Chapter 1

What Computer Graphics Is

Computer graphics is an interdisciplinary field where computer scientists, mathematicians, physicists, engineers, artists and practitioners all gather with the common goal of opening a “window” to the “world". In the previous sentence “window” is the monitor of a computer, the display of a tablet or a smartphone or anything that can show images. The “world” is a digital model, the result of a scientific simulation, or any entity for which we can conceive a visual representation. The goal of this chapter is to provide the first basic knowledge that the reader will need through the rest of the book during the learning of how to develop his/her own interactive graphics application.

1.1 Application Domains and Areas of Computer Graphics

Computer graphics (CG) deals with all the algorithms, methods and techniques that are used to produce a computer-generated image, a *synthetic image*, starting from a collection of data. This data can be the description of a 3D scene, like in a videogame; some physical measures coming from scientific experiments, like in scientific visualization; or statistics collected through the Web visualized in a compact way for summarization purposes, like in an information visualization application. The process of converting the input data into an image is called *rendering*.

During the past twenty years, computer graphics has progressively spread over almost all areas of life. This diffusion has been mainly facilitated by the increasing power and flexibility of the consumer graphics hardware, which provides the capability to a standard PC to render very complex 3D scenes, and by the great effort of the researchers and developers of the computer graphics community to create efficient algorithms, which enable the developer to carry out a wide range of visualization tasks. When you play a computer game, many complex CG algorithms are at work to render your battlefield/space ship/cars, when you go to the cinema you may see your latest movie partly or entirely generated through a computer and a bunch of CG algorithms, when you are writing your business-planning presentation, graphics help you to summarize trends and other information in a easy-to-understand way.

1.1.1 Application Domains

We mentioned just a few examples but CG applications span over a lot of different ambits. Without expecting to be exhaustive, we give here a short list of application fields of CG.

Entertainment Industry: creation of synthetic movies/cartoons, creation of visual special effects, creation of visually pleasant computer games.

Architecture: visualization of how the landscape of a city appears before and after the construction of a building, design optimization of complex architectural structures.

Mechanical Engineering: creation of virtual prototypes of mechanical pieces before the actual realization, for example in the automotive industry.

Design: to enhance/aid the creativity of a designer who can play with several shapes before producing his/her final idea, to test the feasibil-

ity of fabricating objects.

Medicine: to train surgeons through virtual surgery simulations, to efficiently visualize data coming from diagnostic instruments, and to plan difficult procedures on a virtual model before the real intervention.

Natural Science: to visualize complex molecules in drugs development, to enhance microscope images, to create a visualization of a theory about a physical phenomenon, to give a visual representation of physical measures coming from an experiment.

Cultural Heritage: to create virtual reconstructions of ancient temples or archeological sites; to show reconstruction hypotheses, for example how ancient Rome appeared in its magnificence, for conservation and documentation purposes.

1.1.2 Areas of Computer Graphics

As mentioned in the introduction, computer graphics is a very general concept encompassing a wide background knowledge. As such, it has naturally evolved into a number of areas of expertise, the most relevant of which are:

Imaging: In recent years many image processing algorithms and techniques have been adopted and extended by the CG community to produce high quality images/videos. Matting, compositing, warping, filtering and editing are common operations of this type. Some advanced tasks of this type are: *texture synthesis*, which deals with the generation of visual patterns of surface such as bricks of a wall, clouds in the sky, skin, facades of buildings, etc.; *intelligent cut-and-paste*, an image editing operation where the user selects a part of interest of an image and modifies it by interactively moving it and integrates it into the surroundings of another part of the same or other image; *media retargeting*, which consists of changing an image so as

to optimize its appearance in a specific media. A classic example is how to crop and/or extend an image to show a movie originally shot in a cinematographic 2.39 : 1 format (the usual notation of the *aspect ratio* $x : y$ means that the ratio between the width and the height of the image is $\frac{x}{y}$) to the more TV-like 16 : 9 format.

3D Scanning: The process of converting real world objects into a digital representation than can be used in a CG application. Many devices and algorithms have been developed to acquire the geometry and the visual appearance of a real object.

Geometric Modeling: Geometric modeling concerns the modeling of the 3D object used in the CG application. The 3D models can be generated manually by an expert user with specific tools or semi-automatically by specifying a sketch of the 3D object on some photos of it assisted by a specific drawing application (this process is known as *image-based modeling*).

Geometric Processing: Geometric processing deals with all the algorithms used to manipulate the geometry of the 3D object. The 3D object can be *simplified*, reducing the level of details of the geometry component; *improved*, by removing noise from its surface or other topological anomalies; *re-shaped* to account for certain characteristics; *converted into different types of representation*, as we will see in **Chapter 3**; and so on. Many of these techniques are related to the field of *computational geometry*.

Animation and Simulation: This area concerns all the techniques and algorithms used to animate a static 3D model, ranging from the techniques to help the artist to define the movement of a character in a movie to the real-time physical simulation of living organs in a surgery simulator. Much of the work in this area is rooted in the domain of mechanical engineering, from where complex algorithms have been adapted to run on low-end computers and in real time, often trading accuracy of physical simulation for execution speed.

Computational Photography: This area includes all the techniques employed to improve the potential of digital photography and the quality of digitally captured images. This CG topic spans optics, image processing and computer vision. It is a growing field that has allowed us to produce low-cost digital photographic devices capable of identifying faces, refocusing images, automatically creating panoramas, capturing images in high dynamic range, estimating the depth of the captured scene, etc.

Rendering: We have just stated that rendering is the process of producing a final image starting from some sort of data. Rendering can be categorized in many ways depending on the property of the rendering algorithm. A commonly used categorization is to subdivide rendering techniques into *photorealistic rendering, non-photorealistic rendering* or *information visualization*. The aim of photorealistic rendering is to produce a synthetic image as realistic as possible starting from a detailed description of the 3D scene in terms of geometry, both at macroscopic and microscopic levels, and materials. Non-photorealistic rendering (NPR) deals with all the rendering techniques that relax the goal of realism. For example, to visualize a car engine, the rendering should emphasize each of its constituent elements; in this sense a realistic visualization is less useful from a perceptual point of view. For this reason sometimes NPR is also referred to as *illustrative rendering*. *Information visualization* concerns the visualization of huge amounts of data and their relationships, usually it adopts schemes, graphs and charts. The visualization techniques of this type are usually simple; the main goal is to express visually, in a clear way, the data and their underlying relationships.

