UNIVERSITY OF CALIFORNIA
Los Angeles

An Online Collaborative Ecosystem for Educational Computer Graphics

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Garett Douglas Ridge

2018

ABSTRACT OF THE DISSERTATION

An Online Collaborative Ecosystem for Educational Computer Graphics

by

Garett Douglas Ridge
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2018
Professor Demetri Terzopoulos, Chair

This thesis builds upon existing introductory courses in the field of Computer Graphics, aiming to lower the excessive barrier of entry to graphics programming. We introduce tiny-graphics.js, a new software library for implementing educational WebGL projects in the classroom. To mitigate the difficulty of creating graphics-enabled websites and online games, we furthermore introduce the Encyclopedia of Code—a world wide web framework that encourages visitors to learn graphics, build educational graphical demos and articles, host them online, and organize them by topic. We provide our own examples that include custom educational games and tutorial articles, which are already being successfully employed to ease our undergraduate graphics students into the course material. Some of our modules expose students to new graphics techniques, while others are prototypes for new modes of online learning, collaboration, and computing. These include our "Active Textbooks" (educational 3D demos or games embedded in literate-programming-like articles). We introduce "Smart Articles," which divide crowdsourced programming tasks into distributed processes that can be tracked at separate URLs. Beyond education and graphics, our work advances the study of human-computer interaction, user interfaces, and computer-supported cooperative work. We present the results of a case study in using tiny-graphics.js and our online resources in a real-world UCLA undergraduate course, along with our results when attempting to recreate topic-based examples from an existing graphics textbook using our library.

The dissertation of Garett Douglas Ridge is approved.

Douglas S. Parker

Joseph M. Teran

Song-Chun Zhu

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Many people deserve credit for this work. First are my mom and dad, who were always available as my own patient audience for every idea I ever had. Knowing they would read my writings has always influenced my style towards the inclusion of more background details to remain accessible. This document benefited from that. Next is my girlfriend Xin who supported me full-time through the entire process of not one but two dissertation-sized projects, and who cheered me up every day.

Thanks goes to my advisor Demetri Terzopoulos for introducing me to most of the educational topics in this dissertation and many beyond, for exciting discussions about graphics and artificial life I could not have had elsewhere, and also for patiently sticking with me as my advisor despite entire years when it probably seemed that I had already given up. Likewise I want to thank my two undergraduate advisors at the University of Louisville, including Tim Hardin who continued to pursue me after I actually had given up—imploring me to reconsider passing up an undergraduate research position offer I must have only received because of his recommendation. He introduced me to Olfa Nasraoui, my REU advisor, whose lab turned me from a student bound for local industry into someone with a strong application for grad school. Besides granting access to a NASA funded research opportunity, she personally guided me through the process of applying for awards that I never would have known about, ultimately opening the door to UCLA.

I also thank all of my students. Even when I was a new TA, they provided encouragement to build interesting and fun examples to use in class, ultimately leading to the work reported in this dissertation. Their ambition when making graphics course projects and their willingness to try my examples led them to provide constant actionable feedback on my code library, leading it through many successful iterations of UCLA's *CS 174A: Introduction to Computer Graphics* course. I also want to thank my predecessors in UCLA's Computer Graphics & Vision Lab who helped build *CS 174A*, and discussions with whom helped me understand how to solve coding problems, and better model the world mathematically and simulate it graphically.

VITA

2008-2010	Undergraduate Researcher (Image Processing), University of Louisville
2010	B.S. (Computer Science), University of Louisville
2011	Computer Vision Research Intern, Dolby Laboratories
2012	Computer Graphics/Geometry Research Intern, Digital Domain Productions
2015	M.S. (Computer Science), UCLA
2011–2017	Teaching Assistant, Computer Science Department, UCLA

CHAPTER 1

Introduction

Some problems are best solved with visualizations. Finding a way to apply computer graphics to any problem is a great way to make the underlying topics more intuitive and approachable. So how does a student, a programmer, or even a mathematician learn to make use of computer graphics for the first time? Is there a right way to learn this skill?

At universities, teachers in graphics courses must routinely find good answers to this question in behalf of their students. Aside from lacking graphics experience, the students are often pursuing various majors. Some begin graphics courses without even having the programming and math background, and yet graphics can be a way for them to gain this background. Furthermore, motivated outsiders who lack access to such a university course in the first place might still prefer to learn graphics from a programmer's or mathematician's perspective, in order to increase their understanding of both, rather than trying to glean whatever they can from graphics tutorials. Online tutorials are often manuals for pre-packaged commercial solutions anyway—tools that are geared towards outputting a finished product as opposed to educating the end consumer in graphics fundamentals.

Unfortunately, for the task of creating a graphics program, most approaches today come with a high learning curve. The reality is that the processes of computer graphics are quite complicated in hardware and software, with many caveats, such as performance limitations. These extra parts of the graphics learning curve do not particularly help the learner to acquire the math and programming intuition that computer graphics can deliver.

In this thesis, we aspire to lower the bar for graphics learners, and make the job easier for graphics instructors as well. We offer supplementary material for a university (or college-level) Computer Graphics course, and a novel online framework for supporting and expanding

that material.

In particular, we develop a new programming library, "tiny-graphics.js", and a new website, the "Encyclopedia of Code". We use these to introduce new tools and paradigms for education, especially in topics of Computer Graphics and Computer Science in general. The outcome is a dramatic improvement upon the work of Edward Angel associated with his Computer Graphics textbook (Angel and Shreiner, 2014a), including course material, codebase, and online demos, which have been adopted into official introductory courses at the ACM SIGGRAPH conference (Angel and Shreiner, 2016; Angel and Haines, 2017; Angel, 2017). We describe and discuss these improvements, and also provide side by side comparisons to specific web demos accompanying Angel's textbook.

1.1 Contributions of the Thesis

In addition to the developed educational content and novel web-based framework for making more, this thesis offers the following benefits to the graphics, education, and research communities:

1. **A new code library for computer graphics, purpose-built for education.**

   Our tiny-graphics.js framework offers significant qualitative and quantitative improvements over previous libraries made for mainstream graphics courses.

2. **Novel tools for educational and research collaboration.**

   We provide a new hosting service for code. By changing the code in the provided interface, visitors to our website can build their own programs and articles, each of which is given a permanent sub-address on the website. Unlike other currently available online code remixing environments, ours allows programmers to interact with all other users' code submissions at once in one large open-source ecosystem.

   Under our "Smart Article" system, programs running on our website can also receive messages over the web. The messages can come from any source, including academic

research software made in any language. This provides opportunities, with small modifications, to bring more academic software projects to the web in the form of remote visualizations.

3. **Tools that assist teachers.**

   The website currently supports Computer Graphics courses with several lesson-specific interactive demos, replacing certain chalkboard-driven examples with more illustrative interactive games. The custom-made educational code library underlying the website easily supports new tutorials about math, programming, and graphics. Furthermore, it enables teachers or other experts to make and host their own demo-heavy tutorials covering an even broader range of topics.

