

---

# Computer Science (Focus Game Technology) Basic Computer Graphics

## Wolfgang Hürst

Universitair docent

Department of Information and Computing Sciences  
Faculty of Science, Utrecht University



Universiteit Utrecht

---

**This document contains a tutorial for potential students participating in the matching days at the Department of Information and Computing Sciences at Utrecht University who are interested in Computer Science studies with a focus on Gaming Technology. It should be read and worked through in order to prepare for the actual event at the university.**

## Preface

This short tutorial about some of the basics in computer graphics is used as part of the “matching events” that are organized by the University of Utrecht in order to give prospective students an insight into what to expect from their studies and make sure they chose the right field of study – in this case computer science with focus on game technologies. The purpose of this is to give you some insight into the actual topic (e.g. by providing examples of what you will be studying), into approaches and methodologies commonly used in this research field (e.g. how do computer scientists approach and solve typical problems in their field), and in the used terminology (i.e. how computer scientists “think and talk”).

Considering the latter, most computer scientists often speak English. Although there have been some really great contributions to the field of computer science by Dutch researchers, the area as a whole is dominated mostly by international communities (esp. from the US), which is why during your studies, you will read a lot of technical literature that is written in English. Also, some of the courses in the Bachelor program are taught in English as well, because they are also followed by students from our master programs, especially the master Game & Media Technology, which is an international master and thus taught completely in

English. Finally, our university does not only have the ambition to attract and educate the best students but also to recruit internationally recognized faculty members from all over the world. So, because you will be faced with the need to read, understand, and talk English in various situations throughout your studies, we will start with this right now, and present the following tutorial in English.

The text provides an initial introduction into some of the theoretical basics of computer graphics. It should be suited for prospective students with different levels of experience. Hence, if you already had computer science in school or did a lot of programming for yourself, you might not feel challenged enough. Don't worry: during your studies, you will be facing way more ambitious material. Likewise, if you have no real background knowledge in computer science so far, you might feel lost and overwhelmed at times. Again, there is no need to worry. On the matching day at the university, there will be a lecture and exercise session where you are given further insight and background information. In both cases, we will make sure during the personal sessions of the matching day, that you will pick the right subject that is neither too challenging nor too trivial for you.

Enjoy the tutorial, and we are looking forward to welcoming you at the matching day at Utrecht University soon.

## **Content**

1. Introduction: What is computer science?
2. Some basic mathematics for graphics
  - 2.1 Raster displays, pixels, and how to represent them
  - 2.2 Vector calculations
  - 2.3 Modeling and basic geometric entities
  - 2.4 Matrices and vector transformations
3. Rendering and rasterization
  - 3.1 Line drawing algorithms
  - 3.2 The midpoint algorithm
4. Conclusion

**Computer science** or **computing science** is the scientific and practical approach to computation and its applications. A computer scientist specializes in the theory of computation and the design of computational systems.

(from Wikipedia.org)

## 1. Introduction: What is computer science?

You are reading this document because you are interested in studying Computer Science with a focus on Game Technologies at Utrecht University. But what is computer science concretely? And what does it mean to focus on game technologies?

As the name implies, **computer science (CS)** is a scientific study. In particular, it is a science that studies computable processes and structures. It describes the scientific and practical approach to computation and its applications. As such, it covers a very broad range of subfields that can roughly be classified as either *theoretical* or *practical computer science* (the latter is sometimes also called *applied computer science*). More theoretical approaches generally explore fundamental questions related to computation, such as: What kind of problems can we solve with computers (and what cannot be solved)? How can we prove that a program is correct, i.e. that it does what it is supposed to do? Can we automatically verify that a program will not crash? More practical approaches to computer science include artificial intelligence (e.g. how can we make machines think?), computer graphics and visualization (e.g. how can we create a tridimensional virtual world on a computer?), human computer interaction (e.g. how can we make computers more accessible and useful in everyday situations?), and many more. Notice that the distinction between theoretical and applied computer science is not always strict. Many fundamental problems that we study in theoretical computer science are motivated by practical needs. Likewise, many things that we study as part of applied computer science require a strong theoretical background. For example, to produce great computer graphics, you

need to have a good understanding of mathematics, as we will see later on in this tutorial.

A particularly interesting area of applied computing is, of course, gaming and related technologies. In particular, the term **gaming technologies** summarizes all basic techniques, approaches, methods, and tools that we need to create great computer games, simulations, and virtual worlds. Again, this includes various subfields of computer science, such as:

- Artificial Intelligence, that studies the design and analysis of intelligent systems that support people to make better decisions (think of, for example, a more “intelligent” virtual game character);
- Interaction Technology, that studies the interaction between various media, virtual worlds, users, and their environment (think of, for example, innovative game play interactions such as motion and full body tracking);
- Software Systems, that studies the technologies to describe, construct, and adapt software systems, and their architecture, engineering, and deployment (think of, for example, improved infrastructure for real-time support of thousands of online players from all over the world);
- Virtual Worlds, that studies the modeling, visualization, animation, and simulation of virtual worlds and the characters that inhabit them (think of, for example, game avatars that move more realistically than they do in current games).

**Computer graphics** are graphics created using computers and, more generally, the representation and manipulation of image data by a computer with help from specialized software and hardware.

(from Wikipedia.org)



**Figure 1.** Screenshot from the 3D animation movie "Big Buck Bunny", <http://www.bigbuckbunny.org/>

Notice that these are also the four main research clusters of the Department of Information and Computing Sciences at Utrecht University. Of course, further subfields exist that have a high relevance for gaming and game development as well. In the remainder of this tutorial, we want to have a closer look at one example: Computer Graphics, which is of course strongly related to the area of Virtual Worlds, but has also relations to, for example, Interaction Technologies, because graphics are often an integral part of man-machine communication.

In general, **computer graphics** describes all kinds of techniques and tools used to create digital imagery with a computer. We distinguish between 2D graphics, which, well, illustrate 2D images, and 3D graphics, which give us fully tridimensional imagery and virtual worlds (cf. Fig. 1). At Utrecht University, computer graphics is taught as part of the bachelor program at the end of your first year. It is a good example for what to expect in your studies and what will be required, because first, it has a clear relation to game technologies, as many games use 3D graphics. Second, it illustrates why and how mathematic skills are required as part of our studies. We will discuss some basic mathematics in **Section 2**. Third, it is a good example for algorithms and data structures, two important concepts in computer science. Don't worry if you don't know what these are yet. We will talk about it in **Section 3**. Finally, this subject is also well suited to illustrate not only technical knowledge that you will gain during your studies, but also what kind of other skills are commonly required when studying computer science. These include strategic-thinking skills, good problem solving skills, curiosity and creativity. The practical **exercises** throughout the document should

illustrate this and provide some concrete examples. Notice that other skills are necessary, too, especially good communication and the ability to work in a team, because today hardly any game (or software in general) is developed by individuals in pure isolation.

## Exercises

Because this section was a rather informal introduction into the topic, there are also only just a few rather informal exercises for it. Can you answer the following questions?

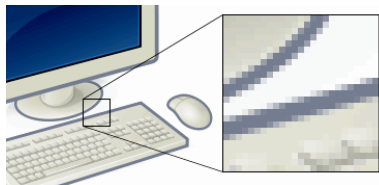
- Q1: Computer science is a science that studies what?
- Q2: What is the distinction between theoretical and practical computer science?
- Q3: Can you think about some typical issues that are studied in practical computer science?

