



Developing Graphics Frameworks with Python and OpenGL

Lee Stemkoski
Michael Pascale



CRC Press
Taylor & Francis Group

Developing Graphics Frameworks with Python and OpenGL



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Developing Graphics Frameworks with Python and OpenGL

Lee Stemkoski
Michael Pascale



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

First edition published 2022
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

© 2022 Lee Stemkoski and Michael Pascale

CRC Press is an imprint of Taylor & Francis Group, LLC

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

“The Open Access version of this book, available at www.taylorfrancis.com, has been made available under a Creative Commons Attribution-Non Commercial-No Derivatives 4.0 license”

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Stemkoski, Lee, author. | Pascale, Michael, author.

Title: Developing graphics frameworks with Python and OpenGL /
Lee Stemkoski, Michael Pascale.

Description: First edition. | Boca Raton : CRC Press, 2021. |

Includes bibliographical references and index.

Identifiers: LCCN 2021002036 | ISBN 9780367721800 (hardback) |

ISBN 9781003181378 (ebook)

Subjects: LCSH: OpenGL. | Computer graphics—Computer programs. |

Python (Computer program language) | Computer graphics—Mathematics.

Classification: LCC T385 .S7549 2021 | DDC 006.6—dc23

LC record available at <https://lccn.loc.gov/2021002036>

ISBN: 978-0-367-72180-0 (hbk)

ISBN: 978-1-032-02146-1 (pbk)

ISBN: 978-1-003-18137-8 (ebk)

DOI: 10.1201/9781003181378

Typeset in Minion Pro
by codeMantra

Contents

Authors, ix

CHAPTER 1 ■ INTRODUCTION TO COMPUTER GRAPHICS	1
1.1 CORE CONCEPTS AND VOCABULARY	2
1.2 THE GRAPHICS PIPELINE	8
1.2.1 Application Stage	9
1.2.2 Geometry Processing	10
1.2.3 Rasterization	12
1.2.4 Pixel Processing	14
1.3 SETTING UP A DEVELOPMENT ENVIRONMENT	17
1.3.1 Installing Python	17
1.3.2 Python Packages	19
1.3.3 Sublime Text	21
1.4 SUMMARY AND NEXT STEPS	23
CHAPTER 2 ■ INTRODUCTION TO PYGAME AND OpenGL	25
2.1 CREATING WINDOWS WITH PYGAME	25
2.2 DRAWING A POINT	32
2.2.1 OpenGL Shading Language	32
2.2.2 Compiling GPU Programs	36
2.2.3 Rendering in the Application	42
2.3 DRAWING SHAPES	46
2.3.1 Using Vertex Buffers	46
2.3.2 An Attribute Class	49

2.3.3	Hexagons, Triangles, and Squares	51
2.3.4	Passing Data between Shaders	59
2.4	WORKING WITH UNIFORM DATA	64
2.4.1	Introduction to Uniforms	64
2.4.2	A Uniform Class	65
2.4.3	Applications and Animations	67
2.5	ADDING INTERACTIVITY	77
2.5.1	Keyboard Input with Pygame	77
2.5.2	Incorporating with Graphics Programs	80
2.6	SUMMARY AND NEXT STEPS	81
CHAPTER 3 ■ MATRIX ALGEBRA AND TRANSFORMATIONS		83
3.1	INTRODUCTION TO VECTORS AND MATRICES	83
3.1.1	Vector Definitions and Operations	84
3.1.2	Linear Transformations and Matrices	88
3.1.3	Vectors and Matrices in Higher Dimensions	98
3.2	GEOMETRIC TRANSFORMATIONS	102
3.2.1	Scaling	102
3.2.2	Rotation	103
3.2.3	Translation	109
3.2.4	Projections	112
3.2.5	Local Transformations	119
3.3	A MATRIX CLASS	123
3.4	INCORPORATING WITH GRAPHICS PROGRAMS	125
3.5	SUMMARY AND NEXT STEPS	132
CHAPTER 4 ■ A SCENE GRAPH FRAMEWORK		133
4.1	OVERVIEW OF CLASS STRUCTURE	136
4.2	3D OBJECTS	138
4.2.1	Scene and Group	141
4.2.2	Camera	142
4.2.3	Mesh	143
4.3	GEOMETRY OBJECTS	144

4.3.1	Rectangles	145
4.3.2	Boxes	147
4.3.3	Polygons	150
4.3.4	Parametric Surfaces and Planes	153
4.3.5	Spheres and Related Surfaces	156
4.3.6	Cylinders and Related Surfaces	158
4.4	MATERIAL OBJECTS	164
4.4.1	Base Class	165
4.4.2	Basic Materials	166
4.5	RENDERING SCENES WITH THE FRAMEWORK	172
4.6	CUSTOM GEOMETRY AND MATERIAL OBJECTS	177
4.7	EXTRA COMPONENTS	184
4.7.1	Axes and Grids	185
4.7.2	Movement Rig	188
4.8	SUMMARY AND NEXT STEPS	192
CHAPTER 5 ■ TEXTURES		193
5.1	A TEXTURE CLASS	194
5.2	TEXTURE COORDINATES	201
5.2.1	Rectangles	202
5.2.2	Boxes	202
5.2.3	Polygons	203
5.2.4	Parametric Surfaces	204
5.3	USING TEXTURES IN SHADERS	206
5.4	RENDERING SCENES WITH TEXTURES	212
5.5	ANIMATED EFFECTS WITH CUSTOM SHADERS	215
5.6	PROCEDURALLY GENERATED TEXTURES	221
5.7	USING TEXT IN SCENES	228
5.7.1	Rendering Text Images	228
5.7.2	Billboarding	232
5.7.2.1	<i>Look-At Matrix</i>	232
5.7.2.2	<i>Sprite Material</i>	236
5.7.3	Heads-Up Displays and Orthogonal Cameras	241

5.8	RENDERING SCENES TO TEXTURES	247
5.9	POSTPROCESSING	254
5.10	SUMMARY AND NEXT STEPS	265
CHAPTER 6 ■ LIGHT AND SHADOW		267
6.1	INTRODUCTION TO LIGHTING	268
6.2	LIGHT CLASSES	271
6.3	NORMAL VECTORS	274
6.3.1	Rectangles	274
6.3.2	Boxes	275
6.3.3	Polygons	276
6.3.4	Parametric Surfaces	276
6.4	USING LIGHTS IN SHADERS	280
6.4.1	Structs and Uniforms	280
6.4.2	Light-Based Materials	282
6.5	RENDERING SCENES WITH LIGHTS	291
6.6	EXTRA COMPONENTS	295
6.7	BUMP MAPPING	298
6.8	BLOOM AND GLOW EFFECTS	302
6.9	SHADOWS	312
6.9.1	Theoretical Background	312
6.9.2	Adding Shadows to the Framework	317
6.10	SUMMARY AND NEXT STEPS	328
INDEX, 331		

Authors

Lee Stemkoski is a professor of mathematics and computer science. He earned his Ph.D. in mathematics from Dartmouth College in 2006 and has been teaching at the college level since. His specialties are computer graphics, video game development, and virtual and augmented reality programming.

