Chapter 1

Introduction

The term “computer graphics” refers to anything involved in the creation or manipulation of images on a computer, including animated images. It is a very broad field, and one in which changes and advances seem to come at a dizzying pace. It can be difficult for a beginner to know where to start. However, there is a core of fundamental ideas that are part of the foundation of most applications of computer graphics. This book attempts to cover those foundational ideas, or at least as many of them as will fit into a one-semester college-level course. While it is not possible to cover the entire field in a first course—or even a large part of it—this should be a good place to start.

This short chapter provides an overview and introduction to the material that will be covered in the rest of the book, without going into a lot of detail.

1.1 Painting and Drawing

The main focus of this book is three-dimensional (3D) graphics, where most of the work goes into producing a 3D model of a scene. But ultimately, in almost all cases, the end result of a computer graphics project is a two-dimensional image. And of course, the direct production and manipulation of 2D images is an important topic in its own right. Furthermore, a lot of ideas carry over from two dimensions to three. So, it makes sense to start with graphics in 2D. An image that is presented on the computer screen is made up of pixels. The screen consists of a rectangular grid of pixels, arranged in rows and columns. The pixels are small enough that they are not easy to see individually. In fact, for many very high-resolution displays, they become essentially invisible. At a given time, each pixel can show only one color. Most screens these days use 24-bit color, where a color can be specified by three 8-bit numbers, giving the levels of red, green, and blue in the color. Any color that can be shown on the screen is made up of some combination of these three “primary” colors. Other formats are possible, such as grayscale, where each pixel is some shade of gray and the pixel color is given by one number that specifies the level of gray on a black-to-white scale. Typically, 256 shades of gray are used. Early computer screens used indexed color, where only a small set of colors, usually 16 or 256, could be displayed. For an indexed color display, there is a numbered list of possible colors, and the color of a pixel is specified by an integer giving the position of the color in the list.

In any case, the color values for all the pixels on the screen are stored in a large block of memory known as a frame buffer. Changing the image on the screen requires changing color values that are stored in the frame buffer. The screen is redrawn many times per second, so that almost immediately after the color values are changed in the frame buffer, the colors of

the pixels on the screen will be changed to match, and the displayed image will change.

A computer screen used in this way is the basic model of raster graphics. The term "raster" technically refers to the mechanism used on older vacuum tube computer monitors: An electron beam would move along the rows of pixels, making them glow. The beam was moved across the screen by powerful magnets that would deflect the path of the electrons. The stronger the beam, the brighter the glow of the pixel, so the brightness of the pixels could be controlled by modulating the intensity of the electron beam. The color values stored in the frame buffer were used to determine the intensity of the electron beam. (For a color screen, each pixel had a red dot, a green dot, and a blue dot, which were separately illuminated by the beam.)

A modern flat-screen computer monitor is not a raster in the same sense. There is no moving electron beam. The mechanism that controls the colors of the pixels is different for different types of screen. But the screen is still made up of pixels, and the color values for all the pixels are still stored in a frame buffer. The idea of an image consisting of a grid of pixels, with numerical color values for each pixel, defines raster graphics.

* * *

Although images on the computer screen are represented using pixels, specifying individual pixel colors is not always the best way to create an image. Another way is to specify the basic geometric objects that it contains, shapes such as lines, circles, triangles, and rectangles. This is the idea that defines vector graphics: Represent an image as a list of the geometric shapes that it contains. To make things more interesting, the shapes can have attributes, such as the thickness of a line or the color that fills a rectangle. Of course, not every image can be composed from simple geometric shapes. This approach certainly wouldn't work for a picture of a beautiful sunset (or for most any other photographic image). However, it works well for many types of images, such as architectural blueprints and scientific illustrations.

In fact, early in the history of computing, vector graphics was even used directly on computer screens. When the first graphical computer displays were developed, raster displays were too slow and expensive to be practical. Fortunately, it was possible to use vacuum tube technology in another way: The electron beam could be made to directly draw a line on the screen, simply by sweeping the beam along that line. A vector graphics display would store a display list of lines that should appear on the screen. Since a point on the screen would glow only very briefly after being illuminated by the electron beam, the graphics display would go through the display list over and over, continually redrawing all the lines on the list. To change the image, it would only be necessary to change the contents of the display list. Of course, if the display list became too long, the image would start to flicker because a line would have a chance to visibly fade before its next turn to be redrawn.