## 2. Some basic mathematics for graphics

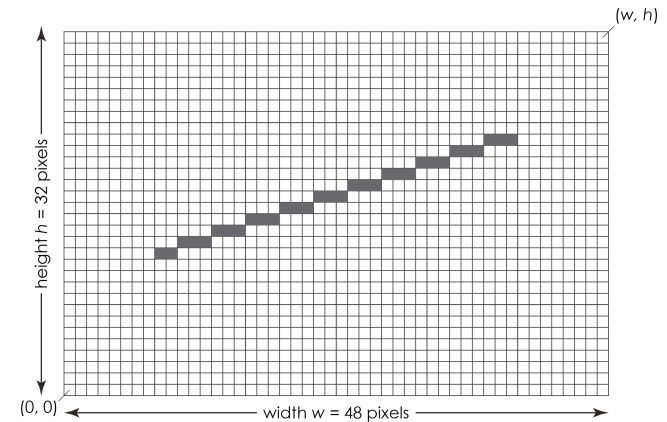
Everyone knows computer graphics. When you turn on your computer or even your cellphone, you usually operate it with a graphical user interface. Graphics are used in computer games and movies (both purely animated ones like *Toy Story* as well as for special effects in regular movies). As said before, we can distinguish between 2D and 3D graphics. In this tutorial, and in the related first year course in the bachelor program, we are mostly interested in 3D graphics representing, for example, whole tridimensional virtual worlds and characters within those worlds. Yet, the screen on which we are showing the 3D worlds is commonly a flat 2D screen. Two important questions that we have to deal with in computer graphics are therefore: How do we represent virtual 3D worlds in a way computers can process? And how do we bring these representations of 3D worlds onto a 2D screen in a way that makes it look realistic? Let's start by looking at the first issue.

### 2.1 Raster displays, pixels, and how to represent them

Most monitors and computer screens used today are so called *raster displays*, i.e. basically a 2D grid of light emitting "dots". In graphics, we call these dots *pixels*. Common screens have a lot of those pixels at a high density (e.g. DVDs have a resolution of  $720 \times 576$ , Blu Rays have up to  $1920 \times 1080$ , and computer screens might have even higher ones). Because of this, the human eye cannot recognize each individual pixel but our brain automatically combines them to a large image (cf. Fig. 2). The simplified image in Figure 3 illustrates how a line segment can be drawn on a raster screen by just "painting" single dots.



**Figure 2.** An example illustrating raster graphics: an image is made up from single "dots" called pixels (right) that are so small and dense that the human eye cannot distinguish them but sees them as a whole picture (left).



**Figure 3.** Simplified illustration of a line that is drawn on a small raster display by coloring individual pixels.

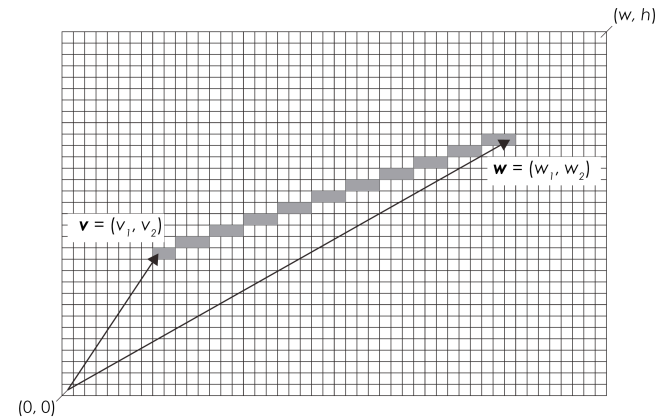
But how can we address the individual pixels on the screen, for example, if we want to tell our program to draw a particular one in black? This is where mathematics comes into play. In particular, we are using the concept of *vectors* and *vector spaces*. Technically, a **vector** is simply defined as an ordered  $n$ -tuple of single values, i.e.

$$\mathbf{v} = (v_1, v_2, v_3, \dots, v_n).$$

As you see above, we are denoting vectors with bold face letters in order to distinguish them from single values, which are called *scalars*. The number  $n$  of these scalars that make up a vector is called its *dimension*. You can imagine that in graphics, we are mostly dealing with 2D and 3D data. (Sometimes, we also need four values to do some processing, but we will not cover that in this tutorial.) For  $n = 2$  and  $n = 3$ , we call the 2-tuples and 3-tuples *pairs* and *triples*, respectively, that

create a 2D and 3D vector space. Roughly speaking, a *vector space* is an abstract concept that describes vectors of a particular dimension and defines operations with them, such as addition, subtraction, multiplication, etc. We will talk about some of them later.

How can we use this abstract mathematical concept of vectors and vector spaces to create, describe, and manipulate 2D and 3D graphics? Let's have a look at our 2D raster display from Figure 3 again. Here, we have a grid that we can describe by its width  $w$  ( $w = 48$  in the image) and height  $h$  ( $h = 32$  in the image). If we count the pixels by row and column starting, for example, from bottom left with  $(0, 0)$  to top right where we have pixel  $(w - 1, h - 1)$ , we can uniquely address each individual pixel on the screen. As you see, we usually describe this as a pair, with the first value indicating the column, and the second one indicating the row of the grid. So, we have an ordered pair of scalar values, which, according to our definition above, can be interpreted as a vector in 2D. Using a reference point, here the value  $(0, 0)$ , we can therefore describe all pixels on the screen with vectors. Even more, we can use vectors to describe more complex geometric shapes. For example, Figure 4 illustrates how we can describe the line segment from Figure 3 by just two vectors representing the end points of the segment. Thus, if we want to draw our line segment on the screen, we do not have to store all points that make up the segment, but only the two end points – and then draw it by coloring all pixels in between these two endpoints. We will learn a procedure for doing this “drawing in between” in Section 3. For now, let's focus on the description of points by vectors.

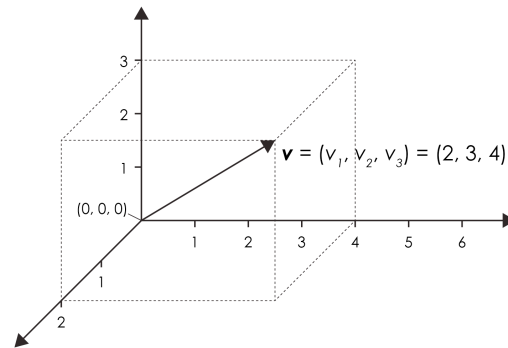


**Figure 4.** Illustration of how the line segment from Figure 3 can be described by two vectors,  $\mathbf{v}$  and  $\mathbf{w}$ , that represent the two endpoints of the segment.

