Preface

Welcome to the third edition of Computer Graphics Through OpenGL: From Theory to Experiments! The first edition appeared late 2010 and the second four years after in 2014. Now, it’s been another four years in which I received a lot of thoughtful and, happily, mostly positive feedback. I heard from instructors, students, as well as individuals using the book for self-study. Their observations together with my own classroom experience started me off a year ago writing a new edition. It’s been a fairly intense past several months, most of my waking hours spent sitting where I am now in front of the computer. But it’s been fun too – being in communion with an imaginary reader who I am trying to enlighten and keep engaged at the same time – and I am pleased with the result. I hope you will be too. Let’s get to the facts.

About the Book

This is an introductory textbook on computer graphics with equal emphasis on theory and practice. The programming language used is C++, with OpenGL as the graphics API, which means calls are made to the OpenGL library from C++ programs. OpenGL is taught from scratch.

The book has been written to be used as a textbook for a first college course, as well as for self-study.

After Chapters 1-16 – the undergraduate core of the book – the reader will have a good grasp of the concepts underpinning 3D computer graphics, as well as an ability to code sophisticated 3D scenes and animation, including games and movies. We begin with classical pre-shader OpenGL before proceeding to the latest OpenGL 4.x (more about our exposition style further on). Chapters 17-21, though advanced, but still mainstream, could be selected topics for an undergraduate course or part of a second course.

Specs

This book comprises 21 chapters, an extended appendix on a fundamental math topic, plus two more appendices containing a math self-test and its solutions. It comes with approximately 180 programs, 270 experiments based on these programs, 750 exercises, including theory and programming exercises, 110 worked examples, and 700 four-color illustrations, include drawings and screenshots. An instructor’s manual containing solutions to selected exercises is available from the publisher. The book was typeset using \LaTeX and figures drawn in Adobe Illustrator.

From the Second Edition to the Third

• Cover to cover revision and reorganization of topics.

New topics include:
Preface

- Timer queries and performance measurement.
- Importing externally created objects.
- Texturing spheres.
- Framebuffer objects.
- Rendering to texture.
- Texture matrices.
- Cube mapping a skybox.
- Shadow mapping curved surfaces.
- OpenGL 4.x:
  - Procedural textures.
  - Specular maps.
  - Normal maps.
  - Multiple program objects.
  - Particle systems with transform feedback.
- 10 new programs, 20 new experiments, 100 new exercises, 10 new examples, 50 new figures.
- Programming environment simplified, programs developed on Windows 10 and Microsoft Visual Studio (MSVS) 2015 and tested on MSVS 2017.

Target Audience

- Students in a first university CG course, typically offered by a CS department at a junior/senior level. This is the primary audience for which the book was written.
- Students in a second or advanced CG course, who may use the book as preparation or reference, depending on the goals. For example, the book would be a useful reference for a study of 3D design – particularly, Bézier, B-spline and NURBS theory – and of projective transformations and their applications in CG.
- Students in a non-traditional setting, e.g., studying alone or in a professional course or an on-line program. The author has tried to be especially considerate of the reader on her own.
- Professional programmers, to use the book as a reference.

Prerequisites

Zero knowledge of computer graphics is presumed. However, the student is expected to know the following:

- Basic C++ programming. There is no need to be an expert programmer. The C++ program serves mainly as an environment for the OpenGL calls, so there’s rarely need for fancy footwork in the C++ part itself.
- Basic math. This includes coordinate geometry, trigonometry and linear algebra, all at college first-course level (or, even strong high school in some cases). For intended readers of the book who may be unsure of their math preparation, we have a self-test in Appendix B, with solutions in Appendix C. The test should tell exactly how ready you are and where the weaknesses are.
Resources

The following are available through the book’s website www.sumantaguha.com:

- Sample chapters, table of contents, preface, subject and program index, math self-test and solutions at the Home page.
- Program source code, developed on a Windows 10 platform using the Microsoft Visual Studio Community 2015 IDE and subsequently tested to run with MSVS Community 2017, which should run on other versions of Windows/MSVS, as well as Mac OS and Linux platforms. The programs are arranged chapter-wise in the top-level folder ExperimenterSource at the Downloads page.
- Guide to installing OpenGL and running the programs on a Windows/MSVS platform at the Downloads page.
- Multiplatform Experimenter software to run the experiments at the Downloads page. Experimenter’s interface is Experimenter.pdf, a file containing all the experiments from the book; except for those in Chapter 1, each is clickable to bring up the related program and workspace. Experimenter is only an aid and not mandatory – each program is stand-alone. However, it is the most convenient way to run experiments in their book order.
- Book figures in jpg format arranged in sequence as one PowerPoint presentation per chapter at the Instructor page.
- Instructor’s manual with solutions to 100 problems – instructors who have adopted the textbook can submit a request at the Instructor page.