But here is the point: For an image that can be specified as a reasonably small number of geometric shapes, the amount of information needed to represent the image is much smaller using a vector representation than using a raster representation. Consider an image made up of one thousand line segments. For a vector representation of the image, you only need to store the coordinates of two thousand points, the endpoints of the lines. This would take up only a few kilobytes of memory. To store the image in a frame buffer for a raster display would require much more memory. Similarly, a vector display could draw the lines on the screen more quickly than a raster display could copy the same image from the frame buffer to the screen. (As soon as raster displays became fast and inexpensive, however, they quickly displaced vector displays because of their ability to display all types of images reasonably well.)

* * *

1.1. PAINTING AND DRAWING

The divide between raster graphics and vector graphics persists in several areas of computer graphics. For example, it can be seen in a division between two categories of programs that can be used to create images: painting programs and drawing programs. In a painting program, the image is represented as a grid of pixels, and the user creates an image by assigning colors to pixels. This might be done by using a "drawing tool" that acts like a painter's brush, or even by tools that draw geometric shapes such as lines or rectangles. But the point in a painting program is to color the individual pixels, and it is only the pixel colors that are saved. To make this clearer, suppose that you use a painting program to draw a house, then draw a tree in front of the house. If you then erase the tree, you'll only reveal a blank background, not a house. In fact, the image never really contained a "house" at all—only individually colored pixels that the viewer might perceive as making up a picture of a house.

In a drawing program, the user creates an image by adding geometric shapes, and the image is represented as a list of those shapes. If you place a house shape (or collection of shapes making up a house) in the image, and you then place a tree shape on top of the house, the house is still there, since it is stored in the list of shapes that the image contains. If you delete the tree, the house will still be in the image, just as it was before you added the tree. Furthermore, you should be able to select one of the shapes in the image and move it or change its size, so drawing programs offer a rich set of editing operations that are not possible in painting programs. (The reverse, however, is also true.)

A practical program for image creation and editing might combine elements of painting and drawing, although one or the other is usually dominant. For example, a drawing program might allow the user to include a raster-type image, treating it as one shape. A painting program might let the user create "layers," which are separate images that can

be layered one on top of another to create the final image. The layers can then be manipulated much like the shapes in a drawing program (so that you could keep both your house and your tree in separate layers, even if in the image of the house is in back of the tree).

Two well-known graphics programs are *Adobe Photoshop* and *Adobe Illustrator*. *Photoshop* is in the category of painting programs, while *Illustrator* is more of a drawing program. In the world of free software, the GNU image-processing program, *Gimp*, is a good alternative to *Photoshop*, while *Inkscape* is a reasonably capable free drawing program. Short introductions to Gimp and Inkscape can be found in Appendix C.

* * *

The divide between raster and vector graphics also appears in the field of graphics file formats. There are many ways to represent an image as data stored in a file. If the original image is to be recovered from the bits stored in the file, the representation must follow some exact, known specification. Such a specification is called a graphics file format. Some popular graphics file formats include GIF, PNG, JPEG, WebP, and SVG. Most images used on the Web are GIF, PNG, or JPEG, but most browsers also have support for SVG images and for the newer WebP format.

GIF, PNG, JPEG, and WebP are basically raster graphics formats; an image is specified by storing a color value for each pixel. GIF is an older file format, which has largely been superseded by PNG, but you can still find GIF images on the web. (The GIF format supports animated images, so GIFs are often used for simple animations on Web pages.) GIF uses an indexed color model with a maximum of 256 colors. PNG can use either indexed or full 24-bit color, while JPEG is meant for full color images.