4. **Educational benefits for visitors.**

   Our online demo and tutorial repository caters to many audiences. It helps students who want to make and share completed graphics course projects (Figure 1.1), researchers who want to make and share graphical demos that introduce peers to research topics, and hobbyists who want to make and share games and animations. It provides one-click code hosting (even anonymously) for all programmers making graphics demos and prototypes.

   Visitors lacking coding skills can still enjoy our website's educational and highly graphical articles submitted by users from the aforementioned groups, broken down by category and audience. Our current selection of articles introduce minor but useful new techniques to the graphics field at large. Compared to Wikipedia, where articles are mainly made of static text and images, the "Active Textbooks" on our service each include little programs that run inside the page and render an animation of the concept in action. As such, each educational illustration potentially becomes an interactive virtual toy.

5. **Smart Articles for consumption by both humans and computers.**

   Our concept of "Smart Articles" is an innovative application of grid computing for

Figure 1.1: These animation projects were created by students using our tiny-graphics.js library during offerings of the UCLA course *CS 174A: Introduction to Computer Graphics*. They are hosted by Professor Terzopoulos on the course page (Terzopoulos, 2017). The library has been an effective educational tool, both for easing the students into learning graphics programming, and for allowing their programs to mathematically achieve visual results like these, which was a motivating force in conceptualizing and implementing course projects.

research.[1] When computer programs access Smart Articles from our server, they can automatically inherit some capabilities on the article's topic. Programs can do this by delegating specialized work for the Smart Article to handle, which is then queued up for the next online visitor to the article. Once a visitor loads that page, the pending work is then executed as a function call on that visitor's computer, prior to the execution or display of any demos embedded on the requested page. The task is thus completed using crowd-sourced computing power along with the specialized capabilities and API of the particular Smart Article in question.

Our website supports lessons about Computer Science topics, which include computationally hard (NP complete) problems. Our web server is capable of distributing the load of any computationally hard demos that are embedded in its articles. The Smart Article paradigm can divide problems into smaller chunks (sub-problems) and send them to each visitor's machine as they view the article, thus applying volunteered CPU time from these several sources to solve the larger problem. Two Smart Articles that are each fully specialized for some graphics-related task can, in parallel, help each other to solve a problem that is larger than their individual topics, by delegating to one another; they do so in separate, easily tracked processes and browser windows.

1.2 Dissertation Outline

Anatomically speaking, this thesis is comprised of two interwoven sub-projects—a JavaScript programming library, which is developed in Chapter 2, and an online encyclopedia website full of examples and tutorials that use the library, which is developed in Chapter 3. The backgrounds and motivations for each of these two components are provided in Sections 2.1 and Section 3.1, which precede the main descriptions of the sub-projects in Section 2.2 and Section 3.2, respectively. These background discussions are broken down by sub-project to explain the open problems from which each project arose.

[1] Unlike prior volunteer-based projects known for mass grid computing, such as Folding@Home (Larson et al., 2009), our system can include arbitrary new research problems submitted by users.

Chapter 4 presents the results of our research, including a case study at UCLA. It also documents our success at using our library to produce web demos that are at least as effective as those associated with Angel's textbook (Angel and Shreiner, 2014a), including side by side comparisons.

Chapter 5 provides a review of similar projects and the diverse academic research upon which our multi-disciplinary project builds, referring to the details of our design from the earlier chapters. We show that we offer a unique intersection of features that stands out among related websites and similar documented projects. Different facets of our project relate to collaborative software hosting (such as GitHub (Dabbish et al., 2012)), online encyclopedias (such as Wikipedia (Selwyn and Gorard, 2016)), frameworks for making advanced WebGL applications (such as three.js (Dirksen, 2013)), and industrial tools for graphical effects (such as Maya and Unity (Govil-Pai, 2006; Labschütz et al., 2011)). Several related projects may be categorized as education in Computer Science and Mathematics, the development of educational web demos and visualizations, and Digital Game Based Learning.

We conclude the thesis in Chapter 6 with a discussion of future plans and research directions.

CHAPTER 2

The tiny-graphics.js Software Library

2.1 Motivation and Background

Making graphics programs requires knowledge and a significant up-front cost even to begin the initial setup. This much is clear from the sheer number of industrial tools that exist for setting up graphics (Hughes et al., 2014; Celes and Corson-Rikert, 1997). These serve not for providing any particular animation capabilities, but just to wrap and simplify basic graphics functionality—the act of projecting 3D triangles onto a 2D plane of pixels.

A student starting to learn graphics, whether in a typical C++ based course or not, must perform a surprising number of preliminary steps even to make the smallest graphics program from scratch, especially if they use correct up-to-date techniques. Many beginners are therefore tempted by tutorials on outdated approaches (Davidovi'c, 2014). They might find instructions for the coding paradigms of prior decades, before modern shader-based graphics cards changed everything. These "pre-shader" approaches were indeed easier to initially approach from scratch—and hence remain very high on internet search rankings—but they are no longer supported. For many years these commands have been removed from the default languages of graphics cards on the market, and they have never been supported on web browsers. Newcomers can avoid this trap only by committing themselves to a big up-front investment: learning the difficult setup steps that are expected of them in newer graphics programming systems.

In current approaches, these laborious initial setup steps are mandatory. A graphics beginner must use them to complete detailed tasks such as the following: 1. Connecting their program to the graphics card and populating the card's memory buffers with vertex

data, which describes points in shapes by their positions and other fields. 2. Managing pointers into that data, matching them appropriately to pointers into variables within the graphics card's separately-managed running code (the "shaders"). 3. Commanding the main program to build another distinct program of correctly-made shader code, to be used by the graphics card for each drawing operation. The latter step is the worst of all; shader code is its own new language and paradigm a student must understand. All of these steps are required before anything at all can be drawn (Angel and Shreiner, 2014a).

All programs that draw 3D graphics must express the above steps using calls to the computer's graphics card (Angel and Shreiner, 2014a), which always exposes built-in functions to the programmer for those very steps. Common interfaces for this are called DirectX and OpenGL, the latter being more widely available and more prevalent in education (Angel and Shreiner, 2014a). The phrase "OpenGL program" often implies the language C++ (Deepak, 2015) due to the nearly universal hold C++ had on graphics education until recently, but this is a misnomer; other languages, including Python, Java, and JavaScript can make OpenGL calls too. All choices cause the same triangle-drawing 3D effects to be executed on the display window. Because of this connection, the act of 3D graphics programming in one language can also be familiar in all the others.

We use WebGL, a form of OpenGL. WebGL is merely the name for JavaScript code that includes OpenGL calls. JavaScript is the language of the web, being the only programming language that works inside of websites in modern browsers. Aside from the change of language to JavaScript, WebGL graphics programming is nearly exactly the same as its more traditional C++ counterpart, using the same API calls in a nearly one-to-one correspondence. For example, any C++ command such as:

```
glBindTexture( GL_TEXTURE_2D, id );
```

which does something in OpenGL, becomes a JavaScript command like:

```
gl.bindTexture( gl.TEXTURE_2D, id );
```

when using WebGL.