Pedagogical Approach

Code and theory have been intertwined as far as possible in what may be called a theory-experiment-repeat loop: often, following a theoretical discussion, the reader is asked to perform validating experiments (run code, that is); sometimes, too, the other way around, an experiment is followed by an explanation of what is observed. It’s kind of like discovering physics.

Why use an API?

Needless to say, I am not a fan of the API-agnostic approach to teaching CG, where focus is on principles only, with no programming practice.

Undergrads, typically, love to code and make things happen, so there is little justification to denying the new student the joy of creating scenes, movies and games, not to mention the pride of achievement. And, why not leverage the way code and theory reinforce one another when teaching the subject, or learning on one’s own, when one can? Would you want Physics 101 without a lab section?

Moreover, OpenGL is very well-designed and the learning curve short enough to fully integrate into a first CG course. And, it is supported on every OS platform with drivers for almost every graphics card on the market; so, in fact, OpenGL is there to use for anyone who cares to.

Note to student: Our pedagogical style means that for most parts of the book you want a computer handy to run experiments. So, if you are going to snuggle up with it at night, make it a threesome with a notebook.

Note to instructor: Lectures on most topics – both of the theory and programming practice – are best based around the book’s experiments, as well as those you develop yourself. The Experimenter resource makes this convenient. Slides other than the plentiful book figures, the latter all available on-line, are rarely necessary.

How to teach modern shader-based OpenGL?

Our point of view needs careful explanation as it is different from some of our peers’. Firstly, to push the physics analogy one more time, even though relativistic mechanics seems to rule the universe, in the classroom one might prefer doing classical physics before relativity theory.
Shaders, which are the programmable parts of the modern OpenGL pipeline, add
great flexibility and power. But, so too, do they add a fair bit of complexity – even
a cursory comparison of our very first program square.cpp from Chapter 2 with
its equivalent in fourth-generation OpenGL, squareShaderized.cpp, complemented
with a vertex and a fragment shader in Chapter 15, should convince the reader of this.

Consider more carefully, say, a vertex shader. It must compute the position
coordinates of a vertex, taking into account all transformations, both modelview
and projection. However, in the classical fixed-function pipeline the user can simply issue
commands such as glTranslatef(), glFrustum(), etc., leaving to OpenGL actual
computation of the transformed coordinates; not so for the programmable pipeline,
where the reader must write herself all the needed matrix operations in the vertex
shader. We firmly believe that the new student is best served learning first how to
transform objects according to an understanding of simply how a scene comes together
physically (e.g., a ball falls to the ground, a robot arm bends at the elbow, etc.) with
the help of ready-to-use commands like glTranslatef(), and, only later, how to
define these transforms mathematically.

Such consideration applies as well to other automatic services of the fixed-function
pipeline which allow the student to focus on phenomena, disregarding initially
implementation. For example, as an instructor, I would much prefer to teach first how
diffuse light lends three-dimensionality, specular light highlights, and so on, gently
motivating Phong’s lighting equation, leaving OpenGL to grapple with its actual
implementation, which is exactly what we do in Chapter 11.

In fact, we find an understanding of the fixed-function pipeline makes the subsequent
learning of the programmable one significantly easier because it’s then clear exactly
what the shaders should try to accomplish. For example, following the fixed-function
groundwork in Chapter 11, writing shaders to implement Phong lighting, as we do in
Chapter 15, is near trivial.

We take a similarly laissez-faire attitude to classical OpenGL syntax. So long as it
eases the learning curve we’ll put up with it. Take for example the following snippet
from our very first program square.cpp:

```
    glBegin(GL_POLYGON);
    glVertex3f(20.0, 20.0, 0.0);
    glVertex3f(80.0, 20.0, 0.0);
    glVertex3f(80.0, 80.0, 0.0);
    glVertex3f(20.0, 80.0, 0.0);
    glEnd();
```

Does it not scream square – even though it’s immediate mode and uses the discarded
polygon primitive? So, we prefer this for our first lesson, avoiding thereby the
distraction of a vertex array and the call glVertexArrays(GL_TRIANGLES, 0, 4),
as in the equivalent 4.x program squareShaderized.cpp, our goal being a simple
introduction of the synthetic-camera model.