The amount of data necessary to represent a raster image can be quite large. However, the data usually contains a lot of redundancy, and the data can be "compressed" to reduce its size. GIF and PNG use lossless data compression, which means that the original image can be recovered perfectly from the compressed data. JPEG uses a lossy data compression algorithm, which means that the image that is recovered from a JPEG file is not exactly the same as the original image; some information has been lost. This might not sound like a good idea, but in fact the difference is often not very noticeable, and using lossy compression usually permits a greater reduction in the size of the compressed data. JPEG generally works well for photographic images, but not as well for images that have sharp edges between different colors. It is especially bad for line drawings and images that contain text; PNG is the preferred format for such images. WebP can use both lossless and lossy compression.

SVG, on the other hand, is fundamentally a vector graphics format (although SVG images can include raster images). SVG is actually an XML-based language for describing twodimensional vector graphics images. "SVG" stands for "Scalable Vector

Graphics," and the term "scalable" indicates one of the advantages of vector graphics: There is no loss of quality when the size of the image is increased. A line between two points can be represented at any scale, and it is still the same perfect geometric line. If you try to greatly increase the size of a raster image, on the other hand, you will find that you don't have enough color values for all the pixels in the new image; each pixel from the original image will be expanded to cover a rectangle of pixels in the scaled image, and you will get multi-pixel blocks of uniform color. The scalable nature of SVG images make them a good choice for web browsers and for graphical elements on your computer's desktop. And indeed, some desktop environments are now using SVG images for their desktop icons.

* * *

A digital image, no matter what its format, is specified using a coordinate system. A coordinate system sets up a correspondence between numbers and geometric points. In two dimensions, each point is assigned a pair of numbers, which are called the coordinates of the point. The two coordinates of a point are often called its *x*-coordinate and *y*-coordinate, although the names "x" and "y" are arbitrary.

A raster image is a two-dimensional grid of pixels arranged into rows and columns. As such, it has a natural coordinate system in which each pixel corresponds to a pair of integers giving the number of the row and the number of the column that contain the pixel. (Even in this simple case, there is some disagreement as to whether the rows should be numbered from top-to-bottom or from bottom-to-top.)

For a vector image, it is natural to use real-number coordinates. The coordinate system for an image is arbitrary to some degree; that is, the same image can be specified using different coordinate systems. I do not want to say a lot about coordinate systems here, but they will be a major focus of a large part of the book, and they are even more important in three-dimensional graphics than in two dimensions.

1.2 Elements of 3D Graphics

When we turn to 3D graphics, we find that the most common approaches have more in common with vector graphics than with raster graphics. That is, the content of an image is specified as a list of geometric objects. The technique is referred to as geometric modeling. The starting point is to construct an "artificial 3D world" as a collection of simple geometric shapes, arranged in three-dimensional space. The objects can have attributes that, combined with global properties of the world, determine the appearance of the objects. Often, the range of basic shapes is very limited, perhaps including only points, line segments, and triangles. A

more complex shape such as a polygon or sphere can be built or approximated as a collection of more basic shapes, if it is not itself considered to be basic. To make a two-dimensional image of the scene, the scene is projected from three dimensions down to two dimensions. Projection is the equivalent of taking a photograph of the scene. Let's look at how it all works in a little more detail.

First, the geometry.... We start with an empty 3D space or "world." Of course, this space exists only conceptually, but it's useful to think of it as real and to be able to visualize it in your mind. The space needs a coordinate system that associates each point in the space with three numbers, usually referred to as the *x*, *y*, and *z* coordinates of the point. This coordinate system is referred to as "world coordinates."

We want to build a scene inside the world, made up of geometric objects. For example, we can specify a line segment in the scene by giving the coordinates of its two endpoints, and we can specify a triangle by giving the coordinates of its three vertices. The smallest building blocks that we have to work with, such as line segments and triangles, are called geometric primitives. Different graphics systems make different sets of primitives available, but in many cases only very basic shapes such as lines and triangles are considered primitive. A complex scene can contain a large number of primitives, and it would be very difficult to create the scene by giving explicit coordinates for each individual primitive. The solution, as any programmer should immediately guess, is to chunk together primitives into reusable components. For example, for a scene that contains several automobiles, we might create a geometric model of a wheel. An automobile can be modeled as four wheels together with models of other components. And we could then use several copies of the automobile model in the scene. Note that once a geometric model has been designed, it can be used as a component in more complex models. This is referred to as hierarchical modeling.