Michael Pascale is a software engineer interested in the foundations of computer science, programming languages, and emerging technologies. He earned his B.S. in Computer Science from Adelphi University in 2019. He strongly supports open source software and open access educational resources.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction to Computer Graphics

THE IMPORTANCE OF COMPUTER graphics in modern society is illustrated by the great quantity and variety of applications and their impact on our daily lives. Computer graphics can be two-dimensional (2D) or three-dimensional (3D), animated, and interactive. They are used in data visualization to identify patterns and relationships, and also in scientific visualization, enabling researchers to model, explore, and understand natural phenomena. Computer graphics are used for medical applications, such as magnetic resonance imaging (MRI) and computed tomography (CT) scans, and architectural applications, such as creating blueprints or virtual models. They enable the creation of tools such as training simulators and software for computer-aided engineering and design. Many aspects of the entertainment industry make use of computer graphics to some extent: movies may use them for creating special effects, generating photorealistic characters, or rendering entire films, while video games are primarily interactive graphics-based experiences. Recent advances in computer graphics hardware and software have even helped virtual reality and augmented reality technology enter the consumer market.

The field of computer graphics is continuously advancing, finding new applications, and increasing in importance. For all these reasons, combined with the inherent appeal of working in a highly visual medium, the field of computer graphics is an exciting area to learn about, experiment with, and work in. In this book, you'll learn how to create a robust framework

capable of rendering and animating interactive three-dimensional scenes using modern graphics programming techniques.

Before diving into programming and code, you'll first need to learn about the core concepts and vocabulary in computer graphics. These ideas will be revisited repeatedly throughout this book, and so it may help to periodically review parts of this chapter to keep the overall process in mind. In the second half of this chapter, you'll learn how to install the necessary software and set up your development environment.

1.1 CORE CONCEPTS AND VOCABULARY

Our primary goal is to generate two-dimensional images of three-dimensional scenes; this process is called *rendering* the scene. Scenes may contain two- and three-dimensional objects, from simple geometric shapes such as boxes and spheres, to complex models representing real-world or imaginary objects such as teapots or alien lifeforms. These objects may simply appear to be a single color, or their appearance may be affected by textures (images applied to surfaces), light sources that result in *shading* (the darkness of an object not in direct light) and *shadows* (the silhouette of one object's shape on the surface of another object), or environmental properties such as fog. Scenes are rendered from the point of view of a virtual camera, whose relative position and orientation in the scene, together with its intrinsic properties such as angle of view and depth of field, determine which objects will be visible or partially obscured by other objects when the scene is rendered. A 3D scene containing multiple shaded objects and a virtual camera is illustrated in Figure 1.1. The region contained within the truncated pyramid shape outlined in white (called a *frustum*) indicates the space visible to the camera. In Figure 1.1, this region completely contains the red and green cubes, but only contains part of the blue sphere, and the yellow cylinder lies completely outside of this region. The results of rendering the scene in Figure 1.1 are shown in Figure 1.2.

From a more technical, lower-level perspective, rendering a scene produces a *raster*—an array of *pixels* (picture elements) which will be displayed on a screen, arranged in a two-dimensional grid. Pixels are typically extremely small; zooming in on an image can illustrate the presence of individual pixels, as shown in Figure 1.3.

On modern computer systems, pixels specify colors using triples of floating-point numbers between 0 and 1 to represent the amount of red, green, and blue light present in a color; a value of 0 represents no amount of that color is present, while a value of 1 represents that color is displayed

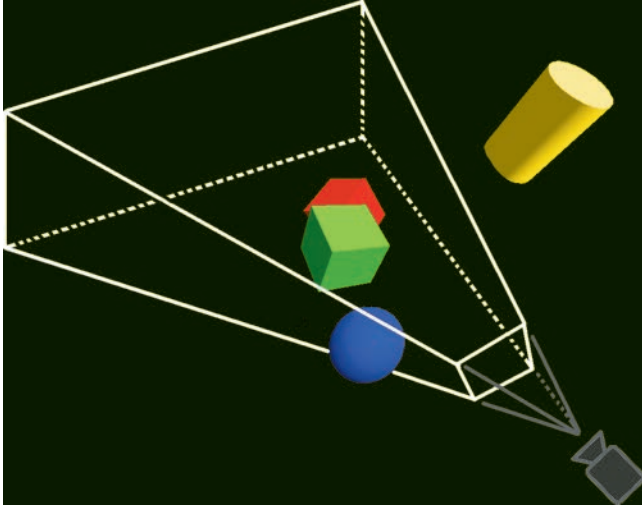


FIGURE 1.1 Three-dimensional scene with geometric objects, viewing region (white outline) and virtual camera (lower right).

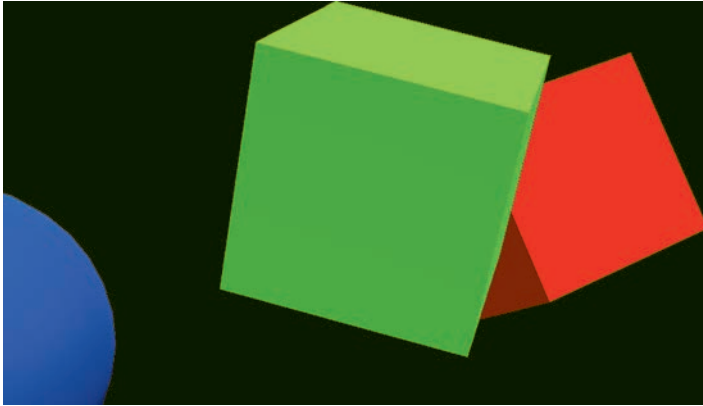


FIGURE 1.2 Results of rendering the scene from Figure 1.1

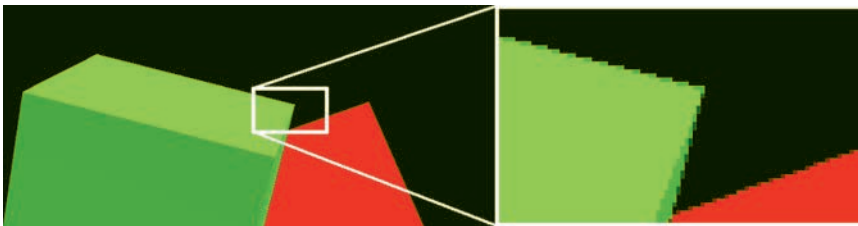


FIGURE 1.3 Zooming in on an image to illustrate individual pixels.