With these thoughts in mind the book starts in Chapter 2 with classical pre-shader
OpenGL, progressing gradually deeper into the API, developing CG ideas in parallel,
in a so-called theory-experiment-repeat loop. So, what exactly is an experiment?
An experiment consists either of running a book program – each usually simple for
the purpose of elucidating a single idea – or attempting to modify one based on an
understanding of the theory in order, typically, to achieve a particular visual result.

By the end of Chapter 14 the student will have acquired proficiency in pre-shader
OpenGL, a perfectly good API in itself. As well, equally importantly, she will have
an understanding of CG principles and those underlying the OpenGL pipeline, which
will dramatically ease her way through the concepts and syntax of OpenGL 4.x, the
newest generation of the API, covered in Chapters 15-16.

Does this kind of introduction to modern OpenGL, via the old and, possibly,
obsolete, not ingrain bad habits? Not at all, from our experience. When push comes
to shove, how hard is it to replace polygons with triangle strips? Or, use vertex buffer
objects (VBOs) and vertex array objects (VAOs) to store data? Does our approach cost timewise? If the goal is OpenGL 4.x, then, yes, it does take somewhat longer, but there are various possible learning sequences through the book and 4.x certainly can be reached and covered in a semester.

In short, then, we believe the correct way to modern OpenGL is through the classical version of the API because this allows the learning process to begin at a high level, so that the student can concentrate on gaining an overall end-to-end understanding of the CG pipeline first, leaving the OpenGL system to manage low-level processes (i.e., those inside the pipeline like setting transformation and projection matrices, defining fragment colors, and such). Once she has a high-level mastery, subsequently “descending” into the pipeline to take charge of fixed-function parts in order to program them instead will, in fact, be far less arduous than if she tried to do both – learn the basics and program the pipeline – at the same time.

Another point to note in this context is that, as noted before, classical OpenGL is a complete API in itself which, in fact, can be more convenient for certain applications (e.g., it allows one access to the readymade GLUT objects like spheres and toruses). There are, as well, thousands of currently live applications written in classical OpenGL, which are not going to be discarded or rewritten any time soon – the reason, in fact, for the Khronos Group to retain the compatibility version of the API – so familiarity with older syntax can be useful for the intending professional.

What about Vulkan?

We thought you might ask. Vulkan is the much-hyped “successor” to OpenGL. It is a highly explicit API, taking the programmer close to the hardware and asking her to specify almost all facets of the pipeline from end to end. Benefits of programming near the hardware include thin drivers, reduced run-time overhead and the ability to expose parallelism in the GPU. (Vulkan is not only a 3D graphics API, but used to program GPU-heavy compute applications as well.)

However, Vulkan’s explicitness and consequent verbosity make it highly unsuitable as an introductory CG API. Here are some program sizes to begin with. The first OpenGL program in the book, square.cpp, which draws a black square on a white background, is about 90 lines of code in pre-shader 2nd generation OpenGL; a functionally equivalent program, squareShaderized.cpp, written in OpenGL 4.3 later on in the book is 190 lines plus 25 lines of shader code; a minimal equivalent Vulkan program, squareVulkanized.cpp, written separately by the author is 1,100 lines (no, that’s no misprint – the reader will find the program at the Downloads page of the book’s website) plus 30 lines of shader code. Figure 1 is a screenshot.

Moreover, explicitness requires a Vulkan programmer to be familiar with the functioning of the graphics pipeline at a low level in order to specify it, which almost instantly disqualifies it from being a beginner’s API. Further, delaying programming until after the pipeline has been covered goes utterly against our own pedagogical approach which is to engage students with code the first day.

So, is OpenGL, or for that matter, this book, of any use for someone intending to learn Vulkan? Well:

(a) The Vulkan graphics pipeline is essentially the same as OpenGL’s. Therefore, learning OpenGL is progress toward Vulkan. Moreover, once a programmer has mastered OpenGL, she has most of what’s needed to “take full charge” of the pipeline, which is what Vulkan is all about.

(b) The runtime gains of Vulkan don’t begin to show up in any significant way until one gets to complex scenes with lots of textures, objects and animation. Less complicated applications - including, obviously, those in any introductory CG book - benefit little performance-wise from being written (or, rewritten) in Vulkan, not justifying the huge overhead in code.

This means that many OpenGL apps are going to stay that way and new ones continue to be written. It’s a matter of knowing which tool to use: pre-shader
OpenGL, OpenGL 4.x, Vulkan, . . . (Hooking up a U-Haul trailer to the back of a Ferrari is never a good idea.)