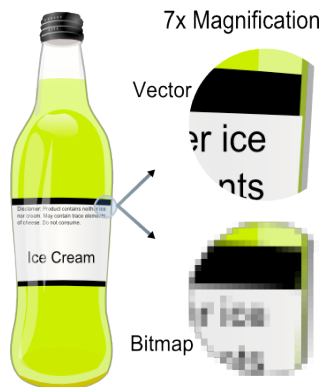
We saw that we can describe points on the screen by vectors with respect to a reference point – here  $(0, 0)$ , the lower left side of our raster display. We call this the *origin* of our vector space. The two dimensions are commonly referred to as *x-value* and *y-value*. Horizontal and vertical operations in the grid are described as operations along the *x-axis* and *y-axis*, respectively. As a consequence, vectors in 2D are often denoted by  $\mathbf{v} = (v_1, v_2)$ ,  $\mathbf{v} = (v_x, v_y)$ , or  $\mathbf{v} = (x, y)$ , and the related scalar values are called *coordinates* of the vector (or *x-* and *y-coordinate* if they are addressed individually).

Now we have a mathematical framework to describe and address our pixels on the screen. But how do we describe 3D data? You probably guessed it: we just add a third coordinate to our vector and use a tridimensional vector space (cf. Figure 5). The third

coordinate is usually referred to as *z-coordinate* and our 3D vectors are denoted by  $\mathbf{v} = (v_1, v_2, v_3)$ ,  $\mathbf{v} = (v_x, v_y, v_z)$ , or  $\mathbf{v} = (x, y, z)$ .



**Figure 5.** Example of a tridimensional vector in a 3D vector space. Notice that the space is not limited like our 2D screen but infinite (of course, 2D spaces are infinite, too, but the screen only represents a finite subset of it). The dotted cube is drawn for better illustration – after all, like our computer screen, paper is only two-dimensional, too, so visualizing 3D data can be kind of difficult.



**Figure 6.** Illustration of the difference between a so-called bitmap, which stores the color of individual pixels in a raster image, and a vector graphic, which stores an abstract representation that can, for example, easily be scaled to different resolutions.

You might wonder why we are using such a complete mathematical framework when it seems so much easier to just describe the individual pixels that make up, for example, the line shown in Figure 3. First, keep in mind that this is a very simple example to illustrate the concept, but we are of course dealing with much more complex visualizations in graphics. Second, if we store a graphic in vector form instead of individual pixels, we are much more flexible, for example, when rescaling it in order to draw our graphic on displays with different resolutions or when resizing the window that shows our image (cf. Figure 6). Finally, using vectors to describe graphics allows us to easily manipulate them using vector calculations – which we will address in the next subsection.

## Exercises

So far, we have only laid out the framework that we will be using in the rest of the tutorial to do some processing and create graphics. Hence, there is not much to exercise yet. However, the end of this section is a good opportunity to reflect on the various things that we learned. For example: can you answer the following questions?

- Q1: What is the formal definition of a vector in 3D?
- Q2: Assume the vector  $\mathbf{v} = (v_1, v_2, v_3, v_4)$ . What is the dimension of this vector?
- Q3: How do we call the  $v_i$  ( $i = 1, \dots, 4$ ) from the vector in Q2?
- Q4: What is the difference between a 2D vector and a scalar value?

## 2.2 Vector calculations

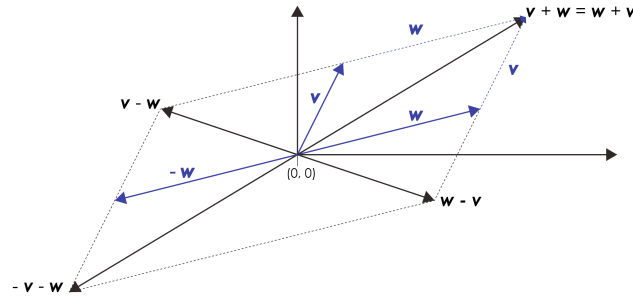
Vectors are not only being used to describe graphics, but also, for example, to animate them, to manipulate images, and to create new from existing ones. One of the simplest operations to create new vectors is by adding two existing ones. The **sum** of two 3D vectors  $\mathbf{v} = (v_1, v_2, v_3)$  and  $\mathbf{w} = (w_1, w_2, w_3)$  is defined as:

$$\mathbf{v} + \mathbf{w} = (v_1 + w_1, v_2 + w_2, v_3 + w_3).$$

Likewise, we can define **subtraction** of two vectors in 3D by:

$$\mathbf{v} - \mathbf{w} = (v_1 - w_1, v_2 - w_2, v_3 - w_3).$$

We see that in both cases the result is again a vector in 3D (or 2D, of course, for addition and subtraction in 2D, which are both defined accordingly). But how does this vector look like? Figure 7 shows an example in 2D.



**Figure 7.** Geometrically, vectors are added by appending the end of the second vector to the beginning of the first one. Subtraction can be seen as addition with a “negative version” of a vector, i.e. a vector pointing in the opposite direction.

From the image, we also see that

$$\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}.$$

We say that vector addition is *commutative*. However, we also see that  $\mathbf{v} - \mathbf{w}$  is not the same as  $\mathbf{w} - \mathbf{v}$ . The image illustrates that the resulting vector is created by appending a “negative version” of vector  $\mathbf{w}$  to vector  $\mathbf{v}$ . Algebraically, we could also write this as:

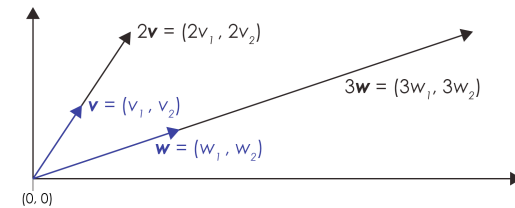
$$\begin{aligned}\mathbf{v} - \mathbf{w} &= (v_1, v_2, v_3) - (w_1, w_2, w_3) \\ &= (v_1, v_2, v_3) + (-w_1, -w_2, -w_3).\end{aligned}$$

So, we multiplied each coordinate of vector  $\mathbf{w}$  with -1 to get the “negative version” of it. Then, we added it to

vector  $\mathbf{v}$  in order to get the new vector  $\mathbf{v} - \mathbf{w}$ . This multiplication of a scalar value (here the value -1) with a vector (here the vector  $\mathbf{w}$ ) can be done with any scalar, and is commonly referred to as **scalar multiplication**. Formally, it is defined as:

$$s \mathbf{v} = (s v_1, s v_2, s v_3)$$

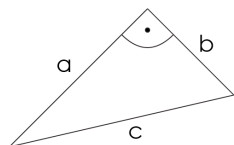
Notice that  $s$  is not printed in bold face, meaning that it is not a vector, but a scalar value that we multiply with the vector, like we multiplied the scalar value -1 with the vector  $\mathbf{w}$  above. We can use this, for example, to modify the size of a vector. (There are other uses for scalar multiplication in graphics, of course, but we are not covering them here.) Figure 8 shows an example in 2D.



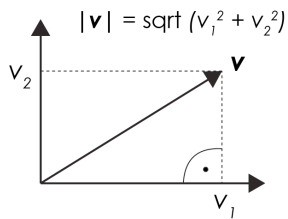
**Figure 8.** Examples for scalar multiplication.

Vectors have certain characteristics that are important when doing calculations with them. We want to look into just a few of them here. For example, sometimes it is necessary to know the **length** of a vector. We can calculate that using the following formula:

$$|\mathbf{v}| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$



Pythagoras:  
 $a^2 + b^2 = c^2$



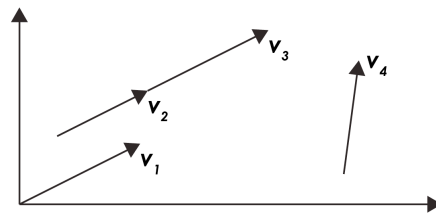
**Figure 9.** The Pythagoras theorem describes the relationship between the lengths of the sides of a triangle with a right angle. If we look at the triangle formed by a vector and its two coordinate values, we can use it to calculate the vector's length.