Suppose that we have constructed a model of a wheel out of geometric primitives. When that wheel is moved into position in the model of an automobile, the coordinates of all of its primitives will have to be adjusted. So what exactly have we gained by building the wheel? The point is that all of the coordinates in the wheel are adjusted *in the same way*. That is, to place the wheel in the automobile, we just have to specify a single adjustment that is applied to the wheel as a whole. The type of "adjustment" that is used is called a geometric transform (or geometric transformation). A geometric transform is used to adjust the size, orientation, and position of a geometric object. When making a model of an automobile, we build *one* wheel. We then apply four different transforms to the wheel model to add four copies of the wheel to the automobile. Similarly, we can add several automobiles to a scene by applying different transforms to the same automobile model.

The three most basic kinds of geometric transform are called scaling, rotation, and translation. A scaling transform is used to set the size of an object, that is, to make it

bigger or smaller by some specified factor. A rotation transform is used to set an object's orientation, by rotating it by some angle about some specific axis. A translation transform is used to set the position of an object, by displacing it by a given amount from its original position. In this book, we will meet these transformations first in two dimensions, where they are easier to understand. But it is in 3D graphics that they become truly essential.

* * *

Next, appearance.... Geometric shapes by themselves are not very interesting. You have to be able to set their appearance. This is done by assigning attributes to the geometric objects. An obvious attribute is color, but getting a realistic appearance turns out to be a lot more complicated than simply specifying a color for each primitive. In 3D graphics, instead of color, we usually talk about material. The term material here refers to the properties that determine the intrinsic visual appearance of a surface. Essentially, this means how the surface interacts with light that hits the surface. Material properties can include a basic color as well as other properties such as shininess, roughness, and transparency.

One of the most useful kinds of material property is a texture. In most general terms, a texture is a way of varying material properties from point-to-point on a surface. The most common use of texture is to allow different colors for different points. This is often done by using a 2D image as a texture. The image can be applied to a surface so that the image looks like it is "painted" onto the surface. However, texture can also refer to changing values for things like transparency or "bumpiness." Textures allow us to add detail to a scene without using a huge number of geometric primitives; instead, you can use a smaller number of textured primitives.

A material is an intrinsic property of an object, but the actual appearance of the object also depends on the environment in which the object is viewed. In the real world, you don't see anything unless there is some light in the environment. The same is true in 3D graphics: you have to add simulated lighting to a scene. There can be several sources of light in a scene. Each light source can have its own color, intensity, and direction or position. The light from those sources will then interact with the material properties of the objects in the scene. Support for lighting in a graphics system can range from fairly simple to very complex and computationally intensive.

* * *

Finally, the image.... In general, the ultimate goal of 3D graphics is to produce 2D images of the 3D world. The transformation from 3D to 2D involves viewing and projection. The world looks different when seen from different points of view. To set up a point of view, we need to specify the position of the viewer and the direction that the viewer is looking. It is also necessary to specify an "up" direction, a direction that will be

pointing upwards in the final image. This can be thought of as placing a "virtual camera" into the scene. Once the view is set up, the world as seen from that point of view can be projected into 2D. Projection is analogous to taking a picture with the camera.

The final step in 3D graphics is to assign colors to individual pixels in the 2D image. This process is called rasterization, and the whole process of producing an image is referred to as rendering the scene.

In many cases the ultimate goal is not to create a single image, but to create an animation, consisting of a sequence of images that show the world at different times. In an animation, there are small changes from one image in the sequence to the next. Almost any aspect of a scene can change during an animation, including coordinates of primitives, transformations, material properties, and the view. For example, an object can be made to grow over the course of an animation by gradually increasing the scale factor in a scaling transformation that is applied to the object. And changing the view during an animation can give the effect of moving or flying through the scene. Of course, it can be difficult to compute the necessary changes. There are many techniques to help with the computation. One of the most important is to use a "physics engine," which computes the motion and interaction of objects based on the laws of physics. (However, you won't learn about physics engines in this book.)