	R	G	B		R	G	B
red	1	0	0	black	0	0	0
orange	1	0.5	0	white	1	1	1
yellow	1	1	0	gray	0.5	0.5	0.5
green	0	1	0	brown	0.5	0.2	0
blue	0	0	1	pink	1	0.5	0.5
violet	0.5	0	1	cyan	0	1	1

FIGURE 1.4 Various colors and their corresponding (R, G, B) values.

at full (100%) intensity. These three colors are typically used since photoreceptors in the human eye take in those particular colors. The triple (1, 0, 0) represents red, (0, 1, 0) represents green, and (0, 0, 1) represents blue. Black and white are represented by (0, 0, 0) and (1, 1, 1), respectively. Additional colors and their corresponding triples of values specifying the amounts of red, green, and blue (often called *RGB values*) are illustrated in Figure 1.4.

The quality of an image depends in part on its *resolution* (the number of pixels in the raster) and *precision* (the number of bits used for each pixel). As each bit has two possible values (0 or 1), the number of colors that can be expressed with N-bit precision is 2^N . For example, early video game consoles with 8-bit graphics were able to display $2^8 = 256$ different colors. Monochrome displays could be said to have 1-bit graphics, while modern displays often feature “high color” (16-bit, 65,536 color) or “true color” (24-bit, more than 16 million colors) graphics. Figure 1.5 illustrates the same image rendered with high precision but different resolutions, while Figure 1.6 illustrates the same image rendered with high resolution but different precision levels.

In computer science, a *buffer* (or *data buffer*, or *buffer memory*) is a part of a computer's memory that serves as temporary storage for data while it is being moved from one location to another. Pixel data is stored in a region of memory called the *framebuffer*. A framebuffer may contain multiple buffers that store different types of data for each pixel. At a minimum, the framebuffer must contain a *color buffer*, which stores RGB values. When rendering a 3D scene, the framebuffer must also contain a *depth buffer*, which stores distances from points on scene objects to the virtual camera. Depth values are used to determine whether the various points on each object are in front of or behind other objects (from the camera's perspective), and thus whether they will be visible when the scene is rendered. If one scene object obscures another and a transparency effect is

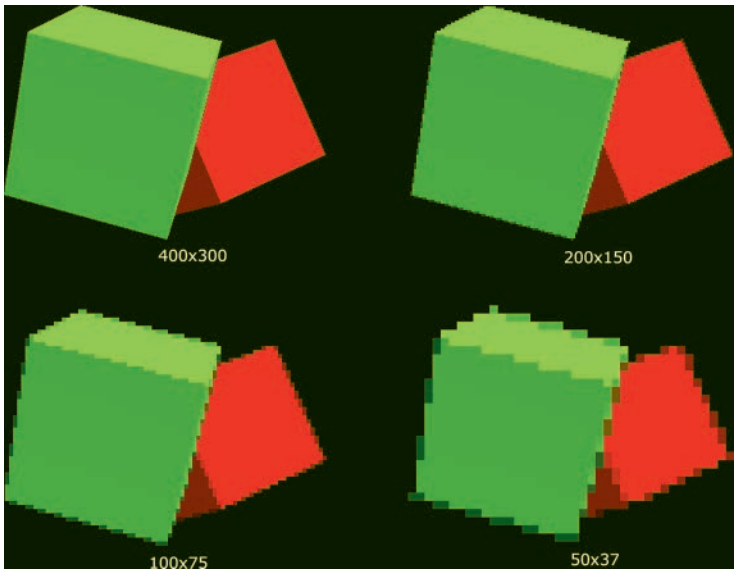


FIGURE 1.5 A single image rendered with different resolutions.

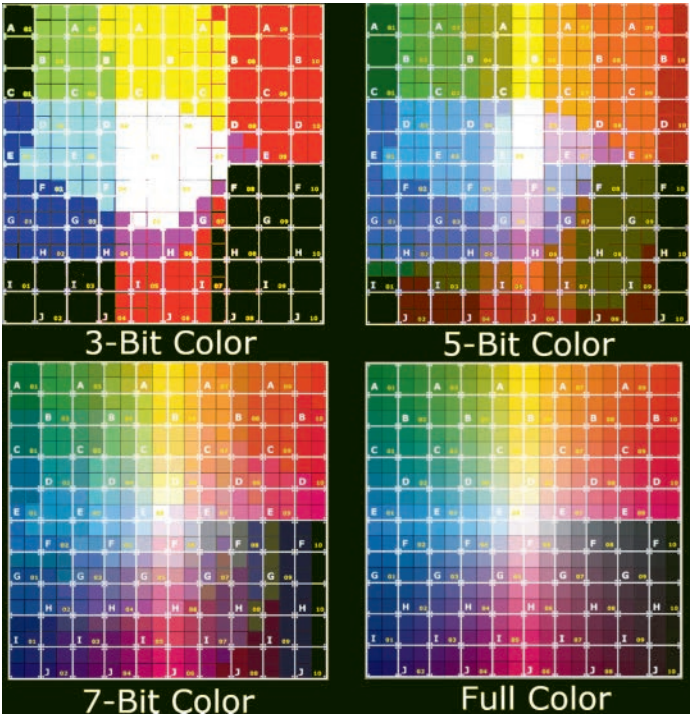


FIGURE 1.6 A single image rendered with different precisions.

desired, the renderer makes use of *alpha values*: floating-point numbers between 0 and 1 that specifies how overlapping colors should be blended together; the value 0 indicates a fully transparent color, while the value 1 indicates a fully opaque color. Alpha values are also stored in the color buffer along with RGB color values; the combined data is often referred to as RGBA color values. Finally, framebuffers may contain a buffer called a *stencil buffer*, which may be used to store values used in generating advanced effects, such as shadows, reflections, or portal rendering.

In addition to rendering three-dimensional scenes, another goal in computer graphics is to create animated scenes. Animations consist of a sequence of images displayed in quick enough succession that the viewer interprets the objects in the images to be continuously moving or changing in appearance. Each image that is displayed is called a *frame*. The speed at which these images appear is called the *frame rate* and is measured in *frames per second* (FPS). The standard frame rate for movies and television is 24 FPS. Computer monitors typically display graphics at 60 FPS. For virtual reality simulations, developers aim to attain 90 FPS, as lower frame rates may cause disorientation and other negative side effects in users. Since computer graphics must render these images in real time, often in response to user interaction, it is vital that computers be able to do so quickly.

