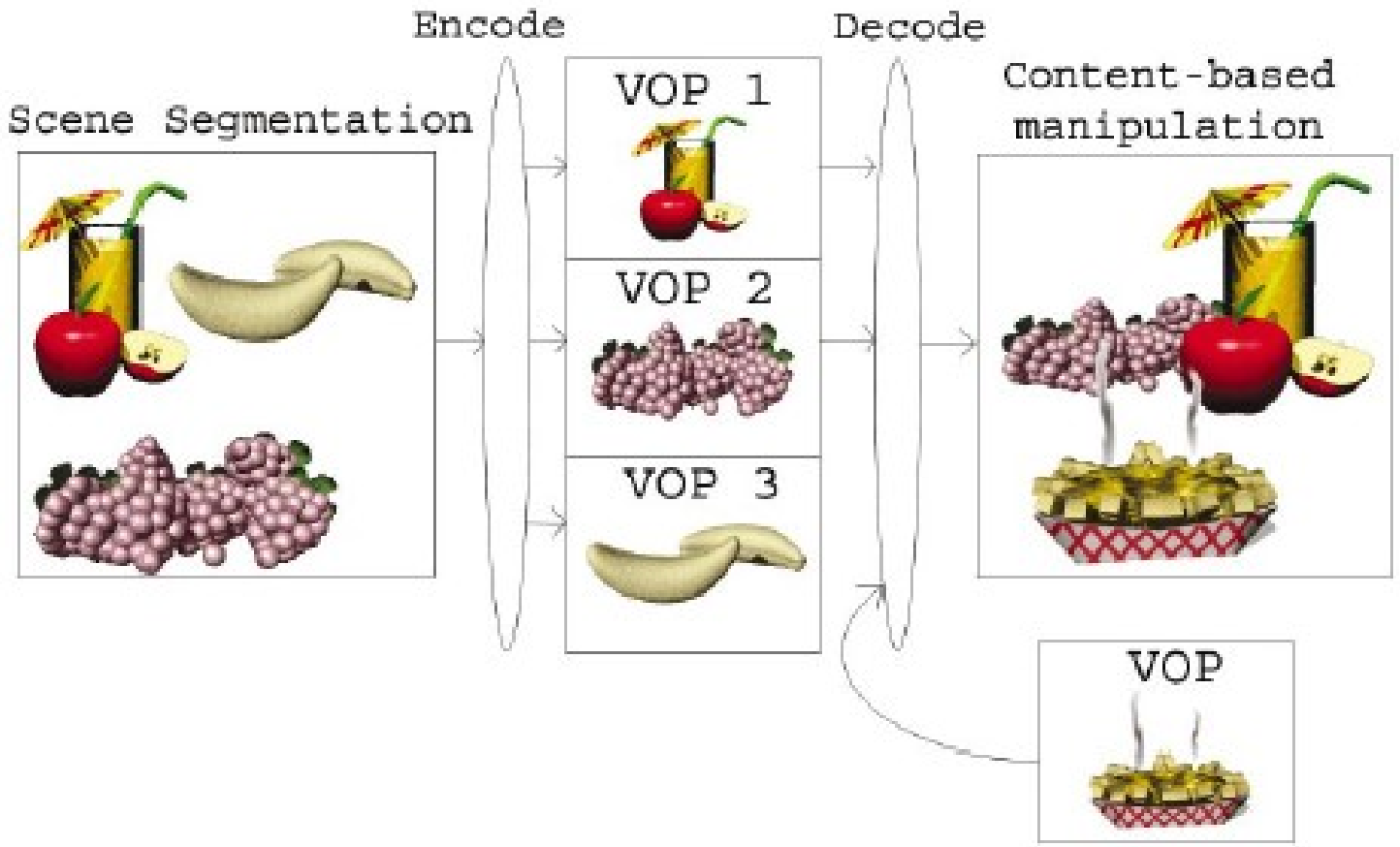


Fig. 12.1: Composition and Manipulation of MPEG-4 Videos.

Overview of MPEG-4

- MPEG-4 (Fig. 12.2(b)) is an entirely new standard for:

  (a) Composing media objects to create desirable audiovisual scenes.

  (b) Multiplexing and synchronizing the bitstreams for these media data entities so that they can be transmitted with guaranteed Quality of Service (QoS).

  (c) Interacting with the audiovisual scene at the receiving end - provides a toolbox of advanced coding modules and algorithms for audio and video compressions.