Another way to classify rendering algorithms is the amount of time they require to produce the synthetic image. The term *real-time rendering* refers to all the algorithms and techniques that can be used to generate the images so fast as to guarantee user interaction with the graphics application. In this ambit, computer game developers have

pushed the technologies to become capable of handling scenes of increasing complexity and realism at interactive rates, which means generating the synthetic image in about 40–50 milliseconds, which guarantees that the scene is drawn 20–25 times per second. The number of times a scene is drawn on the screen of a display surface is called *framerate* and it is measured in *frames-per-second* (fps). Many modern computer games can achieve 100 fps or more. *Offline rendering* deals with all the algorithms and techniques to generate photorealistic images of a synthetic scene without the constraint of interactivity. For example, the images produced for an animation movie are usually the result of off-line algorithms that run for hours on a dedicated cluster of PCs (called *render farm*) and simulate the interaction between the light and the matter by means of *global illumination* (GI) techniques. Traditionally the term global-illumination technique implied off-line rendering. Thanks especially to the improvements in CG hardware, this is not entirely true anymore; there are many modern techniques to introduce effects of global illumination in real-time rendering engines.

1.2 Color and Images

Color is a fundamental aspect of computer graphics. Colors are used to communicate visually in several ways, for example an image full of cold colors (blue, gray, green) give us completely different sensations than an image with hot colors (red, yellow, orange). Colors also influence our attention, for example the color orange captures the attention of an observer more than other colors.

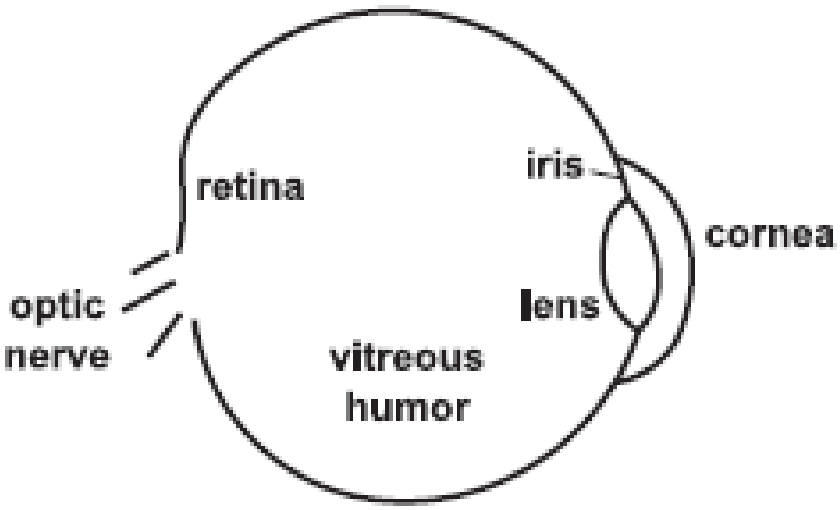
When we talk about color we have to think at two levels: a *physical level*, which concerns the physics rules involved in the creation of a color stimulus, given by the light that hits a surface and then reaches our eye, and a *subjective* or *perceptual* level, which concerns how we perceive such color stimulus. Both the physical and perceptual processes involved in how we see colors allow us to manage the color

creation process. A complete treatment of colors is beyond the scope of this book. Here, we provide some basic concepts that will be useful to us in understanding how to handle colors in our graphics application.

1.2.1 The Human Visual System (HVS)

The *human visual system (HVS)* is composed of the eyes, which capture the light, the physical color stimuli, and the brain, which interprets the visual stimuli coming from them. Our eyes respond to color stimuli. By *color stimulus* we mean a radiation of energy emitted by some source, reflected by an object that hits the retina, entering into the eye through the cornea (see **Figure 1.1**). The retina contains the receptors that generate neural signals when stimulated by the energy of the light incident on them. Not all radiation can be perceived by our visual system. Light can be described by its wavelength; *visible light*, the only radiation that is perceived by the human visual system, has a wavelength range from 380 nm to 780 nm. Infrared and microwaves have wavelength greater than 780 nm. Ultraviolet and X-ray have wavelengths less than 380 nm.

Figure 1.1



Structure of a human eye.

The light receptors on the retina are of two types: rods and cones. Rods are capable of detecting very small amounts of light, and produce a signal that is interpreted as monochromatic. Imagine that one is observing the stars during the night: rods are in use at that moment. Cones are less sensitive to light than rods, but they are our color receptors. During the day, light intensities are so high that rods get saturated and become nonfunctional, and that is when cone receptors come into use. There are three types of cones: They are termed long (L), medium (M) and short (S) cones depending on the part of the visible spectrum to which they are sensitive. S cones are sensitive to the lower part of the visible light wavelengths, M cones are sensitive to the middle wavelengths of the visible light and L cones are sensitive to the upper part of the visible spectrum. When the cones receive incident light, they produce signals according to their sensitivity and the intensity of the light, and send them to the brain for interpretation. The three different cones produce three different signals, which gives rise to the *trichromacy* nature of color (in this sense human beings are *trichromats*). Trichromacy is the reason why different color stimuli may be perceived as the same color. This effect is called *metamerism*. Metamerism can be distinguished into *illumination metamerism* when the same color is perceived differently

when the illumination changes, and *observer metamerism* when the same color stimulus is perceived differently by two different observers.

The light receptors (rods, cones) do not have a direct specific individual connection to the brain but groups of rods and cones are interconnected to form receptive fields. Signals from these receptive fields reach the brain through the optic nerve. This interconnection influences the results of the signals produced by the light receptors. Three types of receptive fields can be classified: black-white, red-green and yellow-blue. These three receptive fields are called *opponent channels* . It is interesting to point out that the black-white channel is the signal that has the highest spatial resolution on the retina; this is the reason why human eyes are more sensitive to brightness changes of an image than to color changes. This property is used in image compression when color information is compressed in a more aggressive way than luminous information.

1.2.2 Color Space

Since color is influenced by many objective and subjective factors, it is difficult to define a unique way to represent it. We have just stated that color is the result of the trichromatic nature of the HVS, hence the most natural way to represent a color is to define it as a combination of three *primary colors*. These primary colors are typically combined following two models: *additive* and *subtractive*.