Beginners should become comfortable with all three programming styles, imperative, object-oriented, and functional, to gain the best understanding of JavaScript in order to use

our library. To avoid missing newer capabilities, all inquires about JavaScript using online search engines should include the search term "es6", the current version name.

2.1.1 Graphics Libraries in Education

The official courses about introductory graphics at the ACM SIGGRAPH conference for the two most recent years (2016 and 2017) have been based on WebGL (Angel and Shreiner, 2016; Angel and Haines, 2017). These were organized by Edward Angel, the author of the most widely cited WebGL course textbook "Interactive Computer Graphics with WebGL" (Angel and Shreiner, 2014a). Angel's graphics textbooks have always included a small software library to get students up and running with hands-on programming experience while learning graphics.

Angel (2017) describes his rationale for moving from his helpful C++ based libraries over to WebGL even after many years of having kept up with updates to OpenGL in C++. There he laments those updates as having progressively increased the difficult learning curve of graphics setup, citing issues of unsupported features in different student hardware and difficulty in setting up uniform C++ compiling environments for all students. Compared to C++, Angel concluded that WebGL has comparable performance to C++, plus the advantages of a standardized environment on all platforms (including phones). WebGL development is easier due to the combination of an interpreted code engine and surprisingly excellent debuggers that are built into the menus of modern web browsers.

Our work finds a number of additional benefits associated with the JavaScript language itself for graphics, due to the presence of functional programming and a powerful type system, and its tendency toward smaller total source code. The need for JavaScript to be interpreted live by other programs (browsers) creates performance limitations, but we do not find them overly restrictive in view of the advantages. WebAssembly (Haas et al., 2017) provides an opportunity to make calls from JavaScript to fast assembly code; it is supported by browser JavaScript APIs. It is also possible to avoid the slower parts of the JavaScript language by using only a subset of language features, either by hand-coding or by using cross compilers

that automatically do so—examples include Emscripten (Zakai, 2011), which translates C++ OpenGL code to WebGL that is only negligibly slower than the native C++ code.

JavaScript has benefits in both the early stages of designing prototypes (due to its functional and interpreted code) and in the late stages (due to options like WebAssembly), leaving few reasons to write code that cannot run in a web browser anymore so long as the application does not demand calls to graphics card features more cutting edge than what browsers at any given time support.

2.2 Design and Benefits of the tiny-graphics.js Software Library

2.2.1 Description of tiny-graphics.js

In WebGL a lot of unrevealing "boilerplate" code is required just to get a single 3D triangle to draw on a web canvas. It is not at all obvious, or even agreed upon (in online examples), how one should organize that repetitive code into functions. The function organization should be flexible enough later, when the programmer will need to dynamically switch out pieces of their program frequently—whether those pieces are other vertex arrays (shapes), other shader programs, textures (images), or entire scenes.

To offer that flexibility, we provide students with a single-file code library called tiny-graphics.js, which is very small (753 lines). It sets up the elements of WebGL, such as shaders and scenes. It also provides button and keyboard interfaces for enabling user input to trigger the aforementioned swapping of the program's components. The buttons could be any that are embedded on the web page from which our library is loaded. Crucially for graphics, our library also includes the machinery for matrix and vector algebra, which the JavaScript language specification lacks (Simpson, 2015).

2.2.1.1 Going Dependency-Free

To better serve as an educational tutorial, our library imports no outside code. It is normally used in programs of three short, human readable JavaScript files. By not importing outside

libraries, we avoid giving an overspecialized skill set to students, making them demystify extra layers of complex API, or risking the dependencies becoming obsolete.

It befits educational libraries to be self-contained and free of outside dependencies, since tutorials are meant to have a limited scope. In another example of this, Angel wisely chose to make his own WebGL helper library dependency-free. He explicitly renounced three.js, a framework for organizing a 3D animation and adding capabilities such as scene graphs (Angel, 2017), citing its abstractions as reasons why it does not fit into the scope of an introductory engineering course concerned with architecture and implementation.

Angel also omits the large, extensible libraries such as jQuery, Angular, and React that help with organizing JavaScript and websites, which to an engineering student would be an unwanted intermediary as they learn the fundamental code objects of the browser and page. Thousands of responses to common JavaScript questions online (Treude and Aniche, 2018) assume as a matter of course that jQuery will be imported into every JavaScript program and its shorthand used throughout. For a beginner, these extra "frameworks" for JavaScript simply clutter search results. If used, they add to the concepts in the learning curve, increasing complication and the amount of code visible to the program. With pure JavaScript a student benefits by learning fewer concepts from the language specification until they can go no deeper; they need not demystify hidden code that leaves room for misunderstandings when diagnosing errors.

JavaScript libraries are typically more than just a couple years old, made without the conveniences of the 2015 "es6" upgrade to JavaScript, so their code is more verbose than it could be. Using them might lock the programmer into the older style, or force them to learn features with overly custom implementations that are now unnecessary since they are universal in es6. Learning to do the same task the "es6 way", meaning from scratch in pure modern JavaScript, is thus a better long-term strategy anticipating es7's arrival, compared to becoming attached to libraries.

Lastly, our dependency-free pure JavaScript approach is a relief from an endemic problem in the current JavaScript community—the dependence on trivial packages that do the

same thing a few lines of basic JavaScript could do. This phenomenon was researched and quantified by Abdalkareem et al. (2017), who found wide acceptance of this practice. They discovered that one sixth of all JavaScript programs imported by package managers are trivial, and nearly half of those import their own dependencies. They explain how over-reliance on trivial packages caused the famous "left-pad" incident that briefly brought down some of the largest internet services like Facebook, Netflix, and AirBnB. This happened when a file called "left-pad" was briefly unpublished, which contained only a dozen lines of JavaScript that performed a trivial padding of spaces onto the left end of strings.

Trivial dependencies certainly pose a problem. Our library represents one effort to counterbalance the widespread over-reliance on dependencies, getting back to the roots of coding using the rich features already built into the JavaScript language itself.

2.2.2 Improvements Over Existing Graphics Tutorial Software

As a baseline of comparison, we will use the WebGL code from Angel's most recent textbook (Angel and Shreiner, 2014a). That is not to say that tiny-graphics.js is merely an incremental improvement over Angel's library; we argue that the scope of tiny-graphics.js is much larger and that it stands on its own. Angel's library is almost entirely geared towards matrix and vector algebra, offering little help in organizing the WebGL program. Our library emphasizes both, allowing users to better compartmentalize demos. Our demos stay small, yet they are able to be more compelling and go deeper into the material than the rather bare-bones web demos Angel provides in his textbook's online resources, as we will show in our comparison in Chapter 4. In Section 3.2 we will show how to use our library to more effectively support a real-world, UCLA introductory computer graphics course by replacing specific lessons in the syllabus. Our example could potentially improve all graphics courses around the world, not just those that already use Angel's textbook, so our library and Angel's are not comparable in scope.