In the early 1990s, computers relied on the *central processing unit* (CPU) circuitry to perform the calculations needed for graphics. As real-time 3D graphics became increasingly common in video game platforms (including arcades, gaming consoles, and personal computers), there was increased demand for specialized hardware for rendering these graphics. This led to the development of the *graphics processing unit* (GPU), a term coined by the Sony Corporation that referred to the circuitry in their PlayStation video game console, released in 1994. The Sony GPU performed graphics-related computational tasks including managing a framebuffer, drawing polygons with textures, and shading and transparency effects. The term *GPU* was popularized by the NVidia Corporation in 1999 with their release of the GeForce 256, a single-chip processor that performed geometric transformations and lighting calculations in addition to the rendering computations performed by earlier hardware implementations. NVidia was the first company to produce a GPU capable of being programmed by developers: each geometric vertex could be processed by a short program, as could every rendered pixel, before the resulting image was displayed on screen. This processor, the GeForce 3, was introduced in 2001 and was also used

in the Xbox video game console. In general, GPUs feature a highly parallel structure that enables them to be more efficient than CPUs for rendering computer graphics. As computer technology advances, so does the quality of the graphics that can be rendered; modern systems are able to produce real-time photorealistic graphics at high resolutions.

Programs that are run by GPUs are called *shaders*, initially so named because they were used for shading effects, but now used to perform many different computations required in the rendering process. Just as there are many high-level programming languages (such as Java, JavaScript, and Python) used to develop CPU-based applications, there are many shader programming languages. Each shader language implements an *application programming interface* (API), which defines a set of commands, functions, and protocols that can be used to interact with an external system—in this case, the GPU. Some APIs and their corresponding shader languages include

- The DirectX API and High-Level Shading Language (HLSL), used on Microsoft platforms, including the Xbox game console
- The Metal API and Metal Shading Language, which runs on modern Mac computers, iPhones, and iPads
- The OpenGL (Open Graphics Library) API and OpenGL Shading Language (GLSL), a cross-platform library.

This book will focus on OpenGL, as it is the most widely adopted graphics API. As a cross-platform library, visual results will be consistent on any supported operating system. Furthermore, OpenGL can be used in concert with a variety of high-level languages using *bindings*: software libraries that bridge two programming languages, enabling functions from one language to be used in another. For example, some bindings to OpenGL include

- JOGL (<https://jogamp.org/jogl/www/>) for Java
- WebGL (<https://www.khronos.org/webgl/>) for JavaScript
- PyOpenGL (<http://pyopengl.sourceforge.net/>) for Python

The initial version of OpenGL was released by Silicon Graphics, Inc. (SGI) in 1992 and has been managed by the Khronos Group since 2006. The Khronos Group is a non-profit technology consortium, whose members

include graphics card manufacturers and general technology companies. New versions of the OpenGL specification are released regularly to support new features and functions. In this book, you will learn about many of the OpenGL functions that allow you to take advantage of the graphics capabilities of the GPU and render some truly impressive three-dimensional scenes. The steps involved in this rendering process are described in detail in the sections that follow.

1.2 THE GRAPHICS PIPELINE

A *graphics pipeline* is an abstract model that describes a sequence of steps needed to render a three-dimensional scene. Pipelining allows a computational task to be split into subtasks, each of which can be worked on in parallel, similar to an assembly line for manufacturing products in a factory, which increases overall efficiency. Graphics pipelines increase the efficiency of the rendering process, enabling images to be displayed at faster rates. Multiple pipeline models are possible; the one described in this section is commonly used for rendering real-time graphics using OpenGL, which consists of four stages (illustrated by Figure 1.7):

- *Application Stage*: initializing the window where rendered graphics will be displayed; sending data to the GPU
- *Geometry Processing*: determining the position of each vertex of the geometric shapes to be rendered, implemented by a program called a *vertex shader*
- *Rasterization*: determining which pixels correspond to the geometric shapes to be rendered
- *Pixel Processing*: determining the color of each pixel in the rendered image, involving a program called a *fragment shader*

Each of these stages is described in more detail in the sections that follow; the next chapter contains code that will begin to implement many of the processes described here.

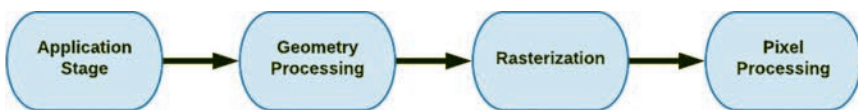


FIGURE 1.7 The graphics pipeline.

1.2.1 Application Stage

The application stage primarily involves processes that run on the CPU. One of the first tasks is to create a window where the rendered graphics will be displayed. When working with OpenGL, this can be accomplished using a variety of programming languages. The window (or a canvas-like object within the window) must be initialized so that the graphics are read from the GPU framebuffer. In the case of animated or interactive applications, the main application contains a loop that re-renders the scene repeatedly, typically aiming for a rate of 60 FPS. Other processes that may be handled by the CPU include monitoring hardware for user input events, or running algorithms for tasks such as physics simulation and collision detection.

Another class of tasks performed by the application includes reading data required for the rendering process and sending it to the GPU. This data may include vertex attributes (which describe the appearance of the geometric shapes being rendered), images that will be applied to surfaces, and source code for the vertex shader and fragment shader programs (which will be used later on during the graphics pipeline). OpenGL describes the functions that can be used to transmit this data to the GPU; these functions are accessed through the bindings of the programming language used to write the application. Vertex attribute data is stored in GPU memory buffers called *vertex buffer objects* (VBOs), while images that will be used as textures are stored in *texture buffers*. It is important to note that this stored data is not initially assigned to any particular program variables; these associations are specified later. Finally, source code for the vertex shader and fragment shader programs needs to be sent to the GPU, compiled, and loaded. If needed, buffer data can be updated during the application's main loop, and additional data can be sent to shader programs as well.

Once the necessary data has been sent to the GPU, before rendering can take place, the application needs to specify the associations between attribute data stored in VBOs and attribute variables in the vertex shader program. A single geometric shape may have multiple attributes for each vertex (such as position and color), and the corresponding data is streamed from buffers to variables in the shader during the rendering process. It is also frequently necessary to work with many sets of such associations: there may be multiple geometric shapes (with data stored in different buffers) that are rendered by the same shader program, or each shape may be rendered by a different shader program. These sets of associations can be

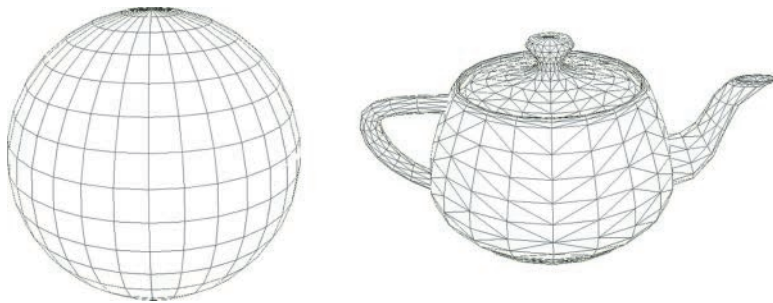


FIGURE 1.8 Wireframe meshes representing a sphere and a teapot.

conveniently managed by using *vertex array objects* (VAOs), which store this information and can be activated and deactivated as needed during the rendering process.