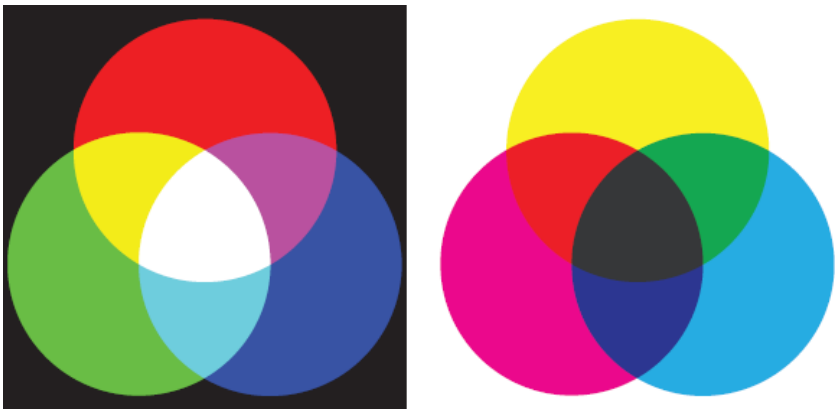
With the additive color model, the stimulus is generated by combining different stimuli of three individual colors. If we think of three lamps projecting a set of primary colors, for example, red, green and blue, on a white wall in a completely dark room, then the white wall will reflect the additive combination of the three colors towards our eye. All the colors can be obtained by properly adjusting the intensity of the red, green and blue light.

With the subtractive color model the stimulus is generated by subtracting the wavelengths from the light incident on the reflector. The most well known example of use is the cyan, magenta, yellow and key (black) model for printing. Assume that we are printing on a white paper and we have a white light. If we add no inks to the paper, then we will see the paper as white. If we put cyan ink on the paper then the paper will look cyan because the ink absorbs the red wavelengths. If we also add yellow onto it then the blue wavelengths will also be absorbed and the paper will look green. Finally, if we add the magenta ink we will have a combination of inks that will absorb all the wavelengths so the paper will look black. By modulating the amount of each primary ink or color we put on paper in theory we can express every color of the spectrum. So why the black ink? Since neither the ink nor the paper is ideal, in the real situation you would not actually obtain black, but just a very dark color and hence black ink is used instead. So in general, a certain amount of black is used to absorb *all* the wavelengths, and this amount is the minimum of the three primary components of the color. For example, if we want the color (c, m, y) the printer will combine:

$$K = \min(c, m, y) \quad (1.1)$$
$$C = c - K$$
$$M = m - K$$
$$Y = y - K \quad (1.2)$$

Figure 1.2 (Left) shows an example of additive primaries while **Figure 1.2** (Right) shows a set of subtractive primaries.

Figure 1.2



(Left) RGB additive primaries. (Right) CMY subtractive primaries.

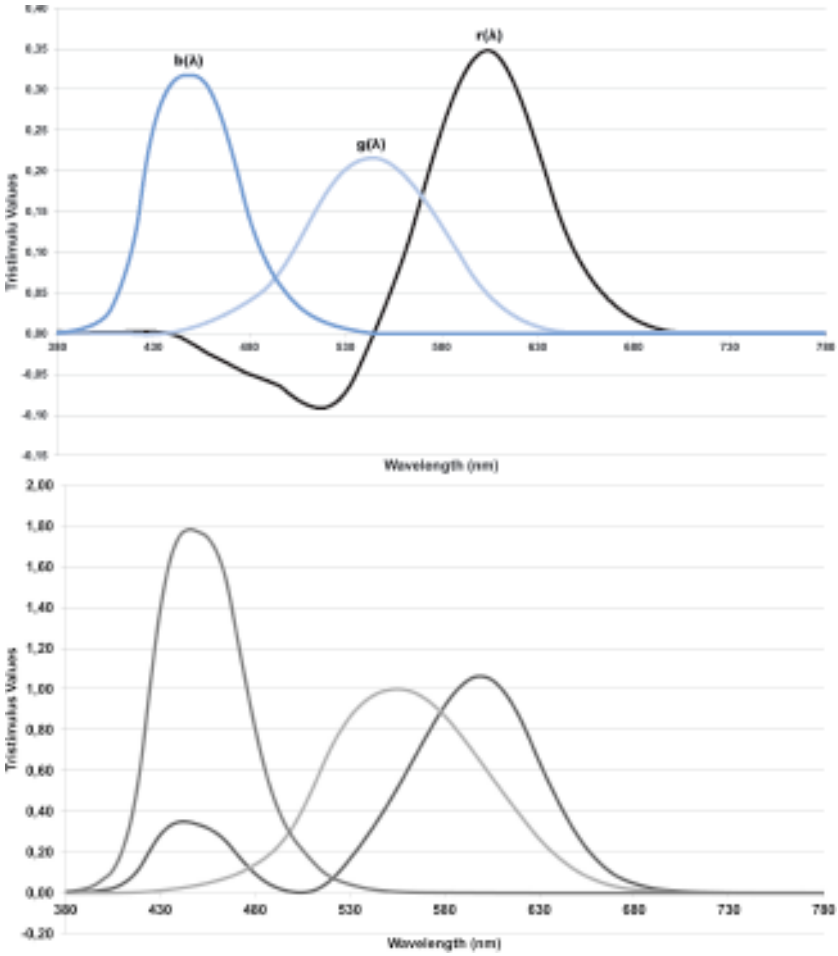
According to the field of applications and the characteristics of the primaries many color system can be defined. In the following we describe some of the most important color systems.

1.2.2.1 CIE XYZ

One of the most important color systems is the one defined by the *Commission International de l'Eclairage* (CIE) as a standard in 1931. This color system is based on the following experiment: after defining certain viewing conditions, like the distance from the target and the illumination conditions, a subject was asked to match two different colors by tuning certain parameters. The collection of responses to the monochromatic colors of a number of visible wavelengths from many subjects (a total of 17 color-normal observers) was used to define the "mean" response of a human observer, named *standard observer*, to the color stimulus. The response of the standard observer was encoded in a set of functions named *color matching functions*, shown in **Figure 1.3**. These functions are usually indicated as $\overline{r}(\lambda), \overline{g}(\lambda)$ and $\overline{b}(\lambda)$ and enable us to define a color by integrating its power spectrum distribution $I(\lambda)$:

$$R = \int_{380}^{780} I(\lambda)\overline{r}(\lambda)d\lambda$$
$$G = \int_{380}^{780} I(\lambda)\overline{g}(\lambda)d\lambda \quad (1.3)$$
$$B = \int_{380}^{780} I(\lambda)\overline{b}(\lambda)d\lambda$$

Figure 1.3



(Top) CIE 1931 RGB color matching function $(\overline{x}(\lambda), \overline{y}(\lambda), \overline{z}(\lambda))$. (Bottom) CIEXYZ color matching functions $\left(\overline{r}(\lambda), \overline{g}(\lambda), \overline{b}(\lambda)\right)$.