We mainly compare against Angel's library since the associated textbook and ACM SIGGRAPH course are mainstream, and widely accepted within the SIGGRAPH community

as the right way to start learning graphics. It is an especially apt comparison since we were able to observe directly through UCLA's own graphics course the improvements tiny-graphics.js makes over Angel's library. These improvements specifically are as follows:

1. **Reduced Clutter for Clearer Examples**

   When students start coding in a graphics framework that is designed to be a tutorial for them, they expect immediately to see concepts they have learned in their graphics course lectures—math concepts such as matrix algebra, points, and projections are the norm in UCLA's introductory graphics course. Students imagine building their first scene and expect to see that the three special matrices they have been taught are for precisely that purpose—translation, scaling, and rotation. But students who use Angel's framework (or most WebGL tutorials) are often faced with finding these familiar math concepts scattered between large sections of WebGL boilerplate clutter, hidden among a lot of unfamiliar words not related to anything learned in lecture. This clutter contains the previously mentioned laborious commands to the graphics card and initial setup steps of shaders and buffers. This is highly predictable code that could instead have been relegated into reusable subroutines set apart from the math operations that are of interest to the student.

   Students who code using our tiny-graphics.js framework will instead be relieved to find that their scene's definition is uncluttered by any boilerplate graphics code or any maintenance calls, providing a dramatic gain over Angel's online textbook demos. If a beginner merely wishes to read one of our existing demos, they will need almost nothing other than an awareness of what a transformation matrix is, and they will immediately recognize conceptual words from their graphics course, including the three special matrices. Armed with this understanding and the ability to read basic JavaScript, viewing the source code of any scene quickly reveals how these special matrices are generated by functions in the code to draw the shapes in the desired positions in the 3D world. That is because the most useful code for students to read, matrix operations juxtaposed with one-line draw calls, stands by itself.

2. **Superior Performance**

We extract more out of JavaScript than Angel did by using more language features. Angel restricted himself to language features that resemble those available in C++. In addition to that limitation, his code also encounters some of JavaScript's pitfalls—ways to code that are deceptively expensive. Built-in browser profiling shows that his program's overall execution time is dominated by just one function, an expensive helper routine that operates upon the arguments of every function call, which is designed to compensate for how Angel uses those arguments. Modern es6 (2015) JavaScript provides a single token in the language (the Rest or Spread operator) that accomplishes the same thing without growing the call stack with another function call. Our library uses this instead, and this is supported by all current browsers.

Additionally, the most performance-critical math functions in Angel's code are hobbled by programming too defensively against the possibility of receiving the wrong input types or amounts, perhaps for fear that students will make mistakes in JavaScript's dynamic type system. As new programmers, however, students do not have the expectation of type safety that comes from a career of C++ programming. Quoting Angel himself, "To them, JavaScript at worst is 'just another language,' and they learn it quickly" (Angel, 2017), which is precisely why students would not be as blindsided by unsafe typing as Angel's own defensive coding presumes. This practice (even including unreachable code paths) was perhaps adopted before Angel fully appreciated how well the powerful in-browser debugger, which he praises in (Angel, 2017), compensates for these risks. Some of the checks are also replaceable by, again, JavaScript's newer Spread and Rest syntax.

Angel (2017) describes the flexibility of JavaScript array types in allowing students to write clearer code. He argues against using es6's fixed-size TypedArrays. However, TypedArrays are in fact capable of anything that his preferred regular arrays can do, including the es6 iterator-based array functions, which are the most helpful of all. Although TypedArrays cannot change their length, most math vectors should be fixed-length, since the math itself does not require that their lengths change, and since

many of them will just be temporaries generated within JavaScript expressions.

Our library uses TypedArrays. This change alone enabled our students to implement their own ray tracers that execute with around five times the performance compared to identical versions that do not use TypedArrays.

We provide a way to repeatably verify this result. Simply navigate your browser to our demo at `https://encyclopediaofcode.glitch.me/Ray_Tracer_Performance`, use the navigator to select the class `Vec`, open the editor, and change the superclass of class `Vec` from `Float32Array` to `Array`. Select "Run With Changes," and observe that the difference in speed of completing the image is indeed about five times.

3. **Simpler Code Due to Better Language Features**

Angel's code library contains several files, the largest being MV.js with 979 lines of source code. This file contains all of Angel's matrix and vector algebra definitions, including a complex routine for computing the 4×4 matrix inverse. In tiny-graphics.js, all of the same matrix and vector functionality is contained within just 129 lines of code, with the remainder of our file being open for other tasks such as organizing the user's WebGL calls. This brevity is achieved by taking advantage of up-to-date es6 (2015) JavaScript features (such as array iterator functions, spread and rest syntax, and arrow functions). Unlike Angel's library, which was ported over from C++ and retained the same style and paradigms, we take advantage of JavaScript's functional language capabilities to express substantially more with the same amount of code.

Despite a tremendous reduction in code complexity, our matrix and vector algebra classes (`Mat` and `Vec`) are in fact more flexible and capable than Angel's versions, generalizing to $M \times N$ matrices and any-size vectors instead of a few hard-coded sizes, while maintaining special-case small vectors that optimize performance just as Angel does, and while typing them as flat buffers whenever certain WebGL calls demand. Functions meant specifically for the 4×4 matrices common in graphics are broken off into their own `Mat4` subclass, while other matrices can assume any size. The code for the `Vec`, `Mat`, and `Mat4` classes each fits within about the size of one screen that can be

perused by the reader all at once without scrolling. Code should be written concisely enough that it can be economically kept in one's head (see `http://paulgraham.com/head.html`); it is easiest to memorize and visualize code when it fits into view on-screen in its entirety.

A newer es6 language feature, called template strings, provides another gain, allowing us to clean up our HTML code. Any extra programs comprised of shader code are instead embedded as simple strings within our JavaScript code. It is considered good practice for HTML documents not to be contaminated with code (Yu et al., 2007); among other benefits, this achieves a more secure separation of code and data (Doupé et al., 2013).

We can organize all our shader language (GLSL) code into appropriate specialized classes for each type of shader, rather than intrusively cluttering the HTML document file with every shader's code, as Angel unfortunately had to do (Angel, 2017). By comparison, the HTML file provided with our library is nearly empty, containing only a couple commands to place panels of our content anywhere. The remainder is left for the student to fill in, and no fixed layout or lengthy content is required.

Lastly, we take increased advantage of the exceptions featured in JavaScript by using them to display messages on the page rather than in the console as Angel does. We wrap all user code routines in "try/catch" blocks. By catching exceptions, the messages we can display are more informative than the default failure mode of graphics applications—the dreaded blank screen.