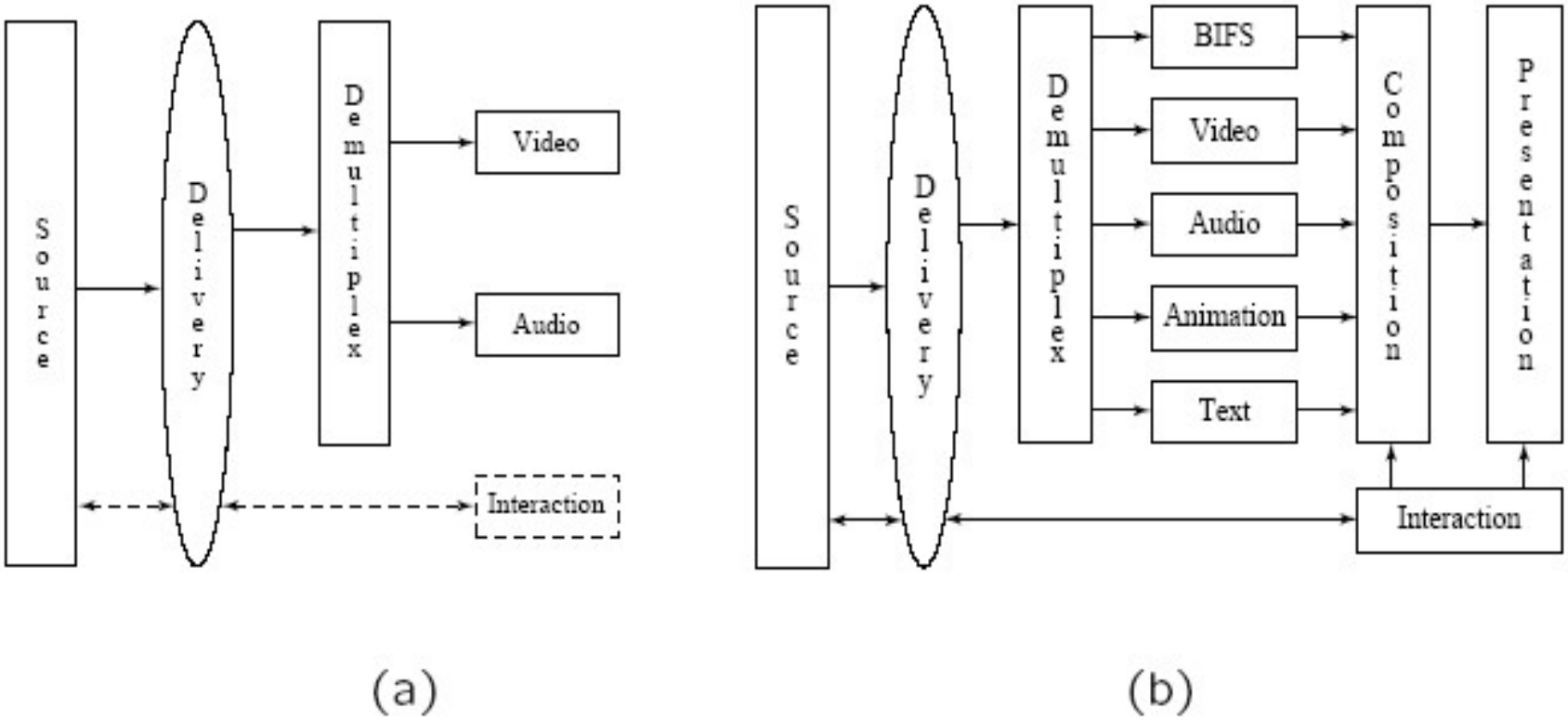


Fig. 12.2: Comparison of interactivities in MPEG standards: (a) reference models in MPEG-1 and 2 (b) MPEG-4 reference model.