The plots on the bottom of **Figure 1.3** shows the CIEXYZ primaries $\overline{x}(\lambda), \overline{y}(\lambda)$ and $\overline{z}(\lambda)$. These matching functions are a transformed version of the CIERGB color matching functions in such a way that they are all positive functions and hence are used to simplify the design of devices for color reproduction. The equation to transform the CIERGB color space to the CIEXYZ color space is the following:

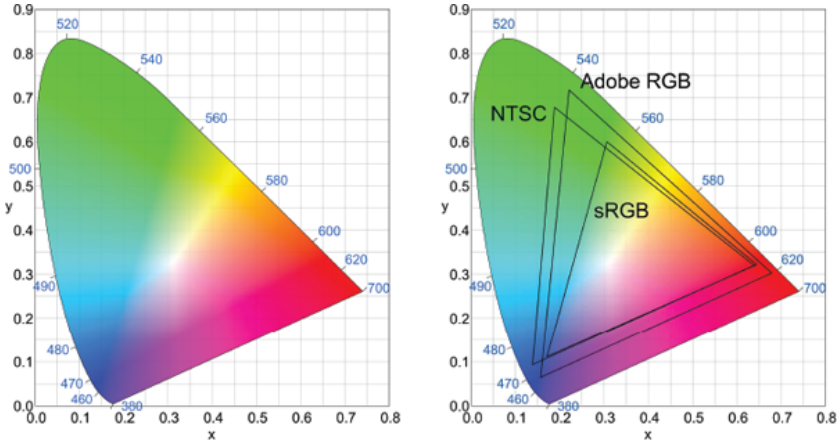
$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4887180 & 0.3106803 & 0.2006017 \\ 0.1762044 & 0.8129847 & 0.0108109 \\ 0.0000000 & 0.0102048 & 0.9897952 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (1.4)$$

The normalized version of the XYZ color coordinates can be used to define the so-called *chromaticity coordinates*:

$$x = \frac{X}{X+Y+Z} \quad y = \frac{Y}{X+Y+Z} \quad z = \frac{Z}{X+Y+Z} \quad (1.5)$$

These coordinates do not completely specify a color since they always sum to one, and hence are specified by only two coordinates, which means a two-dimensional representation. Typically, the *Y* is added together with *x* and *y* to fully define trichromatic color space called *xyY*. The chromaticity coordinates *x* and *y* just mentioned are usually employed to visualize a representation of the colors as in the *chromaticities diagram* shown in **Figure 1.4** (on the left). Note that even the ink used in professional printing is not capable of reproducing all the colors of the chromaticities diagram.

Figure 1.4



(Left) Chromaticities diagram. (Right) Gamut of different RGB color systems.

1.2.2.2 Device-Dependent and Device-Independent Color Space

So far we talked about *device-independent* color space, that is, color spaces that are able to represent every color and that do not take into

account the particular way to physically produce it. In the real world, the color devices, such as printers and monitors, have to deal with the fact that not all the colors are physically reproducible by one system. The *gamut* is the set of colors that a particular device can output. Typically, the color gamut is depicted on the chromaticities diagram just described. **Figure 1.4** (on the right) shows an example of the gamut of commonly used RGB color spaces, such as the Adobe RGB, the sRGB system (defined by HP and Microsoft) and the NTSC RGB color system. Pay attention so as not to confuse these RGB color spaces with the CIERGB color matching functions described in the previous section. CIERGB is a system of color matching functions that can represent any existing colors while a given RGB color space is a color system that uses RGB additive primaries combined in some way to physically reproduce a certain color. Depending on the particular RGB system, different colors can be reproduced according to its gamut.

1.2.2.3 HSL and HSV

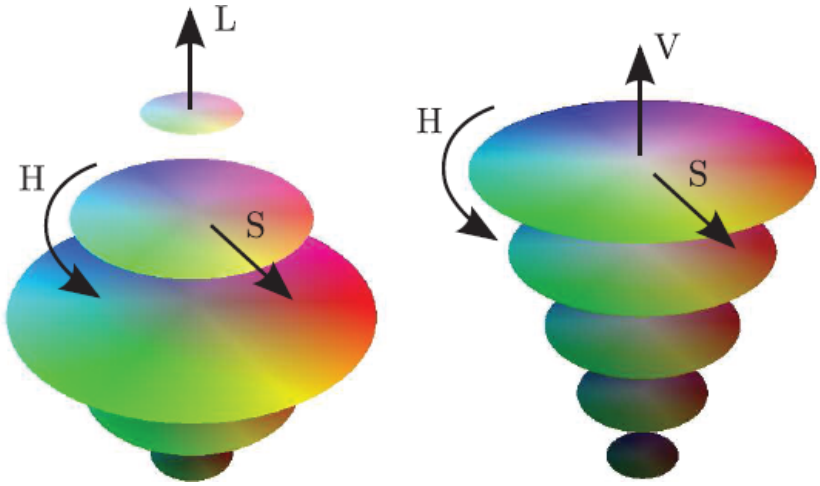
Two color spaces often used in many graphics applications are the *HSL* and the *HSV*. These spaces attempt to use color coordinates that describe colors more intuitively. With HSL a color is described in terms of *hue* (H), *saturation* (S) and *lightness* (L), whereas L is replaced by *value* (V) for the HSV color system. The hue indicates the base chroma of the color to define, the lightness (which is a physical property of the color) is proportional to the brightness of the color (that is how “whitish” we perceive it), and the saturation is somewhat related to the “pureness” of the color to represent.

To better understand what lightness and saturation are, suppose you are a painter and, from your palette, you have chosen a certain hue, say cyan. Then you can choose a shade of gray for the canvas, from black to white, and that is the lightness of the final color. Then you

put the color as small dots on the canvas, and the density of the dots gives you the saturation.

A geometric representation of these two spaces is shown in **Figure 1.5**. As can be seen, the colors of HSL can be naturally mapped on a prism while HSV is mapped on a cone.

Figure 1.5



HSL and HSV color space.