4. **Fully Object-Oriented (Infix Notation for Math)**

In the past, Angel's libraries were object-oriented, with classes for vector and matrix types. These defined helpful overloaded math operators, which allowed the use of familiar algebraic syntax like "u + v" and "u - v" in long math expressions. Unfortunately, in the switch to JavaScript, Angel adopted a style that is more common to JavaScript tutorials. These tutorials (such as the Mozilla Web Docs (chrisdavidmills and other contributors, 2018)) typically avoid discussing Object-Oriented constructs

because of how JavaScript handles them in a conceptually difficult way for audiences with a C++ background, who must first become familiar with JavaScript concepts such as prototypes, closures, and execution contexts. Instead, they promote source code where functions exist as a laundry list in global scope, without classes to keep them organized.

Without classes or overloaded operators, Angel's library painfully uses prefix notation for all math operations, such as `mult( a, b )` for two matrices or `plus( a, b )` for two vectors. Longer expressions made of these functions are no longer intuitively math-like, and worse, result in progressively deeper nested parenthesis in long expressions, since both operands in a pair are nested. Compare the following two code snippets that multiply four matrices, using prefix and infix, respectively:

```
mult( M, mult( N, mult( O, mult( P ) ) ) )
M.times( N ).times( O ).times( P )
```

Notice in the first how the reader is forced mentally to keep track of deeper parenthesis nesting levels and then parse four closing parenthesis at once.

We avoid prefix notation because we utilize classes in JavaScript. Only newer JavaScript tutorials (Sengstacke, 2016; Mott, 2018) mention the "class" keyword brought by the release of es6 in 2015, but it dramatically simplified the language's syntax for object-oriented code. While possible before, going object-oriented no longer requires an intimate understanding of advanced language features such as prototypes. For our need to express math operations, operator overloading itself has never been permitted in JavaScript, but the ability to use object-oriented styles for vector and matrix types provides a workaround; we can achieve infix notation using member functions. A vector object has class methods such as "plus" and "dot" that it can perform upon another vector. The calling vector operand is not nested, and the operator goes between, which creates far more readable math expressions that are easier to paraphrase in terms of math sentences.

Being object-oriented also helps students who attempt to read and familiarize themselves with tiny-graphics.js. All the code is contained inside fourteen classes (seventeen

if you count aliases), and there is no stray code outside of these. Each class can be read sequentially by clicking the code navigator that each demo automatically generates and comes with. Students can be confident that they have read all the code that is affecting their program and that there is nothing hidden that requires demystification and/or that may cause problems later.

5. **Text Rendering**

   The ability to render text onto the canvas and embed it in the 3D world onto objects has been conspicuously absent in Angel's library ever since the move to OpenGL 2.0, with programmable shaders deprecating the built-in functions for rasterizing text. Drawing text is non-trivial in WebGL because there are no built-in drawing functions at all, aside from those for drawing shapes made of points, line segments, or triangles. The additional examples we provide for our library in Section 3.2.3.17 bring text rendering capability back for modern graphics cards.

6. **Improved Shape Drawing Functionality**

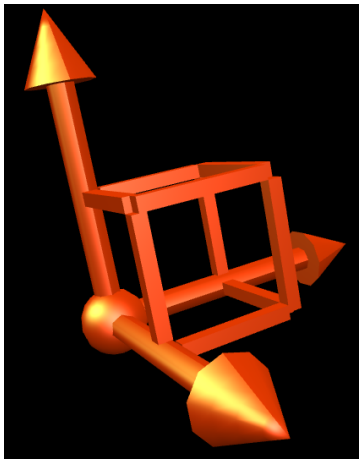
   Any graphics program's primary job is to be good at drawing shapes. Prior to the availability of modern graphics cards with fully custom shapes and programmable shading algorithms, OpenGL applications normally used plugins, such as GLUT, with premade routines for drawing some built in primitive shapes, such as cubes and cylinders. GLUT has long disappeared in favor of the programmable graphics cards, but to this day many tutorials have not adapted well to the absence of its shape drawing functions. Angel's entire WebGL textbook offers helper functions for drawing only a cube, a (subdivision) sphere, and the Utah teapot. Although many hints are given (such as a chapter on spline patches), the various remaining shapes are left as an exercise to the reader. Additional code for a cone and cylinder were found in an adaptation made for UCLA's graphics class when the author began as a teaching assistant; this shows that these simple primitives were easy enough to code from scratch and insert into buffers, but any more unusual shape proved too complex for the paradigms found in a beginner tutorial such as Angel's code.

When configuring shapes, Angel's library is once again held back by its demos' adherence to C++ styles even in JavaScript, ignoring the language's capabilities in functional programming. This includes JavaScript's anonymous functions, which we exploit to create a powerful new way to automatically build shapes as sets of coordinate points. Our method allows extra functions to be passed into a new shape's declaration, which provide deeper descriptions of what to do during the shape generation process.

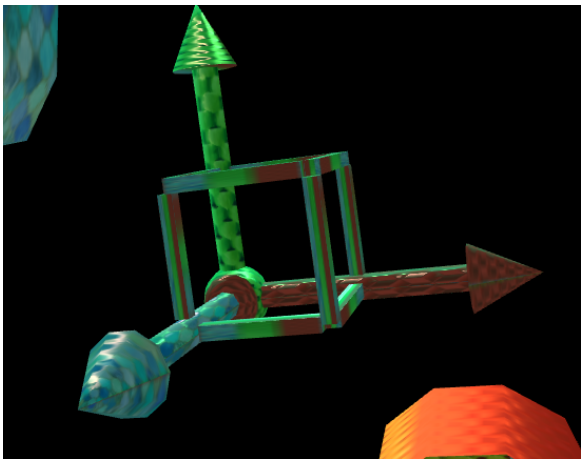
These anonymous functions are fully customizable code blocks, which are more flexible than passing simple flags to direct the shape modelling process. Anonymous functions exist for one-time usage, declared and defined in-line within expressions. Furthermore, one of the new JavaScript capabilities since 2015 is the arrow function, which is the same thing but more concise—arrow definitions provide access to the surrounding object's data fields, and the surrounding function's local temporaries, without needing to type out parameters. With these, very few characters need to be typed in order to pause mid-expression and declare a useful function. Passing these tiny but highly functional units as arguments into our main shape generator function greatly extends its flexibility.

In Section 3.2.3.8 we discuss how this programming practice helped us to make a better Shapes demo on top of tiny-graphics.js. With the help of arrow functions, our demo allows arbitrary operations—matrix transforms or otherwise—to be applied during the shape generation process to generate one point from another. The example we use is a triangulated sheet, which the programmer may deform into a more general surface patch. The sheet's points are arranged in rows and columns, and a tessellation of triangles connects these points by generating a certain predictable pattern of indices. The class produces a deformed grid by executing user-defined steps to reach the next row or column, defined by two anonymous-function callbacks supplied from outside.