1.2.2 Geometry Processing

In computer graphics, the shape of a geometric object is defined by a *mesh*: a collection of points that are grouped into lines or triangles, as illustrated in Figure 1.8.

In addition to the overall shape of an object, additional information may be required to describe how the object should be rendered. The properties or attributes that are specific to rendering each individual point are grouped together into a data structure called a *vertex*. At a minimum, a vertex must contain the three-dimensional position of the corresponding point. Additional data contained by a vertex often includes

- a color to be used when rendering the point
- *texture coordinates* (or UV coordinates), which indicate a point in an image that is mapped to the vertex
- a *normal vector*, which indicates the direction perpendicular to a surface and is typically used in lighting calculations

Figure 1.9 illustrates different renderings of a sphere that make use of these attributes. Additional vertex attributes may be defined as needed.

During the geometry processing stage, the vertex shader is applied to each of the vertices; each attribute variable in the shader receives data from a buffer according to previously specified associations. The primary purpose of the vertex shader is to determine the final position of

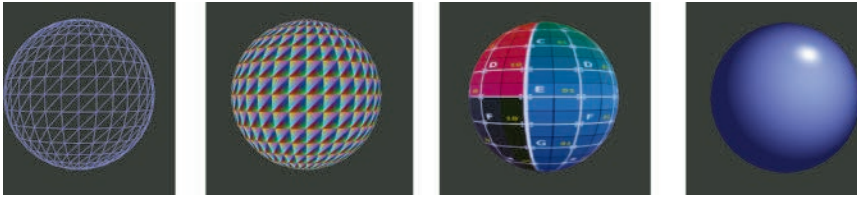


FIGURE 1.9 Different renderings of a sphere: wireframe, vertex colors, texture, and with lighting effects.

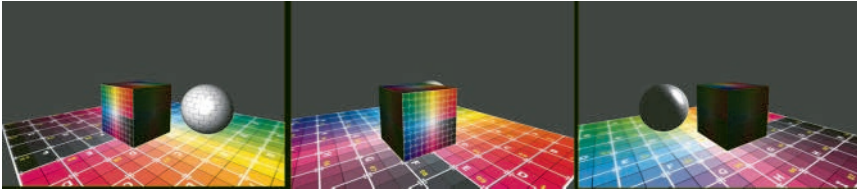


FIGURE 1.10 One scene rendered from multiple camera locations and angles.

each point being rendered, which is typically calculated from a series of transformations:

- the collection of points defining the intrinsic shape of an object may be translated, rotated, and scaled so that the object appears to have a particular location, orientation, and size with respect to a virtual three-dimensional world. This process is called the *model transformation*; coordinates expressed from this frame of reference are said to be in world space
- there may be a virtual camera with its own position and orientation in the virtual world. In order to render the world from the camera's point of view, the coordinates of each object in the world must be converted to a frame of reference relative to the camera itself. This process is called the *view transformation*, and coordinates in this context are said to be in *view space* (or *camera space*, or *eye space*). The effect of the placement of the virtual camera on the rendered image is illustrated in Figure 1.10
- the set of points in the world considered to be visible, occupying either a box-shaped or frustum-shaped region, must be scaled to and aligned with the space rendered by OpenGL: a cube-shaped region consisting of all points whose coordinates are between -1 and 1 .

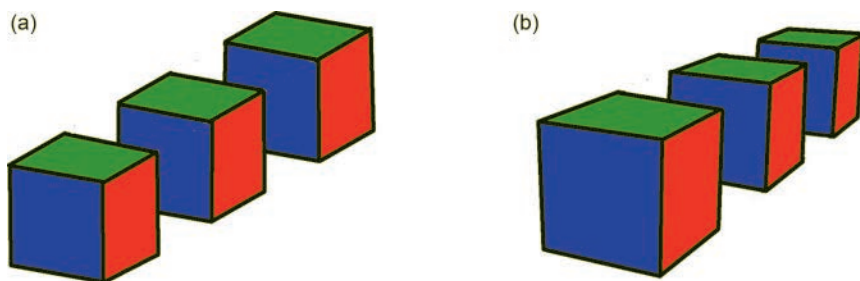


FIGURE 1.11 A series of cubes rendered with orthogonal projection (a) and perspective projection (b).

The position of each point returned by the vertex shader is assumed to be expressed in this frame of reference. Any points outside this region are automatically discarded or *clipped* from the scene; coordinates expressed at this stage are said to be in *clip space*. This task is accomplished with a *projection transformation*. More specifically, it is called an *orthographic projection* or a *perspective projection*, depending on whether the shape of the visible world region is a box or a frustum. A perspective projection is generally considered to produce more realistic images, as objects that are farther away from the virtual camera will require greater compression by the transformation and thus appear smaller when the scene is rendered. The differences between the two types of projections are illustrated in Figure 1.11.

In addition to these transformation calculations, the vertex shader may perform additional calculations and send additional information to the fragment shader as needed.

1.2.3 Rasterization

Once the final positions of each vertex have been specified by the vertex shader, the rasterization stage begins. The points themselves must first be grouped into the desired type of *geometric primitive*: points, lines, or triangles, which consist of sets of 1, 2, or 3 points. In the case of lines or triangles, additional information must be specified. For example, consider an array of points [A, B, C, D, E, F] to be grouped into lines. They could be grouped in disjoint pairs, as in (A, B), (C, D), (E, F), resulting in a set of disconnected line segments. Alternatively, they could be grouped in overlapping pairs, as in (A, B), (B, C), (C, D), (D, E), (E, F), resulting in a set of connected line segments (called a line strip). The type of geometric

primitive and method for grouping points is specified using an OpenGL function parameter when the rendering process begins. The process of grouping points into geometric primitives is called *primitive assembly*.

Once the geometric primitives have been assembled, the next step is to determine which pixels correspond to the interior of each geometric primitive. Since pixels are discrete units, they will typically only approximate the continuous nature of a geometric shape, and a criterion must be given to clarify which pixels are in the interior. Three simple criteria could be

1. the entire pixel area is contained within the shape
2. the center point of the pixel is contained within the shape
3. any part of the pixel is contained within the shape

These effects of applying each of these criteria to a triangle are illustrated in Figure 1.12, where the original triangle appears outlined in blue, and pixels meeting the criteria are shaded gray.