As we can covert coordinates between CIEXYZ and CIERGB color space, it is also possible to convert between RGB and HSL/HSV color space. As an example, we show here how to convert the sRGB color space to the HSL model:

$$H = \begin{cases} \text{undefined} & \text{if } \Delta = 0 \\ 60^\circ \left(\frac{G-B}{\Delta}\right) & \text{if } M = R \\ 60^\circ \left(\frac{B-R}{\Delta}\right) + 120^\circ & \text{if } M = G \\ 60^\circ \left(\frac{R-G}{\Delta}\right) + 240^\circ & \text{if } M = B \end{cases} \quad (1.6)$$

$$L = \frac{M+m}{2} \quad (1.7)$$

$$S = \begin{cases} 0 & \text{if } \Delta = 0 \\ \frac{\Delta}{1-|2L-1|} & \text{otherwise} \end{cases} \quad (1.8)$$

where $M = \max\{R, G, B\}$, $m = \min\{R, G, B\}$ and $\Delta = M - m$. The degrees in the H formula come from the definition of hue on a hexagonal

shape (more details about this can be found here [37]). The HSV conversion is close to the HSL one; the hue is calculated in the same way, while the saturation and the lightness are defined in a slightly different way:

$$S = \begin{cases} 0 & \text{if } \Delta = 0 \\ \frac{\Delta}{V} & \text{otherwise} \end{cases} \quad (1.9)$$

$$V = M \quad (1.10)$$

1.2.2.4 CIELab

CIELab is another color space defined by the CIE in 1976, with very interesting characteristics. The color coordinates of such systems are usually indicated with L^*, a^* and b^* . L^* stands for lightness and a^* and b^* identify the chromaticity of the color. The peculiarity of this color space is that the distance between colors computed as the Euclidean distance:

$$\Delta Lab = \sqrt{\left(L_1^* - L_2^*\right)^2 + \left(a_1^* - a_2^*\right)^2 + \left(b_1^* - b_2^*\right)^2} \quad (1.11)$$

is correlated very well with human perception. In other words, while the distance between two colors in other color spaces cannot be perceived proportionally (for example, distant colors can be perceived as similar colors), in the CIELab color space, near colors are perceived as similar colors and distant colors are perceived as different colors.

The equations to convert a color in CIEXYZ color space to a color in CIELab color space are:

$$\begin{aligned} L^* &= 116\, f(X/X_n) - 16 \\ a^* &= 500\, f(X/X_n) - f(Y/Y_n) \\ b^* &= 200\, f(Y/Y_n) - f(Z/Z_n) \end{aligned} \quad (1.12)$$

where X_n, Y_n, Z_n are normalization factors dependent on the *illuminant* (see next section) and f(.) is the following function:

$$f(x) = \begin{cases} x^{1/3} & \text{if } x > 0.008856 \\ 7.787037x + 0.137931 & \text{otherwise} \end{cases} \quad (1.13)$$

We can see that this formulation is quite complex, reflecting the complex matching between the perception of the stimulus and the color coordinates expressed in the CIEXYZ system.

1.2.3 Illuminant

In the previous section we have seen that the conversion between CIEXYZ and CIELab depends on the particular illuminant assumed. This is always true when we are talking about color conversion between different color spaces.

The different lighting conditions are standardized by the CIE by publishing the spectrum of the light source assumed. These standard lighting conditions are known as *standard illuminants*. For example, Illuminant A corresponds to an average incandescent light, Illuminant B corresponds to the direct sunlight, and so on. The color tristimulus values associated with an illuminant are called *white point*, that is, the chromaticity coordinates of how a white object appears under this light source. For example, the conversion (1.4) between CIERGB and CIEXYZ is defined such that the white point of the CIERGB is $x = y = z = 1/3$ (Illuminant E, equal energy illuminant).

This is the reason why color conversion between different color spaces needs to take into account the illuminant assumed in their definition. It is also possible to convert between different illuminants. For an extensive list of formulas to convert between different color spaces under different illumination conditions, you can take a look at the excellent Web site of Bruce Lindbloom on **http://www.brucelindbloom.com/**, full of useful information, color spaces and related conversions.

1.2.4 Gamma

Concerning the display of colors, once you have chosen a color system to represent the colors, the values of the primaries have to be converted to electrical signals according to the specific display characteristics in order to reproduce the color.

In a CRT monitor, typically, the RGB intensity of the phosphors is a nonlinear function of the voltage applied. More precisely, the power law for a CRT monitor is the applied voltage raised to 2.5. More generally,

$$I = V^{\gamma} \quad (1.14)$$

where V is the voltage in Volts and I is the light intensity. This equation is also valid for other type of displays. The numerical value of the exponent is known as *gamma*.

This concept is important, because this nonlinearity must be taken into account when we want our display to reproduce a certain color with high fidelity; this is what is intended in *gamma correction*.

1.2.5 Image Representation

Images are fundamental to computer graphics systems, both from a hardware and a software point of view. For this reason it is particularly important to understand how an image is represented, which types of images exist, how the graphics hardware handles them, and so on. Here, we provide some basic knowledge about images and their representation. Basically, two ways to define a 2D image exist: *vector image* and *raster image*.

1.2.5.1 Vector Images

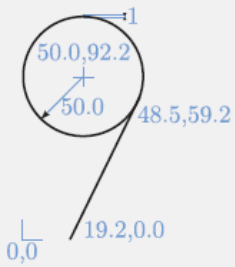
Vector graphics is a way of representing images as a set of basic drawing primitives. Hence, a *vector image* is an image defined by compos-

ing a set of points, lines, curves, rectangles, stars and other shapes. **Figure 1.6** shows a simple example of how the drawing of number 9 could be obtained by combining a circle and a segment.

Figure 1.6

```
NUMBER_OF_PRIMITIVES 2
CIRCLE
center 50.0,92.2
radius 50.0
fill_color white
line_color black
line_thickness 1pt

SEGMENT
FIRST_ENDPOINT 19.2,0.0
SECOND_ENDPOINT 48.5,59.2
line_color black
line_thickness 1pt
```



Example of specification of a vector image (left) and the corresponding drawing (right).

The most well-known formats for vector graphics are the SVG and the PS format.

SVG stands for *scalable vector graphics* and it is a family of specifications of an XML-based file format that also supports dynamic content, that is, animated vector graphics. The SVG specification was developed by the World Wide Web Consortium (W3C), which manages many standards for the Web. In fact, all major modern Web browsers, including Mozilla Firefox, Internet Explorer, Google Chrome, Opera and Safari, have at least some degree of direct support and rendering of SVG markup. SVG supports several primitives for the definition of the image, for example paths (as curved or straight lines), basic shapes (as open or closed polylines, rectangles, circles and ellipses), text encoded in unicode, colors, different gradients/pattern to fill shapes, and so on.