Helper classes of only a couple pages of code allow our library to generate an infinite variety of arbitrary surface patches, surfaces of revolution, and other shapes. This allows not only for the traditional shapes such as spheres, domes, cones, polygons and other surfaces of revolution to be formed, but also any sort of surface patch. In our



(a) Phong shaded to show the composition.



(b) Color coded to distinguish the axes.

Figure 2.1: The axis arrows compound shape, defined in only 17 lines of JavaScript. It reuses many simpler shapes (including cylinders and cones) that were already defined. Copies of the vertex array data of those shapes are contiguously listed inside the axis's own vertex array. Since we provide helper functions that perform this copy, unique compound shapes require little code to specify.

examples these include a sinusoidal egg crate surface, a spiral seashell, and much more general surface patches such as the one shown in Figure 3.19.

Regarding shapes, tiny-graphics.js has more unique built-in features. Compared to Angel's library, we allow shapes to be managed in a more performance-friendly way, by providing operators to combine them. Our library's JavaScript class `Shape` neatly provides the ability to compound multiple defined shapes into a single combined vertex array. We thus eliminate much duplicated code students would normally need to provide when trying to pack complex multi-part shapes into a single performance-friendly buffer. A student can perform a single function call in any Shape definition to insert other defined shapes into the current array, at custom affine transform offsets. Positions and normal vectors are automatically adjusted by the affine transform during insertion.

Figure 2.1 shows an example of a complex drawing of coordinate axes. It was built by compounding several other simpler shapes.

We additionally provide a short routine that automatically converts any user-made

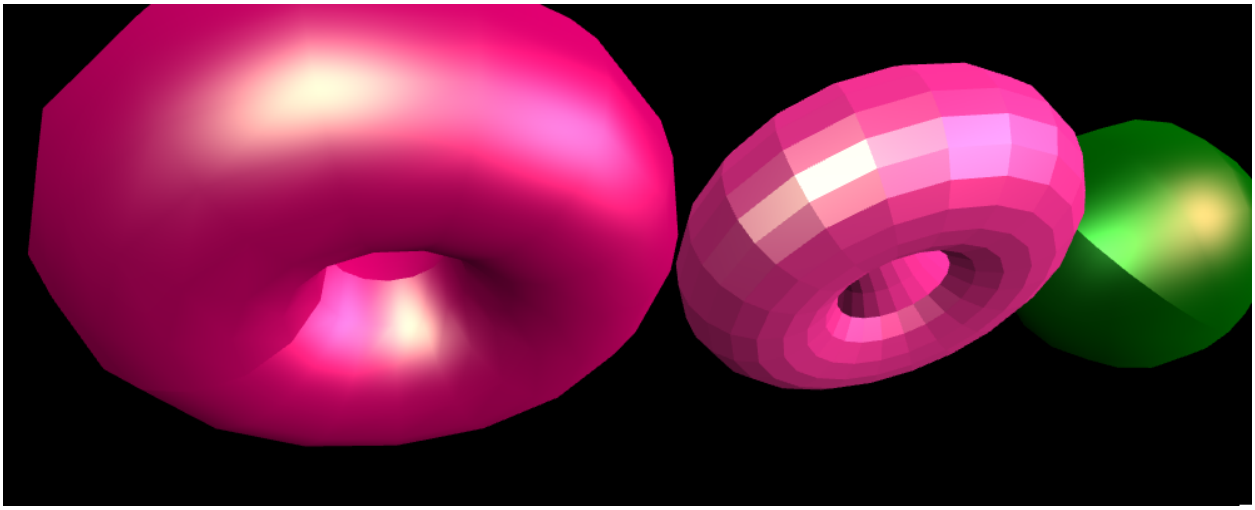


Figure 2.2: The middle shape, a torus with many triangles, has undergone the automatic flat shading capability of our class Shape.

`Shape` class into a flat shaded version by computing new normal vectors and automatically eliminating inter-triangle sharing of vertices so that data (such as texture images) can abruptly vary over triangle edges. Figure 2.2 presents an example of the visual change before and after this routine is called on a `Shape` that has a large vertex array.

2.2.3 Details of tiny-graphics.js

The file tiny-graphics.js contains JavaScript code organized into classes—just over a dozen of them, each averaging about a page of source code. Their relationships are as follows.

Two classes `Vec` and `Mat` introduce vector and matrix algebra lacking in JavaScript. These are standalone and could generally benefit all JavaScript programs. The next class `Mat4` builds upon the matrix algebra functionality a little more by adding functions specialized for the sorts of fixed-size 4×4 matrices common in computer graphics, mirroring Angel's examples. Together, `Vec`, `Mat`, and `Mat4` serve to completely replace Angel's core helper code library for graphics math, with the improvements discussed in Items 2, 3, and 4 listed in the previous section.

The next two classes in our file, `Keyboard_Manager` and `Code_Manager`, also stand on their own and each may be greatly useful to JavaScript programs in general. Each replicates

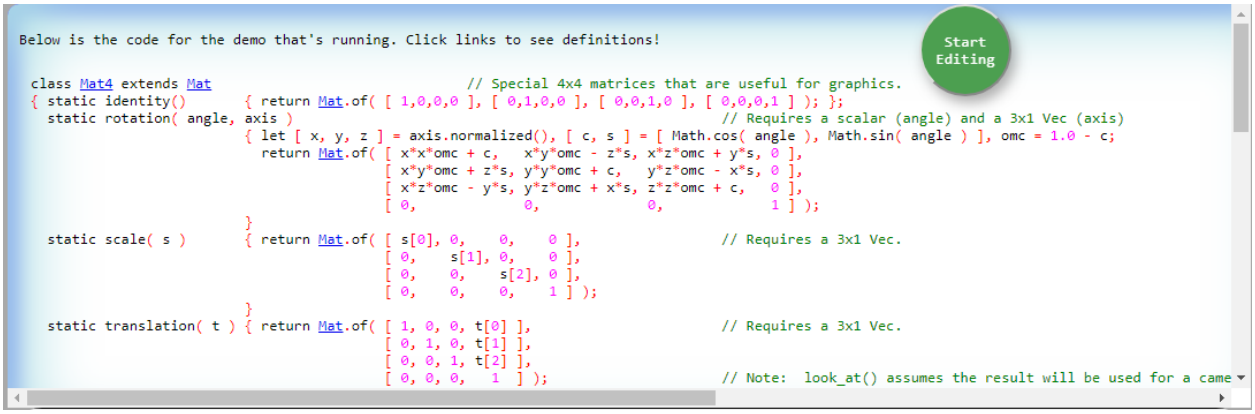


Figure 2.3: A code editor widget made of the Code_Manager class. Clickable links are automatically embedded wherever the names of other JavaScript classes appear. This allows the visitor to navigate tiny-graphics.js as well as the remainder of any program that uses it.

the functionality of some popular code library available on the web.