1.3 Hardware and Software

We will be using OpenGL as the primary basis for 3D graphics programming. The original version of OpenGL was released in 1992 by a company named Silicon Graphics, which was known for its graphics workstations—powerful, expensive computers designed for intensive graphical applications. (Today, you have more graphics computing power on your smart phone.) OpenGL is supported by the graphics hardware in most modern computing devices, including desktop computers, laptops, and many mobile devices. In the form of WebGL, it is the used for most 3D graphics on the Web. This section will give you a bit of background about the history of OpenGL and about the graphics hardware that supports it.

In the first desktop computers, the contents of the screen were managed directly by the CPU. For example, to draw a line segment on the screen, the CPU would run a loop to set the color of each pixel that lies along the line. Needless to say, graphics could take up a lot of the CPU's time. And graphics performance was very slow, compared to what we expect today. So what has changed? Computers are much faster in general, of course, but the big change is that in modern computers, graphics processing is done by a specialized component called a GPU, or Graphics Processing Unit. A GPU includes processors for doing graphics computations; in fact, it can include a large number of such processors that work in parallel to greatly speed up graphical operations. It also

includes its own dedicated memory for storing things like images and lists of coordinates. GPU processors have very fast access to data that is stored in GPU memory—much faster than their access to data stored in the computer's main memory.

To draw a line or perform some other graphical operation, the CPU simply has to send commands, along with any necessary data, to the GPU, which is responsible for actually carrying out those commands. The CPU offloads most of the graphical work to the GPU, which is optimized to carry out that work very quickly. The set of commands that the GPU understands make up the API of the GPU. OpenGL is an example of a graphics API, and most GPUs support OpenGL in the sense that they can understand OpenGL commands, or at least that OpenGL commands can efficiently be translated into commands that the GPU can understand.

OpenGL is not the only graphics API. In fact, it is in the process of being replaced by more modern alternatives, including Vulkan an open API from the same group that is responsible for OpenGL. There are also proprietary APIs used by Apple and Microsoft: Metal and Direct3D. As for the Web, a new API called WebGPU has been under development for some time and is already implemented in some Web browsers. These newer APIs are complex and low-level. They are designed more for speed and efficiency rather than ease-of-use. Metal, Direct3D, and Vulkan are not covered in this textbook, but WebGPU is introduced in Chapter 9. For most of the book, we will use OpenGL, because it provides an easier introduction to 3D graphics, and WebGL, because it is still the major API for 3D graphics in Web browsers.

* * *

I have said that OpenGL is an API, but in fact it is a series of APIs that have been subject to repeated extension and revision. In 2023, the current (and perhaps final) version is 4.6, which was first released in 2017. It is very different from the 1.0 version from 1992. Furthermore, there is a specialized version called OpenGL ES for "embedded systems" such as mobile phones and tablets. And there is also WebGL, for use in Web browsers, which is basically a port of OpenGL ES. It will be useful to know something about how and why OpenGL has changed.

First of all, you should know that OpenGL was designed as a "client/server" system. The server, which is responsible for controlling the computer's display and performing graphics computations, carries out commands issued by the client. Typically, the server is a GPU, including its graphics processors and memory. The server executes OpenGL commands. The client is the CPU in the same computer, along with the application program that it is running. OpenGL commands come from the program that is running on the CPU. However, it is actually possible to run OpenGL programs remotely over a network. That is, you can execute an application program on a remote computer (the

OpenGL client), while the graphics computations and display are done on the computer that you are actually using (the OpenGL server).

The key idea is that the client and the server are separate components, and there is a communication channel between those components. OpenGL commands and the data that they need are communicated from the client (the CPU) to the server (the GPU) over that channel. The capacity of the channel can be a limiting factor in graphics performance. Think of drawing an image onto the screen. If the GPU can draw the image in microseconds, but it takes milliseconds to send the data for the image from the CPU to the GPU, then the great speed of the GPU is irrelevant—most of the time that it takes to draw the image is communication time.

For this reason, one of the driving factors in the evolution of OpenGL has been the desire to limit the amount of communication that is needed between the CPU and the GPU. One approach is to store information in the GPU's memory. If some data is going to be used several times, it can be transmitted to the GPU once and stored in memory there, where it will be immediately accessible to the GPU. Another approach is to try to decrease the number of OpenGL commands that must be transmitted to the GPU to draw a given image.