For each pixel corresponding to the interior of a shape, a *fragment* is created: a collection of data used to determine the color of a single pixel in a rendered image. The data stored in a fragment always includes the *raster position*, also called *pixel coordinates*. When rendering a three-dimensional scene, fragments will also store a *depth* value, which is needed when points on different geometric objects would overlap from the perspective of the viewer. When this happens, the associated fragments would correspond to the same pixel, and the depth value determines which fragment's data should be used when rendering this pixel.

Additional data may be assigned to each vertex, such as a color, and passed along from the vertex shader to the fragment shader. In this case, a new data field is added to each fragment. The value assigned to this field at

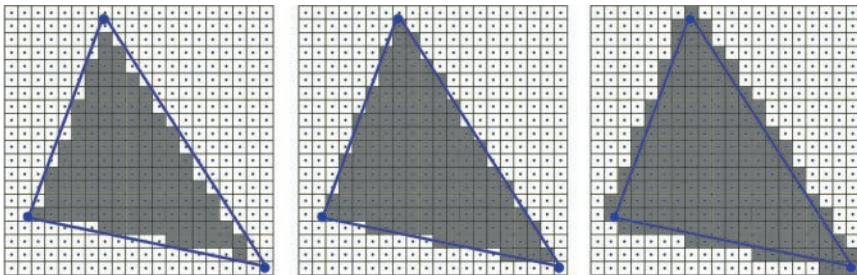


FIGURE 1.12 Different criteria for rasterizing a triangle.

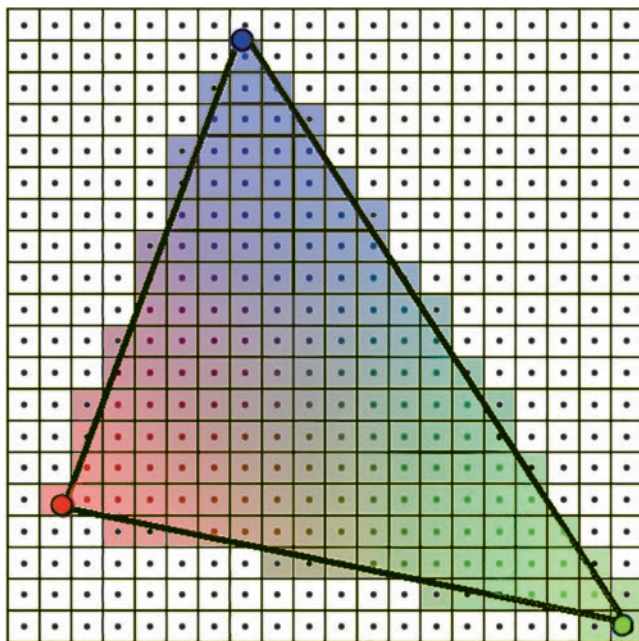


FIGURE 1.13 Interpolating color attributes.

each interior point is *interpolated* from the values at the vertices: calculated using a weighted average, depending on the distance from the interior point to each vertex. The closer an interior point is to a vertex, the greater the weight of that vertex's value when calculating the interpolated value. For example, if the vertices of a triangle are assigned the colors red, green, and blue, then each pixel corresponding to the interior of the triangle will be assigned a combination of these colors, as illustrated in Figure 1.13.

1.2.4 Pixel Processing

The primary purpose of this stage is to determine the final color of each pixel, storing this data in the color buffer within the framebuffer. During the first part of the pixel processing stage, a program called the fragment shader is applied to each of the fragments to calculate their final color. This calculation may involve a variety of data stored in each fragment, in combination with data globally available during rendering, such as

- a base color applied to the entire shape
- colors stored in each fragment (interpolated from vertex colors)

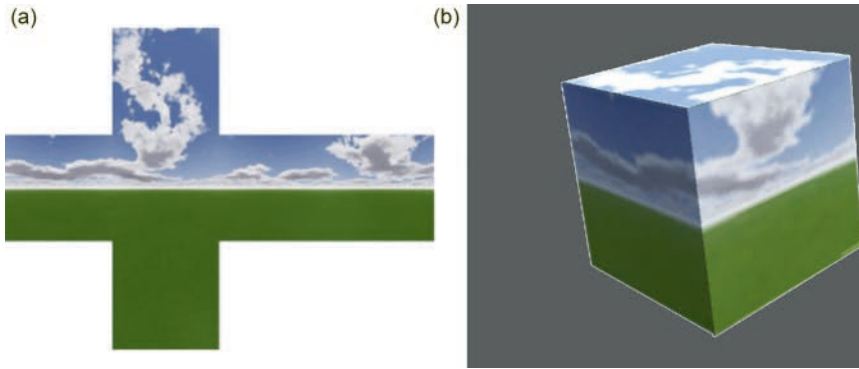


FIGURE 1.14 An image file (a) used as a texture for a 3D object (b).

- textures (images applied to the surface of the shape, illustrated by Figure 1.14), where colors are sampled from locations specified by texture coordinates
- light sources, whose relative position and/or orientation may lighten or darken the color, depending on the direction the surface is facing at a point, specified by normal vectors

Some aspects of the pixel processing stage are automatically handled by the GPU. For example, the depth values stored in each fragment are used in this stage to resolve visibility issues in a three-dimensional scene, determining which parts of objects are blocked from view by other objects. After the color of a fragment has been calculated, the fragment's depth value will be compared to the value currently stored in the depth buffer at the corresponding pixel coordinates. If the fragment's depth value is smaller than the depth buffer value, then the corresponding point is closer to the viewer than any that were previously processed, and the fragment's color will be used to overwrite the data currently stored in the color buffer at the corresponding pixel coordinates.

Transparency is also handled by the GPU, using the alpha values stored in the color of each fragment. The alpha value of a color is used to indicate how much of this color should be blended with another color. For example, when combining a color C_1 with an alpha value of 0.6 with another color C_2 , the resulting color will be created by adding 60% of the value from each component of C_1 to 40% of the value from each component of C_2 . Figure 1.15 illustrates a simple scene involving transparency.