The symbol  $|\mathbf{v}|$  is commonly used to denote the length of a vector. Notice that this is the general definition of the length of a vector in any dimension. For 2D, this equation might look familiar, because it becomes the same as the Pythagoras theorem for triangles, which you might know from school (cf. Figure 9).

An important characteristic when comparing two vectors with each other is the direction in which they are pointing. For example, in Figure 10, we see that the vectors  $\mathbf{v}_1$ ,  $\mathbf{v}_2$ , and  $\mathbf{v}_3$  seem to point in the same direction, whereas  $\mathbf{v}_4$  does not. We call this characteristic **parallel** and define it the following way:

Two vectors  $\mathbf{v}$  and  $\mathbf{w}$  are parallel if and only if one of them is a scalar multiple of the other, i.e. if there exists a scalar value  $s$  with  $s \neq 0$  such that  $\mathbf{v} = s\mathbf{w}$ .

Notice that according to this definition, vectors pointing in the opposite direction are also considered parallel. For example, the vector  $-\mathbf{w}$  in Figure 7 is parallel to  $\mathbf{w}$ .



**Figure 10.** A simple example of three vectors that are parallel to each other (i.e. "pointing in the same direction") and one that is not.

## Exercises

Now that we learned how to do some calculations with vectors, it is good to get some practice.

Q1: In this subsection, we formally defined vector addition and subtraction in 3D. How can those two operations be formally defined in 2D?

Q2: Given the two 3D vectors  $\mathbf{v} = (1, 2, 3)$  and  $\mathbf{w} = (3, 2, 1)$  and two scalar values  $s = 3$  and  $t = -1$ , calculate the following:

- a)  $\mathbf{v} + \mathbf{w}$     b)  $\mathbf{v} - \mathbf{w}$     c)  $\mathbf{w} + \mathbf{v}$   
 d)  $s\mathbf{v}$     e)  $s\mathbf{v} + t\mathbf{w} - t\mathbf{v} + s\mathbf{w}$

Q3: We said that subtraction of vectors is not commutative. Give an example that proves this statement.

Q4: Calculate the length of the vector  $\mathbf{v} = (1, 2, 2)$ .

Q5: Informally, we can say that two vectors are parallel if they point in the same or exactly opposite directions.

- Give an example for two parallel vectors.
- Give an example for two parallel vectors that point in exactly opposite directions.
- Give an example for two vectors that are *not* parallel. Use the formal definition that we introduced in this section to explain why your answer is correct.

### 2.3 Modeling and basic geometric entities

In the previous subsection, we have seen how vectors can be used to describe points (both in 2D and 3D), and learned about some easy operations to manipulate them – in particular, addition, subtraction, and scalar multiplication. Although the arithmetic that we introduced so far is quite simple, we can already use it to describe some basic geometric objects other than single points on the screen (or in a 3D space). For example, we can represent a line in 2D in the following way:

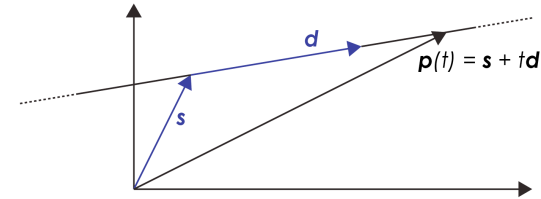
$$\mathbf{p}(t) = \mathbf{s} + t\mathbf{d}$$

with  $\mathbf{s}$  and  $\mathbf{d}$  being two vectors as illustrated in Figure 11, and  $t$  being a variable scalar value. You see that we just used addition of two vectors, i.e. the vectors  $\mathbf{s}$  and  $\mathbf{d}$  to define the location and direction of our line. For this reason, the vector  $\mathbf{s}$  is sometimes called *support vector* of the line, and vector  $\mathbf{d}$  is called its *direction vector*. By multiplying the vector  $\mathbf{d}$  with any scalar value from  $-\infty$  to  $\infty$  (the symbol  $\infty$  denotes infinite values), we can express any point on the line with the vector  $\mathbf{p}(t)$ . But how do we get the direction vector  $\mathbf{d}$  when we have our line just specified by two points, as we indirectly did when drawing the line segment in Figure 4? Well, we just use the other operation that we learned, i.e. subtraction of vectors, and our equation of a line becomes:

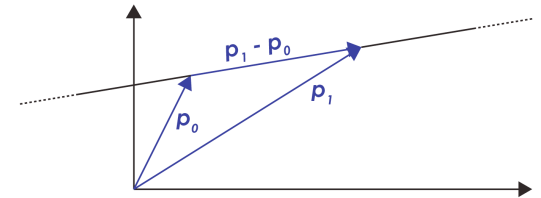
$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0).$$

You see that vector  $\mathbf{p}_0$ , which represents the first of the two points, is used as support vector of our line, and its

direction vector is defined by the difference of the vectors  $\mathbf{p}_1$  and  $\mathbf{p}_0$ , as illustrated in Figure 12.

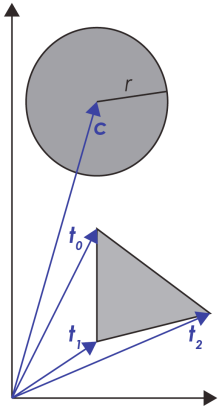


**Figure 11.** Illustration of a line that is defined by two vectors: its support vector  $\mathbf{s}$  and its direction vector  $\mathbf{d}$ .



**Figure 12.** Illustration of a line that is defined by two vectors on the line. The direction vector of this line can be calculated by taking the difference between the vectors of the two points.

Similarly to how we used vectors and the related arithmetic to describe a line, we can define other simple geometric shapes, such as circles, rectangles, triangles, etc. (cf. Figure 13). For example, we can represent a circle by a vector to its center and the circle's radius (which in turn can be described by the length of a vector pointing from this center to any point on the circle's surface). A triangle can be represented by vectors pointing to its three vertices. Four vectors can be used to define a rectangle, and so on.



**Figure 13.** Examples of some basic geometric entities: A triangle defined by the three vectors  $\mathbf{t}_1$ ,  $\mathbf{t}_2$ , and  $\mathbf{t}_3$  representing its vertices, and a circle specified by its center  $\mathbf{c}$  and radius  $r$ .

In graphics, these simple arithmetic shapes are often referred to as **basic geometric entities**, because they are the building blocks that we use to create more complex shapes in a process called **modeling**. Especially triangles are particularly important because they are easy to represent, their manipulation is supported by modern graphics hardware (and thus can be done very fast), and they are powerful enough to allow us to build very complex graphics (cf. Figure 14).

We will not go into further details of modeling as part of this tutorial, but the simple example of how to represent and manipulate lines and line segments with vector arithmetic should illustrate the basic approach of how to describe such complex models in a way that computers can understand and process.