The `Keyboard_Manager` class was loosely based on the "shortcut.js" library by Binny V A (Abraham, 2012), strictly increasing its capabilities, but with far fewer code instructions (by leveraging newer JavaScript features). `Keyboard_Manager` adds keyboard interactions and shortcuts to any website, which is especially useful for interactive editors and games. It solves the nontrivial problem of tracking combinations of keys the user holds down, including modifier keys, function keys, arrows, and symbols. Designers of interactive applications often need to bind increasingly unusual keys and key combinations to their program so that the end user can distinguish between them (and the built-in browser shortcuts that work on every page) without confusion.

The `Code_Manager` class replaces a small library called "js-tokens" by Lydell (2016), which mostly defines a very complex regular expression Lydell built for parsing the JavaScript language. We re-use his expression to give our class the ability to break a string of JavaScript code down into tokens categorized according to whether they are comments, strings, numbers, spaces, identifier names, or other allowed language constructs. Our class gives JavaScript programs the ability to present JavaScript code effectively in interactive editors. We use it inside editors shown in our demos to highlight the displayed code informatively with colors, and to replace class names with links to the definition of that class. This allows the users of our editor to navigate through and understand the very classes we are describing in this

section, plus all other classes outside of tiny-graphics.js used by the demo they are viewing.

The next set of three classes contain nearly all the JavaScript program's WebGL commands. These are called `Vertex_Buffer`, `Shader`, and `Texture`. They provide a code interface for communicating data over to the graphics card to prepare a WebGL program. Each encapsulates the sorts of data its name suggests, which in a graphics program are defined as follows:

- Every shape definition needs one vertex buffer. A vertex buffer is a block of data divided into equal sized chunks, each mapping arbitrary data onto one point (of the shape). This data is numeric, such as the point's position along one of the axes, or perhaps the point's color intensity in one of the three color components.

- A shader is a computer program written for the GPU to execute every time it performs a shape drawing operation. In WebGL, there are two possible operations (computing a vertex or computing a fragment). The vertex operation decides where to draw points in their final on-screen positions, and the fragment operation decides how to color them in or paint the regions between them. Figure 2.4 summarizes shaders.

- A texture is field of color values, like an image file, to be consulted during a fragment operation to project an image onto a 3D shape's exterior.

A fourth class in our file, `Shape`, extends `Vertex_Buffer`'s functionality beyond just holding data to add some geometry tools, since the data being defined is usually geometric in nature (a single shape). In practice, normally `Shape` is used instead of `Vertex_Buffer`.

These three base types, `Vertex_Buffer`, `Shader`, and `Texture`, can boost a standard JavaScript program significantly, although they are not completely standalone—
`Vertex_Buffer` and `Shader` are interdependent, and depend internally on another class in our file called `Graphics_Addresses` made just for them. They also require the user to subclass them in order to fill in desired behavior.

The pieces of code for these base types are modular enough that they can be understood in isolation. Rather than merely hiding the WebGL calls students need to learn to make

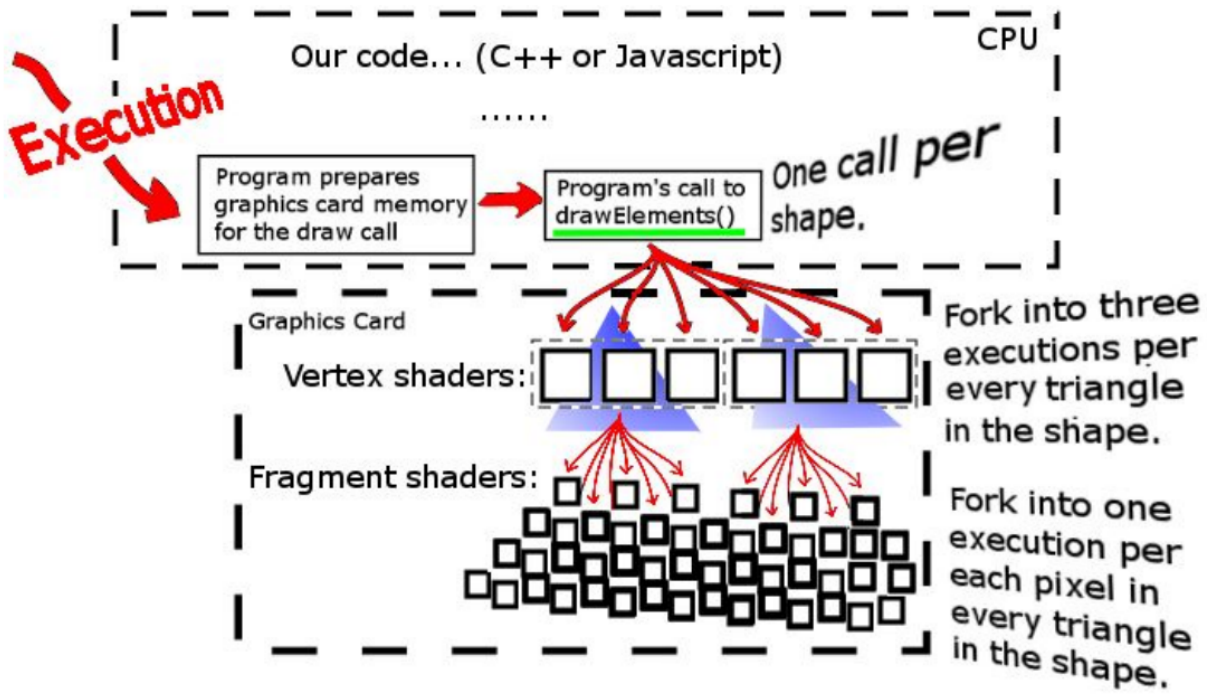


Figure 2.4: A summary of any graphics program's flow as it forks into many programs (called shaders) that run in the GPU when drawing each shape.

behind extra layers, these small classes in a thin layer show the correct grouping and ordering of WebGL calls, factoring out the logic that stays the same every time. This logic was worked out with experience, making sure that the exposed functions are appropriate for a variety of use cases when building our own demos with it.

Two more classes, `Scene_Component` and `Webgl_Manager`, go hand in hand. One `Webgl_Manager` is instantiated for each 2D HTML drawing canvas that appears on the page, and one or more `Scene_Component`s are assigned to it to define what to draw on the canvas.

Getting WebGL to start up initially means the `Webgl_Manager` must instantiate an HTML5 Canvas object, open up a WebGL context on it, and organize everything related to that 3D context. `Webgl_Manager` organizes the event loop of the 3D drawing and interaction area, and it stores what scene(s) to draw, what shapes it is made of, and which textures and shader programs need to referenced—in other words, it stores instances of the aforementioned `Vertex_Buffer`, `Shader`, and `Texture` classes. If their data has not yet been sent to the GPU, it does so, including compiling shaders, and then collects the correct pointers to

them in the GPU matched to pointers in RAM.