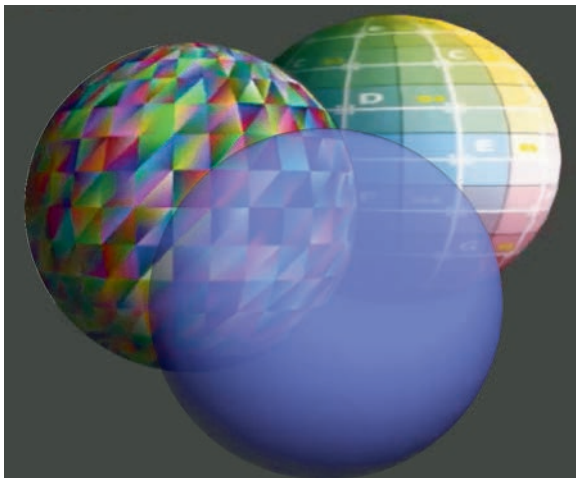


FIGURE 1.15 Rendered scene with transparency.

However, rendering transparent objects has some complex subtleties. These calculations occur at the same time that depth values are being resolved, and so scenes involving transparency must render objects in a particular order: all opaque objects must be rendered first (in any order), followed by transparent objects ordered from farthest to closest with respect to the virtual camera. Not following this order may cause transparency effects to fail. For example, consider a scene, such as that in Figure 1.15, containing a single transparent object close to the camera and multiple opaque objects farther from the camera that appear behind the transparent object. Assume that, contrary to the previously described rendering order, the transparent object is rendered first, followed by the opaque objects in some unknown order. When the fragments of the opaque objects are processed, their depth value will be greater than the value stored in the depth buffer (corresponding to the closer transparent object), and so the opaque fragments' color data will automatically be discarded, rather than blended with the currently stored color. Even attempting to use the alpha value of the transparent object stored in the color buffer in this example does not resolve the underlying issue, because when the fragments of each opaque object are being rendered, it is not possible at this point to determine if they may have been occluded from view by another opaque fragment (only the closest depth value, corresponding to the transparent object, is stored), and thus, it is unknown which opaque fragment's color values should be blended into the color buffer.

1.3 SETTING UP A DEVELOPMENT ENVIRONMENT

Most parts of the graphics pipeline discussed in the previous section—geometry processing, rasterization, and pixel processing—are handled by the GPU, and as mentioned previously, this book will use OpenGL for these tasks. For developing the application, there are many programming languages one could select from. In this book, you will be using Python to develop these applications, as well as a complete graphics framework to simplify the design and creation of interactive, animated, three-dimensional scenes.

1.3.1 Installing Python

To prepare your development environment, the first step is to download and install a recent version of Python (version 3.8 as of this writing) from <http://www.python.org> (Figure 1.16); installers are available for Windows, Mac OS X, and a variety of other platforms.

- When installing for Windows, check the box next to **add to path**. Also, select the options **custom installation** and **install for all users**; this simplifies setting up alternative development environments later.

The Python installer will also install IDLE, Python’s Integrated Development and Learning Environment, which can be used for developing the graphics framework presented throughout this book. A more



FIGURE 1.16 Python homepage: <http://www.python.org>.

sophisticated alternative is recommended, such as Sublime Text, which will be introduced later on in this chapter, and some of its advantages discussed. (If you are already familiar with an alternative Python development environment, you are of course also welcome to use that instead.)

IDLE has two main window types. The first window type, which automatically opens when you run IDLE, is the shell window, an interactive window that allows you to write Python code which is then immediately executed after pressing the Enter key. Figure 1.17 illustrates this window after entering the statements `123 + 214` and `print("Hello, world!")`. The second window type is the *editor window*, which functions as a text editor, allowing you to open, write, and save files containing Python code, which are called *modules* and typically use the .py file extension. An editor window can be opened from the shell window by selecting either **File>New File** or **File>Open...** from the menu bar. Programs may be run from the editor window by choosing **Run>Run Module** from the menu bar; this will display the output in a shell window (opening a new shell window if none are open). Figure 1.18 illustrates creating a file in the editor window containing the following code:

```
print("Hello, world!")
print("Have a nice day!")
```

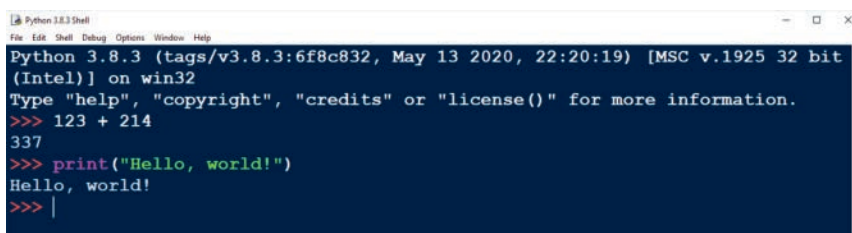


FIGURE 1.17 IDLE shell window.

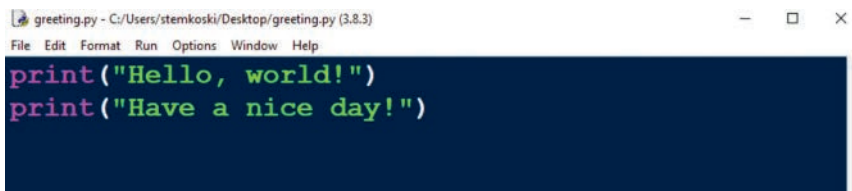
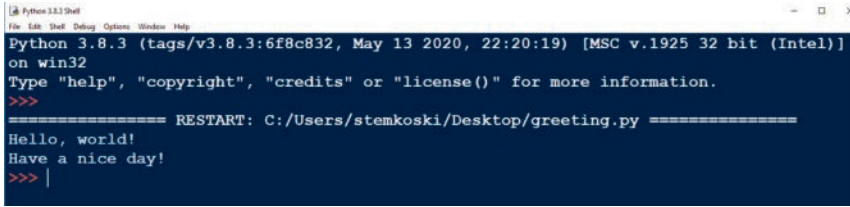


FIGURE 1.18 IDLE editor window.



```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
====>>>
==== RESTART: C:/Users/stemkoski/Desktop/greeting.py =====
Hello, world!
Have a nice day!
====>>> |
```

FIGURE 1.19 Results of running the Python program from Figure 1.18.

Figure 1.19 illustrates the results of running this code, which appear in a shell window.

1.3.2 Python Packages

Once Python has been successfully installed, your next step will be to install some *packages*, which are collections of related modules that provide additional functionality to Python. The easiest way to do this is by using *pip*, a software tool for package installation in Python. In particular, you will install

- Pygame (<http://www.pygame.org>), a package that can be used to easily create windows and handle user input
- Numpy (<https://numpy.org/>), a package for mathematics and scientific computing
- PyOpenGL and PyOpenGL_accelerate (<http://pyopengl.sourceforge.net/>), which provide a set of bindings from Python to OpenGL.