We talked about how to represent the building blocks of such 3D models, i.e. basic geometric entities, when creating a scene or character for a virtual world, and also in 2D when showing the final image on a computer screen. But how do we get this 2D representation of, for example, a nice 3D scene that we created for a computer game? This process is called *perspective projection* because it projects the 3D scene onto a 2D image – our computer screen – by preserving the correct perspective. It is done using matrix multiplication, but covering it will go beyond the scope of this tutorial. Yet, we will address matrices with respect to vector manipulation in the next subsection.

## Exercises

In this subsection we applied the vector space framework to create basic geometric entities, especially

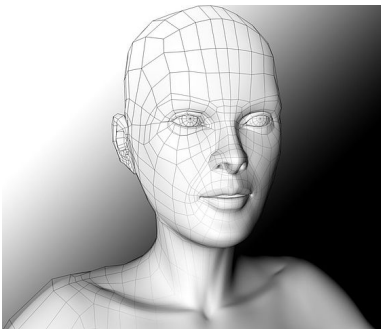
lines. To exercise, let's look at some concrete examples.

- Q1: Given two vectors  $\mathbf{v} = (v_1, v_2, v_3)$  and  $\mathbf{w} = (w_1, w_2, w_3)$ , write down the general equation of a line where  $\mathbf{v}$  is the support vector and  $\mathbf{w}$  is the direction vector.
- Q2: Assume  $\mathbf{v}_1 = (1, 2, 2)$  and  $\mathbf{v}_2 = (4, 3, 3)$  are two points on a line. Write down the equation of that line.
- Q3: Assume that  $\mathbf{c} = (1, 1, 2)$  is the center of a sphere in 3D and  $\mathbf{p} = (2, 5, 6)$  is a point on the sphere's surface. Calculate the radius of that sphere. (Hint: If you don't know how to do this, draw a comparable image of a circle in 2D and read the text in this subsection again carefully.)

## 2.4 Matrices and vector transformations

Matrices are another concept from mathematics that is very important in graphics. Matrices are used, for example, to get the 2D representation of a 3D scene on a computer screen (cf. perspective projection discussed above). Another usage, which we will illustrate here with a few examples, is transformation of vectors and larger objects that are represented by them.

So, what are matrices? Similarly to vectors, they are formally defined as an ordered set of scalar values. However, in contrast to vectors, here the scalars are sorted in a two-dimensional grid. In particular, we say a **matrix** is defined by  $n$  times  $m$  values sorted in  $n$  rows and  $m$  columns, as illustrated here:



**Figure 14.** Example of a more complex 3D model made out of polygons.

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

We call the number  $n$  of rows and the number  $m$  of columns of a matrix its *dimensions*. A matrix of dimensions  $n$  and  $m$  is often referred to as an  $n \times m$  *matrix*. In graphics, we are commonly interested in 2x2 and 3x3 matrices. 4x4 matrices are often used as well, but we won't cover them in this tutorial. The scalar values  $a_{ij}$  that make up the matrix are called its *coefficients*.

As you can see above, matrices are commonly denoted by bold face upper case letters in order to distinguish them from vectors for which we use lower case bold face letters. Yet, according to our definition, a vector can actually be seen as a special kind of matrix, i.e. a 2x1 matrix in 2D and a 3x1 matrix in 3D. We just have to use a different notation, i.e. instead of  $\mathbf{v} = (v_1, v_2, v_3)$ , we denote our vectors by:

$$\mathbf{v} = (v_1, v_2, v_3) = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

In fact, this is just a convention of how to denote vectors, and many books use either of the two options,

or both of them interchangeable (notice the commas in the "horizontal" version). We will do so, too. This will come in handy when doing manipulations of vectors using matrix arithmetic.

In this tutorial, we will just look into one example, i.e. **matrix multiplication**, which is defined by:

$$\mathbf{C} = \mathbf{AB} \text{ with } c_{ij} = \sum_{k=1}^{n_A} a_{ik} b_{kj}. \quad (\text{Equation 2.1})$$

The  $c_{ij}$  represent the coefficients of the new matrix  $\mathbf{C}$  that we create by multiplying the two matrices  $\mathbf{A}$  and  $\mathbf{B}$  with each other. The formula above might become clearer by a concrete example:

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} 6 & 5 & 1 & -3 \\ -2 & 1 & 8 & 4 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 5 & 0 & 2 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 6 & 2 & 2 \\ 37 & 5 & 16 \end{pmatrix}$$

Let's look for example at the coefficient  $c_{22}$  in our new matrix  $\mathbf{C}$  (cf. the red values in the equation above). According to our calculation scheme (equation 2.1), it is calculated by:

$$\begin{aligned} c_{22} &= a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} \\ &= -2*0 + 1*1 + 8*0 + 4*1 \\ &= 5. \end{aligned}$$

You might have already noticed that this definition of matrix multiplication only works if the dimensions of the two matrices fulfill certain conditions. In particular, the second dimension of the first matrix must be the

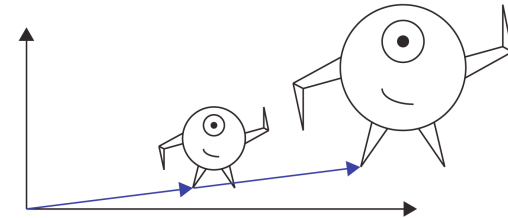
same as the first dimension of the second one. In the example above, we have  $m_A = n_B$  (if  $n_A$ ,  $m_A$  and  $n_B$ ,  $m_B$  denote the dimensions of the matrices **A** and **B**, respectively), so we can multiply them. But because  $n_B \neq m_A$ , the product of **B** and **A** is however not defined. If you don't see it immediately why this is the case, it helps writing it down. If you try to calculate a coefficient of the new matrix, you will see that it doesn't work because 'the numbers don't match'. This should also indicate why in graphics we are mostly interested in 2x2 and 3x3 matrices – because we can multiply them with a vector, which, as we said, is technically nothing else than a 2x1 and 3x1 matrix in 2D and 3D, respectively. But what is the result of such a multiplication? In 2D we get:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{pmatrix}$$

And in 3D we have:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_{21}x + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{pmatrix}$$

We see that a vector multiplied with a matrix give us again a vector of the same dimensions, just with different values. And that's exactly how matrices can be used to manipulate and transform vectors. For example, we can use a matrix to do some scaling of 2D and 3D models by just multiplying all the vectors that make up this model with this scaling matrix, as illustrated for a simplified case in Figure 15.



**Figure 15.** A simple example for scaling in 2D by a factor of 2. If we apply this to all vectors that are used to specify this simple model (as illustrated on with the one pointing to the right foot), we can easily modify the model. Notice that scaling is done with respect to the origin. Scaling around an arbitrary center can be done easily, too, but requires a special step that is not covered in this tutorial.

The following operation scales a vector in 2D by the factor of 2:

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x \\ 2y \end{pmatrix}$$

You might notice that matrix multiplication is not really necessary to achieve this transformation, but we could have easily done this by a simple scalar multiplication of the vector **v** with the scalar  $s = 2$ . That's true indeed, but matrix multiplication allows us to do more powerful scaling operations as well. For example, we can use a different scaling factor for the x- and y-coordinates, as illustrated in the following example, which scales the x-coordinates of an object down by half while at the same time increasing its y-values by 3:

$$\begin{pmatrix} 1/2 & 0 \\ 0 & 3 \end{pmatrix}$$

Figure 16 illustrates this again using our simplified model.