Nevertheless, if you are of the Vulkan or bust frame of mind then the advice we would have is to study this book up to Chapter 16, when you will have a solid understanding of fourth-generation OpenGL, then pick up, say, the canonical Vulkan guide [131], through which you should then be able to make quick progress.

Capsule Chapter Descriptions

Part I: Hello World
Chapter 1: An Invitation to Computer Graphics
A non-technical introduction to the field of computer graphics.

Chapter 2: On to OpenGL and 3D Computer Graphics
Begins the technical part of the book. It introduces OpenGL and fundamental principles of 3D CG.

Part II: Tricks of the Trade
Chapter 3: An OpenGL Toolbox
Describes a collection of OpenGL programming devices, including vertex arrays, vertex buffer and array objects, mouse and key interaction, pop-up menus, and several more.

Part III: Movers and Shapers
Chapter 4: Transformation, Animation and Viewing
Introduces the theory and programming of animation and the virtual camera. Explains user interactivity via object selection. Foundational chapter for game and movie programming.

Chapter 5: Inside Animation: The Theory of Transformations
Presents the mathematical theory behind animation, particularly linear and affine transformations in 3D.

Chapter 6: Advanced Animation Techniques
Describes frustum culling, occlusion culling as well as orienting animation using both Euler angles and quaternions, techniques essential to programming games and busy scenes.

Part IV: Geometry for the Home Office
Chapter 7: Convexity and Interpolation
Explains the theory of convexity and the role it plays in interpolation, which is the procedure of spreading material properties from the vertices of a primitive to its interior.

Chapter 8: Triangulation
Describes how and why complex objects should be split into triangles for efficient rendering.

Chapter 9: Orientation
Describes how the orientation of a primitive is used to determine the side of it that the camera sees, and the importance of consistently orienting a collection of primitives making up a single object.

Part V: Making Things Up
Chapter 10: Modeling in 3D Space
Systematizes the principles of modeling both curves and surfaces, including Bézier and fractal. Shows how to import objects from external design environments. Foundational chapter for object design.

**Part VI: Lights, Camera, Equation**

Chapter 11: Color and Light
Explains the theory of light and material color, the interaction between the two, and describes how to program light and color in 3D scenes. Foundational chapter for scene design.

Chapter 12: Textures
Explains the theory of texturing and how to apply textures to objects and render to a texture.

Chapter 13: Special Visual Techniques
Describes a set of special techniques to enhance the visual quality of a scene, including, among others, blending, billboardling, stencil buffer methods, image and pixel manipulation, cube mapping a skybox, and shadow mapping.

**Part VII: Pixels, Pixels, Everywhere**

Chapter 14: Raster Algorithms
Describes low-level rendering algorithms to determine the set of pixels on the screen corresponding to a line or a polygon.

**Part VIII: Programming Pipe Dreams**

Chapter 15: OpenGL 4.3, Shaders and the Programmable Pipeline: Liftoff
Introduces 4th generation OpenGL and GLSL (OpenGL Shading Language) and how to vertex and fragments shaders to program the pipeline, particularly to animate, light and apply textures.

Chapter 16: OpenGL 4.3, Shaders and the Programmable Pipeline: Escape Velocity
Continuing onto advanced 4th generation OpenGL topics, including, among others, instanced rendering, shader subroutines, transform feedback, particle systems, as well as tessellation and geometry shaders.

**Part IX: Anatomy of Curves and Surfaces**

Chapter 17: Bézier
Describes the theory and programming of Bézier primitives, including curves and surfaces.

Chapter 18: B-Spline
Describes the theory and programming of (polynomial) B-spline primitives, including curves and surfaces.

Chapter 19: Hermite
Introduces the basics of Hermite curves and surfaces.

**Part X: Well Projected**

Chapter 20: Applications of Projective Spaces: Projection Transformations and Rational Curves
Applies the theory of projective spaces to deduce the projection transformation in the graphics pipeline. Introduces rational Bézier and B-spline, particularly NURBS, theory and practice.

**Part XI: Time for a Pipe**

Chapter 21: Pipeline Operation
Preface

Gives a detailed view of the synthetic-camera and ray-tracing pipelines and introduces radiosity.

Appendix A: Projective Spaces and Transformations
A CG-oriented introduction to the mathematics of projective spaces and transformations. Provides a complete theoretical background for Chapter 20 on applications of projective spaces.

Appendix B: Math Self-Test
A self-test to assess math readiness for intending readers.

Appendix C: Math Self-Test Solutions
Solutions for the math self-test.

Figure 2: Chapter dependence chart: dashed arrows represent weak dependencies.
Suggested Course Outlines

See the chapter dependencies in Figure 2.