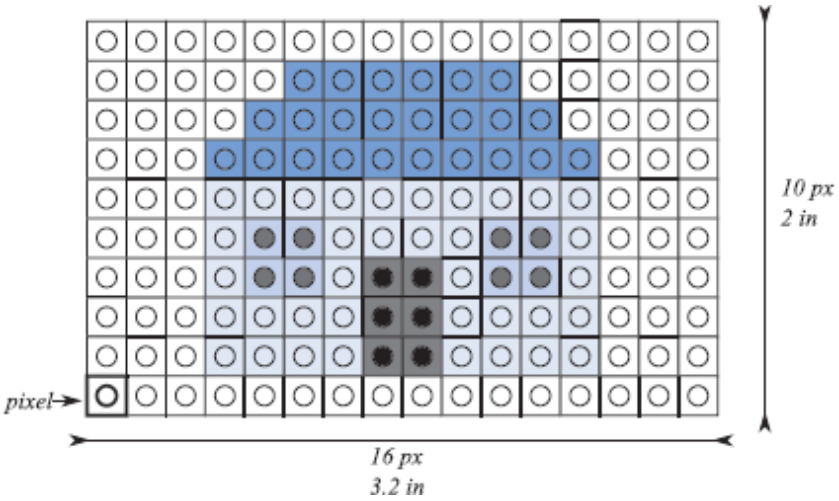
PS stands for *PostScript* and it is a programming language for printing illustrations and text. It was originally defined by John Warnock and Charles Geschke in 1982. In 1984 Adobe released the first laser printer driven by the first version of PostScript language and nowa-

days Adobe PostScript 3 is the de-facto worldwide standard for printing and imaging systems. Since it has been designed to handle the high quality printing of pages it is more than a language to define vector graphics; for example, it allows us to control the ink used to generate the drawing.

1.2.5.2 Raster Images

A *raster image* is, for example, the one you see on the screen of your PC and it is made by a rectangular arrangement of regularly placed small colored tiles, named *pixels* (*pict*ure *el*ements). You can make your own raster image by assembling together some squared Lego® pieces of uniform color and size to form a certain picture, as shown in **Figure 1.7**. The *size in pixels* of an image is the number of pixels along its width and height. In the Lego example it is 16 × 10. The size in pixels of an image is often used interchangeably with its *pixel resolution*, or just *resolution*, but there is a quite important difference to underline. Resolution deals with how small are the details an image can represent and it is commonly quantified to how *close* two lines can be on the image without appearing as a single line. The unit of measure of resolution is *pixels per inch*, that is, how many pixels are in one inch. In the example of **Figure 1.7** the resolution is 16/3.2 = 5*px*/*in*.

Figure 1.7



The image of a house assembled using Lego® pieces.

This means that the same image may have different resolutions depending on which media is used to reproduce it. For example, if the size in pixels of both your mobile phone and your TV is 1280 × 800 it does not mean they have the same resolution because the mobile has the same number of pixels in a much smaller space. However, sometimes the size in pixels is used to indicate the resolution. This is because media are made to be observed at a certain distance. The mobile phone is made to be seen from 20 cm away, while the TV screen is usually seen at a couple of meters. In other words, they occupy a similar portion of our field of view and this is the reason why it makes sense just to use the number of pixels to indicate the resolution.

A pixel can be defined by a scalar or a vector of values, depending on the nature of the image. For example, in a *grayscale image*, each pixel is a scalar value representing the brightness of the image in that position (see **Figure 1.8**). In a color image, instead, each element is represented by a vector of multiple scalar components (typically three) that identifies the color of that image's location.

Figure 1.8

A grayscale image. (Left) The whole picture with a highlighted area whose detail representation (Right) is shown as a matrix of values.

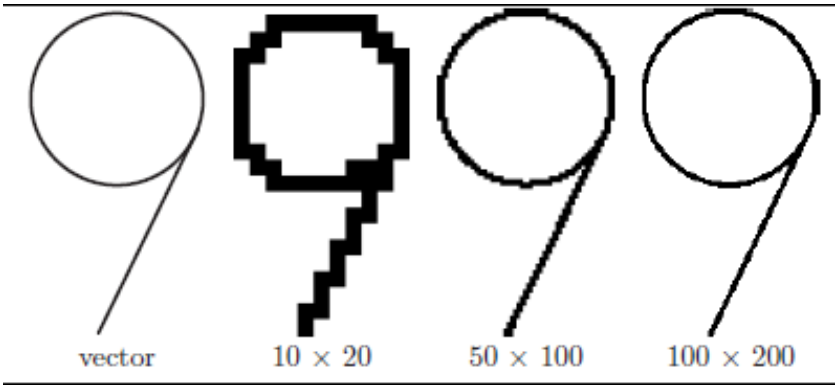
The length of the vector used to define a pixel defines the number of *image channels* of the image. A color image represented by the RGB color space has three channels; the red channel, the green channel and the blue channel. This is the most common representation of a *color image*. An image can have more than three components; additional components are used to represent additional information, or for certain types of images like multi-spectral images where the color representation requires more than three values to represent multiple wavelength bands. One of the main uses of four-channel images is to handle *transparency*. The transparency channel is usually called *alpha channel*. See **Figure 1.9** for an example of an image with a transparent background.

Figure 1.9

(Left) Original image with opaque background. (Middle) Background color made transparent by setting alpha to zero (transparency is indicated by the dark gray-light gray squares pattern). (Right) A composition of the transparent image with an image of a brick wall.

In the comparison of raster images with vector images, the resolution plays an important role. The vector images may be considered to have infinite resolution. In fact, a vector image can be enlarged simply by applying a scale factor to it, without compromising the quality of what it depicts. For example, it is possible to make a print of huge size without compromising its final reproduction quality. Instead, the quality of a raster image depends heavily on its resolution: as shown in **Figure 1.10**, a high resolution is required to draw smooth curves well, which is natural for a vector image. Additionally, if the resolution is insufficient, the pixels become visible (again see **Figure 1.10**). This visual effect is called *pixellation*. On the other hand, a vector image has severe limitations to describe a complex image like a natural scene. In this and similar cases, too many primitives of very small granularity are required for a good representation and hence a raster image is a more natural representation of this type of image. This is the reason why vector images are usually employed to design logos, trademarks, stylized drawings, diagrams, and other similar things, and not for natural images or images with rich visual content.