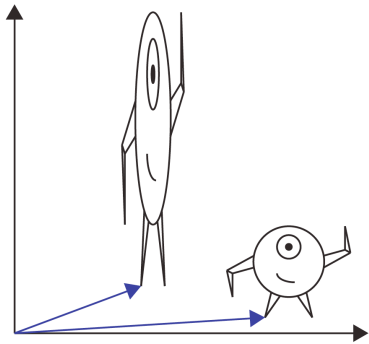
In these two examples, our transformation matrices always had zero values in the top right and bottom left corner. Yet, there is no need to do that. By choosing appropriate values for all matrix coefficients, we can easily create other transformations than scaling. For example, we can make a reflection along the  $y$ -axis using the following matrix:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

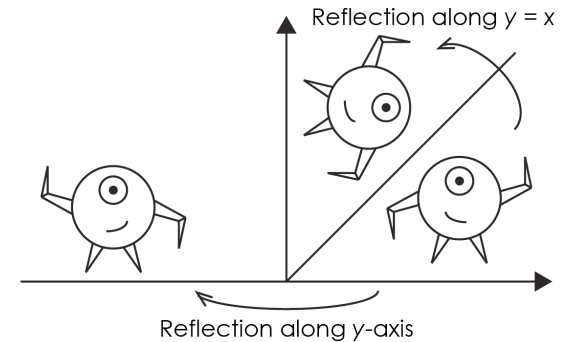
Or, we can reflect or model along the line  $x = y$  using this operation:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Examples for these two reflections are given in Figure 17.



**Figure 16.** A simple example for non-uniform scaling where the model increases in size by 3 in direction of the  $y$ -coordinate, and decreases by  $1/2$  in the direction of the  $x$ -coordinate.



**Figure 17.** A simple example for reflection along the  $y$ -axis and along the line  $y = x$ .

## Exercises

In this subsection we introduced matrices, learned how to multiply them if they fulfill certain conditions, and saw how we can use them to transform vectors.

Q1: Assume a vector  $\mathbf{v} = (1, 3, 1)$  and the following matrix:

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 4 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix}$$

a) Calculate the matrix product  $\mathbf{A} \mathbf{v}$

b) Calculate the matrix product  $\mathbf{v} \mathbf{A}$

(Hint: if you are having problems solving part b), answer question Q2 first ;)

Q2: Assume an  $n_A \times m_A$  matrix **A** and an  $n_B \times m_B$  matrix **B**.

- a) Give the criteria that has to be fulfilled so that the matrix product  $\mathbf{C} = \mathbf{AB}$  is defined.
- b) Give the criteria that has to be fulfilled so that the matrix product  $\mathbf{C} = \mathbf{BA}$  is defined.
- c) Give an example of two matrices **A** and **B** where the product  $\mathbf{AB}$  is defined, but the product  $\mathbf{BA}$  is not.
- d) Give an example for two matrices **A** and **B** where the product  $\mathbf{AB}$  and the product  $\mathbf{BA}$  are both defined.

Q3: We saw how matrix multiplication can be used to transform vectors and looked at a few examples. Now let's think about some other transformations.

- a) Give a matrix that scales a vector in 2D by a factor of 4 in x-direction and by a factor of 2 in y-direction.
- b) Give a matrix that scales a vector in 3D by a factor of 2 in x-direction, by a factor of 3 in y-direction, and by a factor of 4 in z-direction.
- c) Give a matrix that reflects a vector in 2D along the x-axis.
- d) Give a matrix that reflects a vector in 2D along the line  $y = -x$ .

### 3. Rendering and rasterization

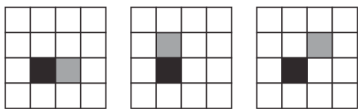
In the last section we learned how to represent and manipulate basic geometric entities using the example of lines and line segments. And we gained an initial understanding of how more complex models in 2D and 3D could be built. Yet, so far, we only talked about the shape of such models, but not their color and texture, how to consider light that is reflected from them, etc. This step of putting the color onto grid models created in the modeling phase is called **shading**. It is what turns the 3D models shown, for example, in Figure 14 into full, nice looking graphics such as the one in Figure 1. In computer graphics, we commonly summarize the perspective projection that we talked about in Section 2.3 and the shading of the resulting scene with the term **rendering**. In this tutorial, we will only talk about a very small, yet important step of the rendering process: **rasterization**, which describes the step of actually drawing pixels on the screen.

Again, we will use the rather simple example of a line segment to illustrate the general approach. We said that we could describe a line segment by two vectors representing the end points of our segment. We also learned how to calculate the line that is implicitly defined by these two end points. However, we haven't yet talked about how we can actually draw the pixels in between. Given the two vectors **v** and **w** in Figure 4, how can we decide which of the pixels on our raster display have to be turned black?

To do this, we need to describe an approach that achieves this. In computer science, we call this an **algorithm**. Algorithms are a very important, maybe one of the most important concepts in computer

In mathematics and computer science, an **algorithm** is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

(from Wikipedia.org)



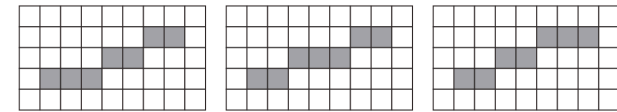
**Figure 19.** After drawing a pixel (black), for the next one to draw (gray) either the  $x$ -coordinate, or the  $y$ -coordinate, or both must increase by 1. Otherwise, the line would have a gap.

science. If you are familiar with programming, you probably know what an algorithm is, and even if the term does not sound familiar to you, you surely have written one already. In fact, even if you have no programming experience at all, you might have already designed an algorithm or at least have executed one. Because informally, an algorithm is nothing else than a specification of a procedure or single steps that have to be done to achieve a certain goal. In that sense, a cooking receipt can be seen as an algorithm. The goal that is achieved when you execute this algorithm is the creation of a nice meal. In computer science, the algorithm is basically the blueprint for our program, and the result is what the program does when we execute it on our computer. Algorithms are extensively studied in theoretical computer science with respect to different criteria, such as correctness (e.g. does our program calculate what it's supposed to do?), efficiency and run-time (e.g. is the algorithm that I designed faster than the one commonly used for this problem?), storage requirements (e.g. does this implementation require too much storage to be executed on a regular computer?), and many more. Of course, these things are relevant in practical computer science as well. For example, we do not only want our graphics to look good, but when we are playing a game, we want the virtual characters to react in real-time. And of course, we want our drawings to be correct and that our programs do not to crash.

### 3.1 Line drawing algorithms

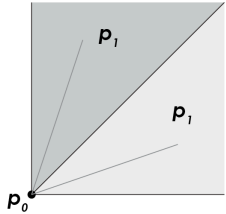
After we know now what an algorithm is, let's look into a concrete example: an algorithm that gives us the position of the pixels on a screen that we have to draw when we want to show a line segment represented by

two vectors  $\mathbf{v}$  and  $\mathbf{w}$ . In graphics, such algorithms are generally called, who would have guessed that, *line drawing algorithms*. The fact that we are talking about algorithms and not a single one already indicates that there is not a unique answer to this problem. Possible solutions can differ, for example with respect to the criteria noted above, but also with respect to the actual outcome. For example, different approaches might create lines with a different thickness. Even if our goal is to draw the thinnest possible line between two pixels that has no gap, our solution might not be unique. Figure 18 illustrates a simple case with three possible solutions, each of which representing a "reasonable" line fulfilling this goal.