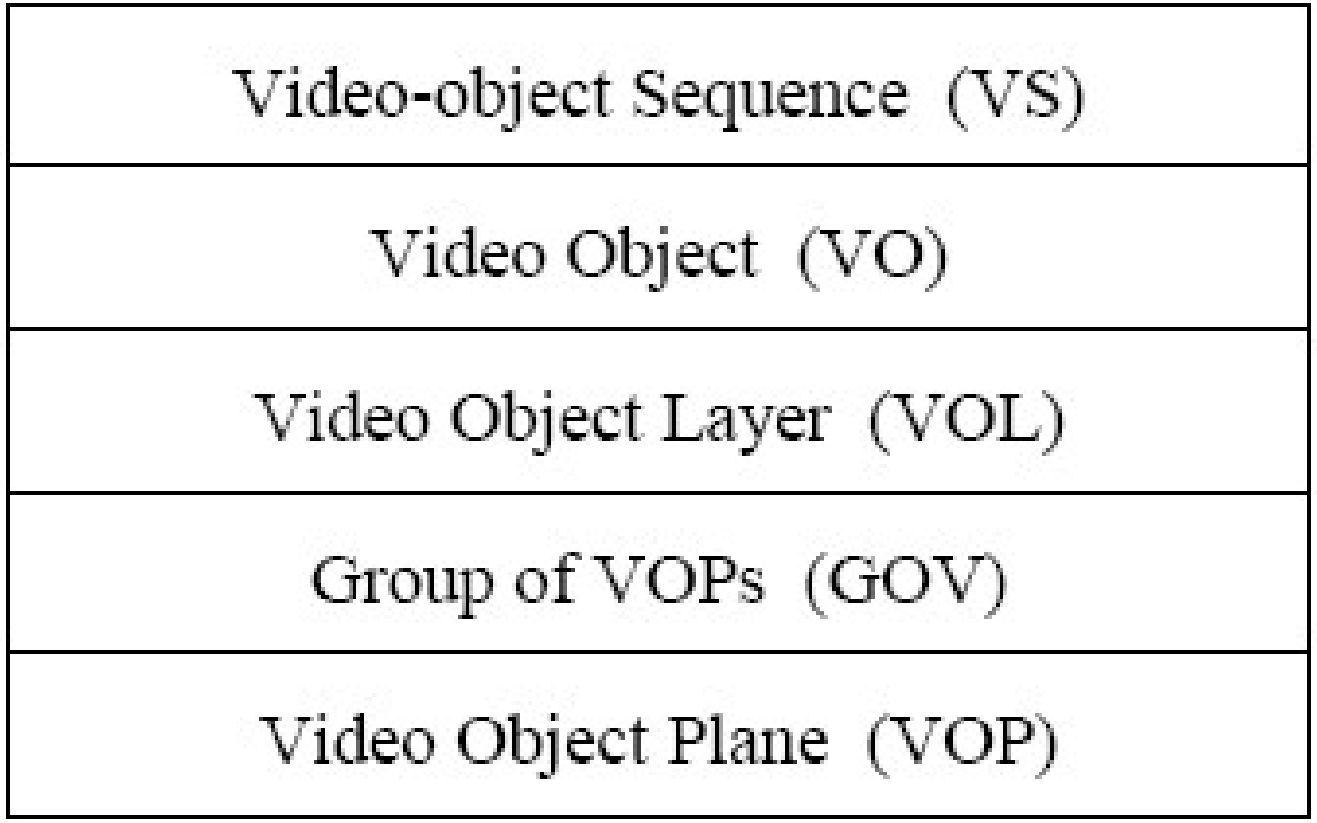


Fig. 12.3: Video Object Oriented Hierarchical Description of a Scene in MPEG-4 Visual Bitstreams.

12.5 MPEG-4 Part10/H.264

- The H.264 video compression standard, formerly known as H.26L, is being developed by the Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG.
- Preliminary studies using software based on this new standard suggests that H.264 offers up to 30-50% better compression than MPEG-2, and up to 30% over H.263+ and MPEG-4 advanced simple profile.

- The outcome of this work is actually two identical standards:

  ISO MPEG-4 Part10 and ITU-T H.264.

- H.264 is currently one of the leading candidates to carry High Definition TV (HDTV) video content on many potential applications.

H.264 Core Features

- VLC-Based Entropy Decoding:

  Two entropy methods are used in the variable-length entropy decoder: Unified-VLC (UVLC) and Context Adaptive VLC (CAVLC).

- Motion Compensation (P-Prediction):

- Uses a tree-structured motion segmentation down to 4×4 block size

  (16×16, 16×8, 8×16, 8×8, 8×4, 4×8, 4×4).

- This allows much more accurate motion compensation of moving objects. Motion vectors can be up to half-pixel or quarter-pixel accuracy.

H.264 Core Features

- Intra-Prediction (I-Prediction):
- H.264 exploits much more spatial prediction than in previous video standards such as H.263+.

- Uses a simple integer-precision 4×4 DCT, and a quantization scheme with nonlinear step-sizes.

- In-Loop Deblocking Filters.

Baseline Profile Features

- The Baseline profile of H.264 is intended for real-time conversational applications, such as video conferencing.
- It contains all the core coding tools of H.264 discussed above and the following additional error-resilience tools, to allow for error-prone carriers such as IP and wireless networks:
- Arbitrary slice order (ASO).
- Flexible macroblock order (FMO).
- Redundant slices.

Main Profile Features

- Represents non-low-delay applications such as broadcasting and stored-medium.
- The Main profile contains all Baseline profile features (except ASO, FMO, and redundant slices) plus the following:
- B slices.
- Context Adaptive Binary Arithmetic Coding (CABAC).
- Weighted Prediction.

Extended Profile Features

- Represents non-low-delay applications such as broadcasting and stored-medium.
- The eXtended profile (or profile X) is designed for the new video streaming applications.
- This profile allows non-low-delay features, bitstream switching features, and also more error-resilience tools.

12.6 MPEG-7

- The main objective of MPEG-7 is to serve the need of audiovisual content-based retrieval (or audiovisual object retrieval) in applications such as digital libraries.
- Nevertheless, it is also applicable to any multimedia applications involving the generation (content creation) and usage (content consumption) of multimedia data.
- MPEG-7 became an International Standard in September 2001 - with the formal name Multimedia Content Description Interface.

Applications Supported by MPEG-7

- MPEG-7 supports a variety of multimedia applications. Its data may include still pictures, graphics, 3D models, audio, speech, video, and composition information.
- These MPEG-7 data elements can be represented in textual format, or binary format, or both.
- Fig. 12.17 illustrates some possible applications that will benefit from the MPEG-7 standard.