If you are using Windows, open Command Prompt or PowerShell (run with administrator privileges so that the packages are automatically available to all users) and enter the following command, which will install all of the packages described above:

```
py -m pip install pygame numpy PyOpenGL
PyOpenGL_accelerate
```

If you are using MacOS, the command is slightly different. Enter

```
python3-m pip install pygame numpy PyOpenGL
PyOpenGL_accelerate
```

To verify that these packages have been installed correctly, open a new IDLE shell window (restart IDLE if it was open before installation). To check Pygame, enter the following code, and press the Enter key:

```
import pygame
```

You should see a message that contains the number of the Pygame version that has been installed, such as "pygame 1.9.6", and a greeting message such as "Hello from the pygame community". If instead you see a message that contains the text `No module named 'pygame'`, then Pygame has not been correctly installed. Furthermore, it will be important to install a recent version of Pygame—at least a development version of Pygame 2.0.0. If an earlier version has been installed, return to the command prompt and in the install command above, change `pygame` to `pygame==2.0.0.dev10` to install a more recent version.

Similarly, to check the Numpy installation, instead use the code:

```
import numpy
```

In this case, if you see no message at all (just another input prompt), then the installation was successful. If you see a message that contains the text `No module named 'numpy'`, then Numpy has not been correctly installed. Finally, to check PyOpenGL, instead use the code:

```
import OpenGL
```

As was the case with testing the Numpy package, if there is no message displayed, then the installation was successful, but a message mentioning that the module is not installed will require you to try re-installing the package.

If you encounter difficulties installing any of these packages, there is additional help available online:

- Pygame: <https://www.pygame.org/wiki/GettingStarted>
- Numpy: <https://numpy.org/install/>
- PyOpenGL: at <http://pyopengl.sourceforge.net/documentation/installation.html>

1.3.3 Sublime Text

When working on a large project involving multiple files, you may want to install an alternative development environment, rather than restrict yourself to working with IDLE. The authors particularly recommend Sublime Text, which has the following advantages:

- lines are numbered for easy reference
- tabbed interface for working with multiple files in a single window
- editor supports multi-column layout to view and edit different files simultaneously
- directory view to easily navigate between project files in a project
- able to run scripts and display output in console area
- free, full-featured trial version available

To install the application, visit the Sublime Text website (<https://www.sublimetext.com/>), shown in Figure 1.20, and click on the “download” button (whose text may differ from the figure to reference the operating system you are using). Alternatively, you may click the download link in the navigation bar to view all downloadable versions. After downloading, you will need to run the installation program, which will require administrator-level privileges on your system. If unavailable, you may alternatively download a “portable version” of the software, which can

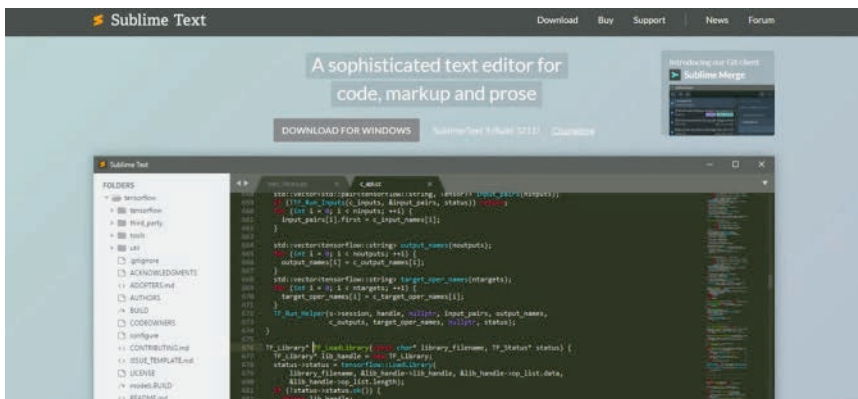


FIGURE 1.20 Sublime Text homepage

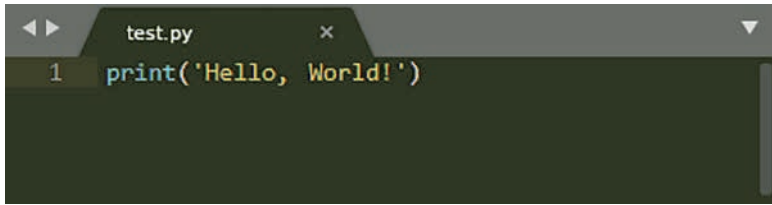


FIGURE 1.21 Sublime Text editor window.

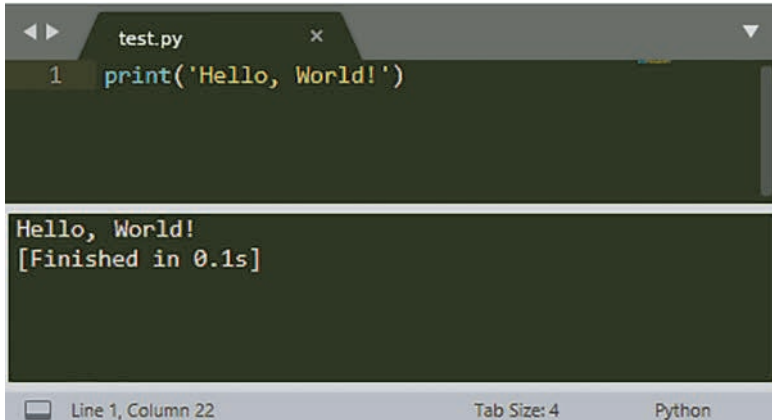


FIGURE 1.22 Output from Figure 1.21.

be found via the download link previously mentioned. While a free trial version is available, if you choose to use this software extensively, you are encouraged to purchase a license.

After installation, start the Sublime Text software. A new editor window will appear, containing an empty file. As previously mentioned, Sublime Text can be used to run Python scripts automatically, provided that Python has been installed for all users of your computer and it is included on the system path. To try out this feature, in the editor window, as shown in Figure 1.21, enter the text:

```
print("Hello, world!")
```

Next, save your file with the name `test.py`; the `.py` extension causes Sublime Text to recognize it as a Python script file, and syntax highlighting will be applied. Finally, from the menu bar, select `Tools > Build` or press the keyboard key combination `Ctrl + B` to build and run the application. The output will appear in the console area, as illustrated in Figure 1.22.

1.4 SUMMARY AND NEXT STEPS

In this chapter, you learned about the core concepts and vocabulary used in computer graphics, including rendering, buffers, GPUs, and shaders. Then, you learned about the four major stages in the graphics pipeline: the application stage, geometry processing, rasterization, and pixel processing; this section introduced additional terminology, including vertices, VBOs, VAOs, transformations, projections, fragments, and interpolation. Finally, you learned how to set up a Python development environment. In the next chapter, you will use Python to start implementing the graphics framework that will realize these theoretical principles.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>