**Figure 18.** Three possible solutions to the task of drawing the thinnest possible line with no gaps between two given endpoints.

Assume we want to create an algorithm that draws the thinnest possible line between two points that has no gaps. Diagonal connections between two pixels (cf. Fig. 18) are not considered a gap. Because of these assumptions, when we move from one pixel to the next one along the line (e.g. the first one that we have to drawn on the screen in the direction of our endpoint  $\mathbf{p}_1 = (x_1, y_1)$  after drawing our starting vector  $\mathbf{p}_0 = (x_0, y_0)$ ), there are three options for the coordinates  $x$  and  $y$  of this new point:  $x$  increases by 1,  $y$  increases by 1 or both increase by 1 (cf. Fig. 19). In fact, there are three more options, because  $x$ ,  $y$ , or both could also decrease by 1. Because these cases are symmetrical, they can be handled in a similar way. We will ignore them for



**Figure 20.** If the point  $p_1$  is in the upper left dark gray area, the line has more rise than run. Its  $y$ -value must increase by 1 in each step. If it is in the lower right light gray area, the line has more run than rise. Its  $x$ -value must increase by 1 in each step.

now in order to simplify the illustration and assume that our end point  $p_1$  is always to the right and above of our starting point  $p_0$ . (It is a good exercise though to write down how the algorithm that we create has to change for the cases that we are omitting for now.) If we assume that  $p_1$  is always to the right and above of  $p_0$ , there are two options when drawing the pixels from the line segment from left to right and bottom to top, and we already saw them in Figure 19: Either the  $x$ -coordinate increases by 1 in each step (if the line has more “run” than “rise”, cf. Fig. 19, left and right) or the  $y$ -coordinate increases by 1 in each step (if the line is moving faster in  $y$  than in  $x$ , cf. Fig. 19, center and right). If you are having problems understanding why, draw a grid and try coloring cells that represent a thin line without gaps that does not fulfill either of these characteristics. You will quickly see that it is impossible. The first case ( $x$ - coordinate increases by 1 in each step) is also illustrated in Figure 18.

So, at least one of the coordinate values has to increase by 1 in each step. How much the other coordinate value increases depends on the **slope**  $m$  of the line, which can be calculated by:

$$m = \frac{y_1 - y_0}{x_1 - x_0} \quad (\text{Equation 3.1})$$

Notice that because of our assumption that  $p_1$  is to the right and above of  $p_0$ , we also must have  $x_1 > x_0$  and  $y_1 > y_0$ . As a consequence, our slope is always positive. If it has more “run” than “rise”,  $m$  can be any value larger than 0 up to and including 1. If it has more “rise” than “run”, it can have any value larger than 1, including

infinitely large numbers. Figure 20 illustrates these two cases.

### 3.2 The midpoint algorithm

In the preceding subsection, we analyzed certain conditions that a line made of pixels on a raster display always has to fulfill. Now, let’s see how we can use this to specify an algorithm that draws the line segment, i.e. all pixels on the line between  $p_0 = (x_0, y_0)$ , and  $p_1 = (x_1, y_1)$ .

If the slope of the line is between 0 and 1,  $x$  always increases by 1 (cf. preceding subsection and Fig. 20). Let’s look at this case first. With each increment of  $x$ , the corresponding  $y$ -value must either increase by 1 or it stays the same. Otherwise, we would get a gap in our line. Hence, the skeleton of our algorithm could look like this:

```

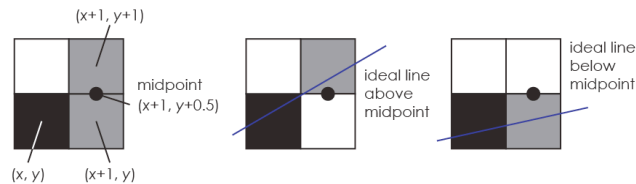
y = y0
for x = x0 to x1 do
  draw(x,y)
  if (some condition) then
    y = y + 1

```

The `draw(x, y)` command puts the color onto the pixel at position  $(x, y)$  on the screen. The rest of this code specifies the actual values of  $x$  and  $y$ . The **for** and **if-then** statements are common constructs in programming languages, but easy to understand even if you have no programming experience at all. The **for** statement is a common way to do a certain action a number of times – in our case, draw a pixel for increasing values of  $x$ . Here, it starts by assigning the first value,  $x_0$ , to  $x$ . Then it executes all the following intended code, i.e. it draws the pixel  $(x_0, y_0)$ , which is

the start of our line segment. Because we are only looking at cases now where the  $x$ -coordinate value increases by 1 in each step, it continues to do this for each  $x$ -value from  $x_0$  to  $x_1$ , which is the end of our line segment. But how do we specify the right  $y$ -coordinate value? In the first step  $y$  is  $y_0$ , because that is the value we assigned to it in the very first line of our code. In the next step  $x$  increases by 1, but for the  $y$ -value we must decide on some condition if it stays the same or if it increases as well (cf. Fig. 19, left and right). This is expressed by the `if` statement. If the condition, which we will specify soon, is fulfilled, we increase  $y$  by 1 for the next pixel. If not, it stays the same. But how can we decide that, i.e. what condition has to be fulfilled?

One approach is the so-called midpoint algorithm, which looks at the middle between the two options we have for drawing a pixel. To understand its basic idea, let's have a look at Figure 21 that illustrates the ideal line, both pixel options that we can draw, and the midpoint in between them.



**Figure 21.** The basic idea of the midpoint algorithm: After drawing point  $(x, y)$ , we have to decide if we need to draw point  $(x+1, y)$  or point  $(x+1, y+1)$  next (left image). If the midpoint between these points is below the ideal line, it is better to draw the upper one (center), if it is above the ideal line, it is better to draw the lower one (right).

When drawing the next pixel with  $x$ -coordinate value  $x+1$ , we have two options: to draw  $(x+1, y)$  or to draw

$(x+1, y+1)$ . Looking at the position of the ideal line with respect to the midpoint between both pixels can help us making the best decision. If the midpoint is below the ideal line, we should draw the upper pixel (cf. Fig. 21, left). If it is above the ideal line, we should draw the lower one (cf. Fig. 21, right).

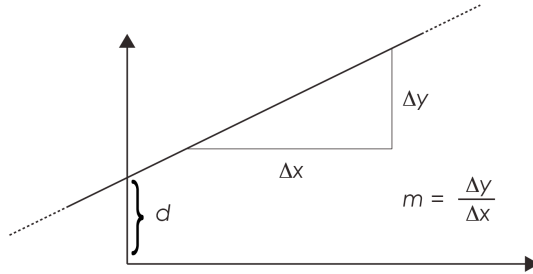
You probably guessed it: we can use this as the condition in the `if` statement of our algorithm to decide if  $y$  should be increased by 1 in the next drawing step or not. Yet, we need to express it in a formal way that our computer can understand.

In the previous section, we used two vectors to specify a line. In the context now, it is more convenient to describe the line in another way, i.e. by using the so-called *slope intercept form* (which is one of the most common forms to represent a line, so you might already know this from school). It is defined as:

$$y = m x + d$$

(Equation 3.2)

with  $m$  being the slope and  $d$  being the distance of the line from the origin  $(0, 0)$  as illustrated in Figure 22.