Figure 1.10



Vector vs raster images. (Left) A circle and a line assembled to form a "9." (From Left to Right) The corresponding raster images at increased resolution.

1.3 Algorithms to Create a Raster Image from a 3D Scene

When we talk about *rendering algorithm* we mean the method we use to display a 3D scene on the screen. In this section we want to show the two basic *rendering paradigms* that are usually employed to generate synthetic images.

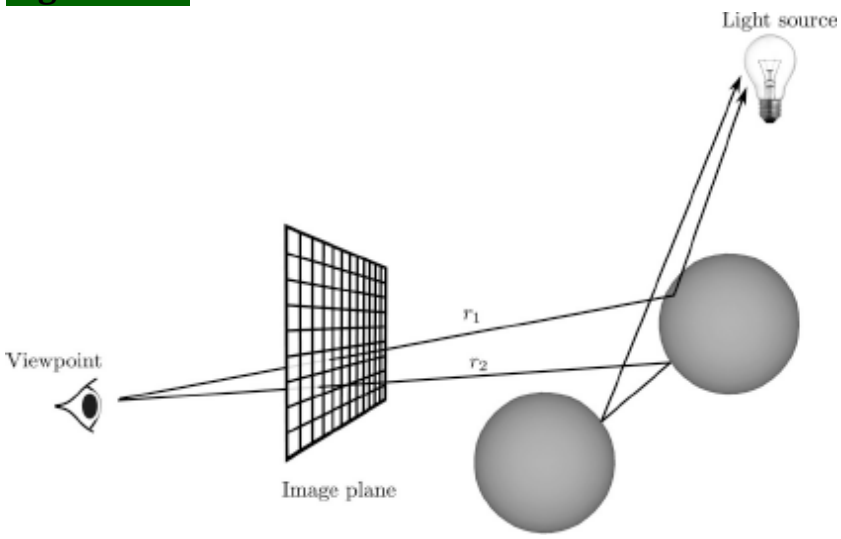
1.3.1 Ray Tracing

As we will see in **Chapter 6**, we see things because light sources emit *photons* : elementary particles of light travel into space, that hit surfaces, change direction, and finally a part of them reaches our eyes. The color we see depends on the properties of the light sources and the materials.

Suppose we simulate this process and compute all the light-matter interactions to trace the path of each single photon until it possibly reaches our eye. It is obvious that most of them will not reach our eyes at all, they will simply get lost in space. So, why bother doing all the costly computation when we are actually interested only in the photons that follow the paths to our eyes? The idea behind *ray tracing*

is to do the reverse process, that is, start from the eye and trace rays into the scene, computing how they interact with the matter to see which ones finally reach a light source. If a ray reaches the light source, it means that a photon will traverse the reverse path from the light to the eye. The idea of ray tracing is illustrated in **Figure 1.11**.

Figure 1.11



A schematic concept of ray tracing algorithm. Rays are shot from the eye through the image plane and intersections with the scene are found. Each time a ray collides with a surface it bounces off the surface and may reach a light source (ray r_1 after one bounce, ray r_2 after two bounces).

In its basic form ray tracing can be described by the following algorithm (**Listing 1.1**):

For each pixel of the image:

1. Construct a ray from the viewpoint through the pixel
2. For each object in the scene
    1. 2.1 Find its intersection with the ray
3. Keep the closest of the intersection points
4. Compute the color at this closest point

LISTING 1.1: Basic ray tracing.

The color of the point is computed by taking into account the properties of the material and the characteristics of the light sources. We learn more about this in **Chapter 6**. Note that it is not required, in this implementation, that the ray must reach a light source. By omitting Step 4 from the above algorithm, what we produce is a *visibility map* of the objects of the scene, that is, a map where for each pixel we have the portion of the 3D scene visible from it. In this case, the algorithm is called *ray casting* and, in practice, it becomes a visibility technique and not a rendering technique, since we simply stop at the first hit of the ray.

The more complete version (see **Listing 1.2**) is what is normally intended as the *classic ray tracing* algorithm:

For each pixel of the image:

1. Construct a ray from the viewpoint through the pixel
2. Repeat until TerminationCondition
    1. 2.1. For each object in the scene
        1. 2.1.1. Find its intersection with the ray
    2. 2.2. Keep the closest of the intersection points
    3. 2.3. Compute the change of direction of the ray
3. Compute the color by taking into account the path of the ray

LISTING 1.2: Classic ray tracing.

The TerminationCondition at Step 2 may include a number of criteria. If we do not have a time limit it could be simply if the ray hits an emitter or exits the scene. Since we cannot run the algorithm forever we need to add termination conditions to decide to drop the quest for an emitter, putting a limit on the number of iterations (that is, the number of time a ray bounces off a surface), or the distance traveled by the ray, or the time spent on computation for the ray. When the tracing is over, that is, the termination condition is reached, we assign a color to the pixel on the base of the surfaces hit by the ray.

The change of direction computed in Step 2.3 depends on the type of material. The basic case is where the surface is considered perfectly *specular* like an ideal mirror and light is *specularly reflected*. When we want to consider more generic materials, we should consider that when a light ray hits a surface it may be *refracted*, which means that the light ray enters the object, as happens for glass, marble, water or skin; or *diffuses*, that is, it is reflected uniformly in many directions. So, the light ray that reaches our eye is the contribution of all these effects: it is the sum of photons emitted by several emitters that have traveled the scene and have changed their direction many times. The consequence of the algorithm is that each time a ray hits the surface one or more rays may be generated depending on the nature of the material. In this case one ray can generate several paths.

We will learn more details about reflection and refraction, and in general about light-matter interaction in the Lighting and Shading chapter (**Chapter 6**), and more details about classic ray tracing and path tracing in the Global Illumination chapter (**Chapter 11**).