(1) Undergraduate one-semester first CG course:

This course should be based on Chapters 1-16, though full coverage might be ambitious for one semester. Instructors may pick topics to emphasize or skip, depending on their goals for the course and the chapter dependence chart.

For example, for more practice and less theory, a possible sequence would be 1 → 2 → 3 → 4 → 6 (only frustum culling) → 7 → 8 → 9 → 10 (skip curve/surface theory) → 11 → 12 → 13 → 15 → 16.

Even this abbreviated sequence may be hard to pack into one semester. Please keep in mind that when choosing what to write about the author preferred to err on the side of excess rather than less. So, almost always will the instructor find more material in a chapter than she cares to teach – we leave her to pick her way out.

The most effective teaching method with this book is to base discussion around experiments – both from the book and those the instructor develops herself. Our Experimenter software makes this especially convenient. Students should be involved in the experiments, running code simultaneously on their own machines in class. Use of slides should be minimized except, possibly, for the plentiful book figures, which are available to download, arranged as one PowerPoint presentation per chapter.

(2) Advanced CG courses:

This book could serve as a reference for a study of 3D design – particularly, Bézier (Chapter 17), B-spline (Chapter 18) and rational Bézier and NURBS theory (Chapter 20) – and of projective transformations and their applications (Appendix A and Chapter 20). From a practical point of view, Chapters 15-16 go fairly deep into the fourth generation of OpenGL and the GLSL, useful for students who may be familiar with only the classical pipeline.

(3) Self-study:

A recommended first pass would be 1 → 2 → 3 → 4 → 7 → 8 → 9 (go light on 7-9 if your math is rusty) → 10 (skip theory) → 11 → 12 → 13 → 15 → 16.

Following this the student should take up a fair-sized programming project, returning to the book as needed. For the theoretically-inclined there’s a lot to keep her busy in Chapters 5 and 17-21.

Acknowledgments

I owe a lot to many people, most of all students whom I have had the privilege of teaching in my CG classes over the years at UW-Milwaukee and then the Asian Institute of Technology.

I thank KV, Ichiro Suzuki, Glenn Wardius, Mahesh Kumar, Le Phu Binh, Maria Sell and, especially, Paul McNally, for their support at UWM, where I began to teach CG and learn OpenGL.

I am grateful to my colleagues and the staff and students at AIT for such a pleasant environment, which allowed me to combine teaching and research commitments with the writing of a book.

Preface

I am grateful to Kumpee Teeravech, Kanit Tangkathach, Thanapoom Veeranitimm, Pongpon Nilaphruek, and Wuttinan Sereethavekul, students of my CG course at AIT, for allowing me to use programs they wrote.

I owe an enormous debt of gratitude to my former student Chansophea Chuon for hundreds of hours of help with the first edition, which got this book off the ground in the first place. I thank Somying Pongpimol for her brilliant Illustrator drawings. She drew several of the figures based on my rather amateurish original Xfig sketches. I would like to thank Olivier Nicole for help with the book’s website.

I am especially grateful to Brian Barsky for encouraging me to persevere after seeing an early and awkward draft of the first edition, and subsequently inviting the finished product to the series he was then editing. Relatedly, I thank my neighbor Doug Cooper two floors down for putting me in touch with Brian at the time Brian was scouting prospective authors.

I want to acknowledge the production team at Taylor & Francis who went out of their way for this book. Particularly, I want to thank my editor Randi Cohen who is as professional and efficient as she is pleasant to deal with.

I am grateful to readers of the first and second editions, as well as reviewers who looked over drafts and proposals, whose comments led, hopefully, to significant improvements.

I have nothing but praise for the DJs and kindly staff of the Mixx nightclub in downtown Bangkok whose dance floor provided a much-needed sanctuary for me to blow off steam after long hours on the computer.

I acknowledge the many persons and businesses who were kind enough to allow me to include images to which they own copyrights.

On a personal note, I express my deep gratitude to Dr. Anupam De for keeping Kamaladi healthy enough that I could concentrate on the first edition through the few years that I spent writing it.

Finally, I must say that had I not had the opportunity to study computer science in the United States and teach there, I would never have reached a position where I could even contemplate writing a textbook. It’s true, too, that had I not moved to Thailand, this book would never have begun to be written. This is an enchanting country with a strangely liberating and lightening effect – to which thousands of expats can attest – that encourages one to express oneself.

Website and Contact Information

The book’s website is at www.sumantaguha.com. Users of the book will find there various resources including downloads. The author welcomes feedback, corrections and suggestions for improvement emailed to him at sg@sumantaguha.com.