**Figure 22.** The slope intercept form of a line  $y = mx + d$  is defined by a line's slope  $m$  and its distance from the origin on the  $y$ -axis.

But how can we get the correct values for  $m$  and  $d$ , when we only have the two points  $\mathbf{p}_0 = (x_0, y_0)$  and  $\mathbf{p}_1 = (x_1, y_1)$ ? Because  $m$  can be calculated with any values of  $x$  and  $y$  as shown in the image, we can use equation 3.1. We can calculate  $d$  in the following way: Because  $\mathbf{p}_0 = (x_0, y_0)$  is a point on the line, it must fulfill the equation. In particular, if we set  $(x, y) = (x_0, y_0)$ , the following equation must be fulfilled:

$$y_0 = \frac{y_1 - y_0}{x_1 - x_0} x_0 + d$$

Notice that we already used equation 3.1 to represent the slope  $m$  here. Now,  $d$  is the only unknown in this equation. Solving for  $d$  give us:

$$d = y_0 - \frac{y_1 - y_0}{x_1 - x_0} x_0$$

and our original equation 3.2 becomes:

$$y = \frac{y_1 - y_0}{x_1 - x_0} x + y_0 - \frac{y_1 - y_0}{x_1 - x_0} x_0$$

or, if we multiply it with  $(x_1 - x_0)$ :

$$(x_1 - x_0) y = (y_1 - y_0) x + y_0 (x_1 - x_0) - (y_1 - y_0) x_0$$

Doing some arithmetic transformations, i.e.:

$$(x_1 - x_0) y = (y_1 - y_0) x + x_1 y_0 - x_0 y_0 - x_0 y_1 + x_0 y_0$$

we can simplify this to:

$$(x_1 - x_0) y = (y_1 - y_0) x + x_1 y_0 - x_0 y_1.$$

Bringing everything to one side gives us:

$$(x_1 - x_0) y - (y_1 - y_0) x - x_1 y_0 + x_0 y_1 = 0$$

(Equation 3.3)

This is still a valid representation of our line, but of course, it looks completely different (it is actually called the *implicit representation* of a line). But because the equation represents our line, it must be fulfilled for all points  $(x, y)$  that are on this line. That is, if we put two concrete values for  $x$  and  $y$  into the equation and calculate the result, we know that the point represented by these two coordinates is on the line if the outcome is 0. Otherwise it is not. Yet, it also give us an easy way to check if a point is above or below a line – which is exactly what we want and the reason why we did all these transformations. Look at Figure 21 again. We have drawn the pixel at position  $(x, y)$ , and now we have to decide which pixel to draw next, i.e. if we should draw  $(x+1, y)$  or  $(x+1, y+1)$ . If equation 3.3 is zero for  $(x+1, y+0.5)$ , we know that the midpoint is

directly on the line. In that case, either option would be a good result, so we can just pick one. Let's say in such a case we always pick the lower one, i.e. we draw pixel  $(x+1, y)$ . In all other cases, the result of the equation will be either  $>0$  or  $<0$ . Because the term  $(x_1 - x_0)y$  is always positive (remember that we are only looking at cases where  $p_1$  is above  $p_0$ ), the midpoint must be above the line if the result is  $>0$ , and below it otherwise. Hence, we can write our algorithm as:

```

y = y0
for x = x0 to x1 do
  draw(x,y)
  if ((x1-x0)y - (y1-y0)x - x1 y0 + x0 y1 < 0) then
    y = y + 1

```

Yeah, it took us a lot of calculations to get there, but the result is a nice, compact piece of code that does exactly what we want. Well, if and only if we are dealing with lines that fulfill our assumptions, i.e.  $p_1$  is above  $p_0$ , and  $m$  is between 0 and 1. However, the other cases can be specified in a similar way. For example, if we look for slope values of  $m$  between 1 and  $+\infty$ , the  $y$  coordinate increases by 1 in each step and we have basically the same algorithm but just have to switch the  $x$  and  $y$  values. We will not do this here, but it is a good exercise to think about all possible cases and extend the algorithm accordingly.

### Exercises

By looking at different conditions that a line drawn between two pixels on the screen has to fulfill, we were able to write a small piece of code that implements this. In order to understand an algorithm, it is often a good idea to look at a concrete example. Let's do that now.

Q1: Assume we have the following "screen" with a resolution of 12x8 pixels, and we want to draw a line between the two black ones, i.e. between the points  $p_0 = (1, 1)$ , and  $p_1 = (10, 7)$  using the midpoint line drawing algorithm. Complete the table below by filling in the values of  $x$  and  $y$  directly before the draw command is executed, and draw the related pixel in the grid on the left.

STEP	x	y
1	1	1
2		
3		
4		
5		
6		
7		
8		
9		
10	10	7

```

y = y0
for x = x0 to x1 do
  draw(x,y)
  if ((x1-x0)y - (y1-y0)x - x1 y0 + x0 y1 < 0) then
    y = y + 1

```

Q2: Our algorithm only works for "flat" lines that have more run than rise, i.e. lines with a slope  $m$  between 0 and 1. How would you extend it, so we can also draw lines with slopes from 1 to  $+\infty$ ? Write down your solution.

## 4. Conclusion

This concludes our little trip through some of the theoretical basics of computer graphics. Of course, we have omitted many things. For example, we did not cover how perspective projection works, and only used one color to draw our lines but completely ignored shading, i.e. consideration of color variations, texture, influence of lighting, etc. Yet, we also learned a lot. We started in Section 1 with a general introduction. Then introduced a mathematical framework to do some processing that is necessary to create computer graphics in Section 2, before looking at the example of an algorithm, i.e. the midpoint algorithm for line drawing in Section 3.

We have purposely chosen a more theoretical topic for this tutorial, because most people have some practical experience but do not know what to expect from the theoretical part of their studies. Be aware though that computer science, especially with the focus on game technology, is a very practical study with lots of interesting and exciting project work, too. Yet, from the tutorial it should also be clear that to do this great practical work, having a solid theoretical background is important. The goal of this tutorial was not just to teach you some basics of computer graphics, but also to illustrate typical problems that are studied in computer science and how they are commonly approached. The examples we used here were pretty simple (esp. when compared to the complex models used in video games and animated movies). But they should illustrate the general idea behind it quite well. We had many definitions and formalizations that might seem “over the top” now, but are important for doing

more complex and challenging things. After all, computer science is an exact science, so we need to specify and formalize everything to be sure that our solutions produce the correct outcome, our algorithms work, and our programs don’t crash.

---

## References and sources

Figure 1:

[http://commons.wikimedia.org/wiki/](http://commons.wikimedia.org/wiki/File:Big_Buck_Bunny_thumbnail_vlc.png)

[File:Big\\_Buck\\_Bunny\\_thumbnail\\_vlc.png](http://commons.wikimedia.org/wiki/File:Big_Buck_Bunny_thumbnail_vlc.png)

Figure 2:

<http://en.wikipedia.org/wiki/File:Pixel-example.png>

Figure 6:

<http://en.wikipedia.org/wiki/File:VectorBitmapExample.png>

Figure 14:

[http://en.wikipedia.org/wiki/File:Polygon\\_face.jpg](http://en.wikipedia.org/wiki/File:Polygon_face.jpg)