The cost of ray tracing can be estimated as follows: for each ray we need to test its intersection with all the m objects, at most for k bounces (where k is the maximum number of bounces allowed) and for each of the n_r rays shot. In general, given n_p as the total number of pixels of the screen, $n_r \geq n_p$ because we want *at least* one ray per pixel. So we have:

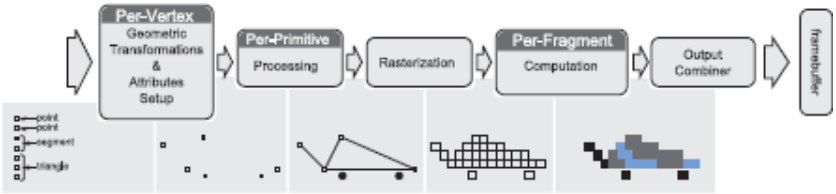
$$cost\,(ray\ tracing) = n_p\ k \sum_{i=0}^{m} Int\,(o_i) \qquad (1.15)$$

where $Int(o_i)$ is the cost of testing the intersection of a ray with the object o_i. However, it is possible to adopt acceleration data structures that mostly reduce $\sum_{i=0}^{m} Int(o_i)$ to $O(log(m))$ operations.

1.3.2 Rasterization-Based Pipeline

The *rasterization-based* rendering pipeline is the most widely used method to render images at an interactive rate and it is ubiquitously used on all the graphics boards, although the implementations change among different hardware, developers and over time. Here, we describe a logical abstract description of the rendering pipeline that we will refer to through the rest of the book and which is illustrated in **Figure 1.12**.

Figure 1.12



Logical scheme of the rasterization-based pipeline.

The figure shows the sequence of operations that turns the specification of a set of geometric primitives into an image on the screen. We will use a modified version of this scheme throughout the book when necessary for specifying further details. You may find many similar schemes in other texts or on the Web, often specialized to some specific API and/or hardware. We will see one of these in depth in the next chapter.

The *input* to the rendering pipeline is a series of geometric primitives: *points, segments* (which are called *lines* in all the APIs), *triangles* and *polygons*.

All the geometric primitives are defined by vertices (one for the point, two for the segment and so on). When we specify a *vertex*, we will usually provide its coordinates, but there are many other attributes that we may want to use. For example we may associate a color value to the vertex, or a vector indicating the speed, or in general any value that makes sense for our application. The first stage of

the pipeline, *per-vertex transformations and attributes setup*, processes all the vertices and transforms their values in a user-specified way. Usually in this stage we decide from where to look at our scene by applying linear transformations (rotations, translations and scaling), but we can also displace all the vertices along a direction by a time dependent value, thus making the car in the example to move.

The next stage is the *primitive processing*, which takes the transformed vertices and the primitive specified by the user and outputs points, segments and triangles to the next stage. The role of this stage may seem minor, because it looks just like it is passing the user input to the next stage. In older schemes of the rasterization-based pipeline you may actually find it collapsed with the *rasterization* stage and/or under the name of *primitive assembly*. Nowadays, this stage not only passes the input to the rasterization stage, but it may also create new primitives from the given input; for example it may take one triangle and output many triangles obtained by subdividing the original one.

The *rasterization* stage converts points, lines and triangles to their raster representation and interpolates the value of the vertex attributes of the primitive being rasterized. The rasterization stage marks the passage from a world made by points, lines and polygons in 3D space to a 2D world made of pixels. However, while a pixel is defined by its coordinates in the image (or in the screen) and its color, the pixels produced by the rasterization may also contain a number of interpolated values other than the color. These “more informative” pixels are called *fragments* and are the input to the next stage, the *per-fragment computation*. Each fragment is processed by the per-fragment computation stage that calculates the final values of the fragment’s attributes. Finally, the last stage of the pipeline determines how each fragment is combined with the current value stored in the *framebuffer*, that is, the data buffer that stores the image during its formation, to determine the color of the corresponding pixel. This

combination can be a blending of the two colors, the choice of one over the other or simply the elimination of the fragment.

The cost for the rasterization pipeline is given by processing (that is, transforming) all the vertices n_v and rasterizing all the geometric primitives:

$$cost\,(rasterization) = K_{tr}\, n_v + \sum_{i=0}^{m} Ras\,(p_i) \qquad (1.16)$$

where K_{tr} is the cost of transforming one vertex and $Ras(p_i)$ is the cost of rasterizing a primitive.

1.3.3 Ray Tracing vs Rasterization-Based Pipeline

Which is the better rendering paradigm, ray tracing or rasterization, is a long-running debate that will not end with this section. Here, we want only to highlight advantages and disadvantages of both paradigms.

1.3.3.1 Ray Tracing Is Better

Ray tracing is designed to consider global effects of the illumination, because it follows the bounces of each ray reaching the eye. So, while transparencies, shadows and refractions are “naturally” included in ray tracing, each of them requires an amount of tricks to be done with rasterization, and often the combination of several global effects is hard to achieve. Note that the version of a ray tracer that produces the same result as the rasterization is obtained by limiting the number of bounces to 1, that is, if we do only ray casting.

With ray tracing we may use any kind of surface representation, provided that we are able to test the intersection with a ray. With the rasterization pipeline, every surface is ultimately discretized with a number of geometric primitives and discretization brings approximation. In a practical simple example we may consider a sphere. With

ray tracing we have a precise intersection (up to the numerical precision of machine finite arithmetic) of the view ray with an analytic description of the sphere, with the rasterization the sphere approximated with a number of polygons.

Ray tracing is simple to implement, although hard to make efficient, while even the most basic rasterization pipeline requires several algorithms to be implemented.

1.3.3.2 Rasterization Is Better

Although it is the common assumption, it would be unfair to state that rasterization is faster than ray tracing. However, rasterization has a linear and more predictable rendering time because each polygon is processed in the same way and independently of the others. At most, the rasterization time depends on how big each polygon is on screen, but modern hardware is highly parallel and optimized for this task. Ray tracing may be implemented so that the time is logarithmic with the number of primitives (so even less than linear) but we need to know all the elements of the scene in advance in order to build acceleration data structures that allow us to compute ray-primitive intersection in a fast manner. This also means that if the scene is *dynamic*, that is, if the elements in the scene move, these data structures have to be updated.

Rasterization naturally handles *antialiasing*. This is a concept that we will see in detail in **Section 5.3.3**, but we can give a basic idea by saying that oblique lines will look *jagged* on the screen since pixels discretize it. With rasterization, this problem can be leveraged by modulating the color of the pixels adjacent to the line segments. In ray tracing, instead, we have to shoot multiple rays for a single pixel to cope with aliasing.

Historically the graphics hardware has been developed for the rasterization pipeline, because of its linearity and streamable nature.

However, it should be said that current graphics hardware is way more general than rasterization-friendly graphics accelerators. They may be considered more like highly parallel, very fast processors with some limitations on memory access and management, therefore also amenable for a ray tracer implementation.