OpenGL draws primitives such as triangles. Specifying a primitive means specifying coordinates and attributes for each of its vertices. In the original OpenGL 1.0, a separate command was used to specify the coordinates of each vertex, and a command was needed each time the value of an attribute changed. To draw a single triangle would require three or more commands. Drawing a complex object made up of thousands of triangles would take many thousands of commands. Even in OpenGL 1.1, it became possible to draw such an object with a single command instead of thousands. All the data for the object would be loaded into arrays, which could then be sent in a single step to the GPU. Unfortunately, if the object was going to be drawn more than once, then the data would have to be retransmitted each time the object was drawn. This was fixed in OpenGL 1.5 with Vertex Buffer Objects. A VBO is a block of memory in the GPU that can store the coordinates or attribute values for a set of vertices. This makes it possible to reuse the data without having to retransmit it from the CPU to the GPU every time it is used.

Similarly, OpenGL 1.1 introduced texture objects to make it possible to store several images on the GPU for use as textures. This means that texture images that are going to be reused several times can be loaded once into the GPU, so that the GPU can easily switch between images without having to reload them.

* * *

As new capabilities were added to OpenGL, the API grew in size. But the growth was still outpaced by the invention of new, more sophisticated techniques for doing graphics.

Some of these new techniques were added to OpenGL, but the problem is that no matter how many features you add, there will always be demands for new features—as well as complaints that all the new features are making things too complicated! OpenGL was a giant machine, with new pieces always being tacked onto it, but still not pleasing everyone. The real solution was to make the machine **programmable**. With OpenGL 2.0, it became possible to write programs

to be executed as part of the graphical computation in the GPU. The programs are run on the GPU at GPU speed. A programmer who wants to use a new graphics technique can write a program to implement the feature and just hand it to the GPU. The OpenGL API doesn't have to be changed. The only thing that the API has to support is the ability to send programs to the GPU for execution.

The programs are called shaders (although the term doesn't really describe what most of them actually do). The first shaders to be introduced were vertex shaders and fragment shaders. When a primitive is drawn, some work has to be done at each vertex of the primitive, such as applying a geometric transform to the vertex coordinates or using the attributes and global lighting environment to compute the color of that vertex. A vertex shader is a program that can take over the job of doing such "per-vertex" computations. Similarly, some work has to be done for each pixel inside the primitive. A fragment shader can take over the job of performing such "per-pixel" computations. (Fragment shaders are also called pixel shaders.)

The idea of programmable graphics hardware was very successful—so successful that in OpenGL 3.0, the usual per-vertex and per-fragment processing was deprecated (meaning that its use was discouraged). And in OpenGL 3.1, it was removed from the OpenGL standard, although it is still present as an optional extension. In practice, all the original features of OpenGL are still supported in desktop versions of OpenGL and will probably continue to be available in the future. On the embedded system side, however, with OpenGL ES 2.0 and later, the use of shaders is mandatory, and a large part of the OpenGL 1.1 API has been completely removed. WebGL, the version of OpenGL for use in web browsers, is based on OpenGL ES, and it also requires shaders to get anything at all done. Nevertheless, we will begin our study of OpenGL with version 1.1. Most of the concepts and many of the details from that version are still relevant, and it offers an easier entry point for someone new to 3D graphics programming.

OpenGL shaders are written in GLSL (OpenGL Shading Language). Like OpenGL itself, GLSL has gone through several versions. We will spend some time later in the course studying GLSL ES, the version used with WebGL and OpenGL ES. GLSL uses a syntax similar to the C programming language.

* * *

As a final remark on GPU hardware, I should note that the computations that are done for different vertices are pretty much independent, and so can potentially be done in parallel. The same is true of the computations for different fragments. In fact, GPUs can have hundreds or thousands of processors that can operate in parallel. Admittedly, the individual processors are much less powerful than a CPU, but then typical per-vertex and per-fragment computations are not very complicated. The large number of processors, and the large amount of parallelism that is possible in graphics computations, makes for impressive graphics performance even on fairly inexpensive GPUs.