Unlike how most "first WebGL program" tutorials are organized, this pair of classes shows the proper way to organize a WebGL program that contains *multiple* independent canvases and multiple possible shaders and textures that all co-exist interchangeably without stepping on one another. Historically, this is generally *not* something beginning WebGL students have managed to achieve in UCLA's introductory computer graphics course prior to the introduction of our library. This is because WebGL constructs such as shaders, textures, and shapes can require dozens of lines of code to toggle on and off; there is no built-in way in WebGL to concisely describe a scene that switches between many of these things.

Finally, tiny-graphics.js ends with the definition of our "Widget" classes, which will be explained in Section 3.2. These determine the layout of the web page the visitor sees when launching the JavaScript program. They can be used by the programmer to embed custom interactive panel elements (widgets) into the page on which they are working that also holds the WebGL animation. These panels include the canvases that show 3D scenes, text, and buttons. The contents of these classes are not crucial to understanding WebGL and can be ignored by a beginner. They are nonetheless beneficial to beginners who decide to use the coding interface of our programming website, which showcases demos made using the tiny-graphics.js library. The purpose, functionality, and specific page layout of this website is the subject of Section 3.2.

2.2.4 Using tiny-graphics.js

To use tiny-graphics.js, a JavaScript program needs to provide, at minimum, a subclass of `Shape`, a subclass of `Shader`, and a subclass of `Scene_Component`. That is all any program must provide. Rather than implementing all three of these from scratch, a user will typically start with three pre-made ones, and then customize one or more of them. Particularly, all new demos involve making a custom `Scene_Component` subclass, even a simple one.

Here are the specific steps a student follows: They decide on a name for their `Scene_Component` and register that name in their .html web page file so that their ob-

ject gets automatically instantiated. In a blank file, they begin coding a new subclass of `Scene_Component`, naming it that; they can re-use an example and make adjustments later. Then, with no more than a couple lines of code each, they give their subclass a `constructor()` method that lists the class variables, the camera matrix, and the shapes that will be used. They give their class another method called `display()` into which the student need only start typing familiar words such as "translation", "rotation", and "times" to manipulate a matrix variable into place. The actual drawing is accomplished by one-line calls to the `draw()` functions of each shape. No other boilerplate code is necessary to distract students from this main thought process—one of "manipulate matrix, draw, repeat."

2.2.4.1 Comparison to Using angel.js

Although Angel's library is small and easy to understand, it is comparatively much easier for students using tiny-graphics.js to see what sort of code to type in order to see immediate drawing effects from any JavaScript file that imports tiny-graphics.js but is otherwise blank. In Angel's library it is harder to start adding code with predictable drawing effects, due to distracting code not related to the concepts from their course lecture.

To begin with Angel's code, a student must include all of Angel's files plus their own blank .js file, and then supply a "render" function that must call itself in an infinite loop using the browser's event queue. Because of the inflexibility of this design—a single function that infinitely repeats itself—there is no analogy to the object-oriented tiny-graphics.js system, where multiple scenes can be added, removed, and rendered with reliable timing.

Inside Angel's `render()`, no matter what scene a student draws, a lot of repetitive boilerplate code will be necessary. Computing different matrix products as `render()` progresses through each shape is the easy part, due to Angel's helpful matrix library, MV.js. Unfortunately, however, between those lines of familiar math code, students must add much more. Drawing each individual shape involves a complex selection of active and inactive GPU buffers, sending messages to the GPU to turn those buffers on and off to select one of their list of shapes (of which Angel only demonstrates a couple), managing pointers to GPU

variables to move the matrix and shape's color over, a lot of boilerplate code to turn on and load a texture, calling a special file to initialize shaders that clutter the user's HTML file and, lastly, calling the low-level WebGL function `drawElements` (or `drawArrays`) to get a cube or sphere or custom shape to appear.

This is to say nothing of the long setup commands students must add before calling their render function to send their shapes, shaders, and textures over to the GPU in the first place, which in Angel's demos does not at all factor out the most reusable logic in the way our `Webgl_Manager` class does.

In general, for all the reasons mentioned earlier in this chapter, the experience of using our library is dramatically better for students compared to the inconvenience of relatively unstructured WebGL calls that themselves do not teach them much of relevance. When using tiny-graphics.js, the students observe how to design a flexible WebGL program that requires the least amount of effort to add shapes, textures, shaders, and scenes. They see a sampling of the most up-to-date JavaScript idioms used effectively in all the ways mentioned thus far.

CHAPTER 3

The Encyclopedia of Code

3.1 Motivation and Background

Characteristically, Introductory Computer Graphics is one of the most cross-disciplinary courses in an undergraduate Computer Science curriculum. Ambitious university students who do graphics course projects often wind up exploring math, physics, art, anatomy, and far more, depending on what they are modeling, animating, and rendering. At UCLA we observe students enrolling in *CS 174A: Introduction to Computer Graphics* from a similarly wide range of majors, likely out of an interest for its interdisciplinary applications.

By having a wider subject scope, a course in Computer Graphics stands out to students compared to those they previously completed in the same department. At UCLA for instance, the first Computer Science course *CS 31: Introduction to Computer Science*, has a clear beginning and end—it teaches a freshman how to code in C++ assuming zero background and, then, upon reaching the topics of object-oriented programming, algorithms, and data structures, it is time for the course to end, and those topics are saved for the subsequent course, *CS 32*. While graphics courses also have their usual progression through self-contained subjects, they have a far more blurry endpoint due to the course topic's inextricable relation to the graphics industry.

The final week(s) of a typical introductory graphics course cover the diverse special effects topics and applications explored by the industry. These industry techniques normally appear in the computer graphics research literature, making each of them fair game as "extra credit" topics students may add to a term project. Graphics courses end in a way that encourages students to branch off into different areas of the graphics research field, such as computer

animation, which is covered in depth in UCLA's course *CS 174C: Computer Animation*.

These circumstances at UCLA are what gave rise to this project. The Encyclopedia of Code began as a set of supplemental class materials about various graphics applications and techniques. This extra material for *CS 174A: Introduction to Computer Graphics* took the form of demos that ran within websites. Each separate web page of the supplemental material demonstrated an "advanced topic" in Computer Graphics that a student could integrate for extra credit.

Over time, our "advanced topics" demos covering common industry tricks (such as collision detection) were augmented by additional educational articles on graphics fundamentals (such as matrix transformations), and the "encyclopedia" began to outgrow its original purpose. Parts of the *174A* course itself were automated into educational games that were playable on the website, helping students visualize concepts from class, such as vector math, frustums, change of bases, and the most difficult concept from the exams—matrix ordering in long chains. This marked a transition of the resource into resembling the material of an online course.

3.1.1 Online Courses

Angel's recent online WebGL course, *Interactive Computer Graphics with WebGL*, presented on Coursera (Angel, 2017), is an extension of the published curriculum from Angel that our project sought to improve. As a Massive Open Online Course (MOOC), it showed signs of high global demand for the material; 14,500 students signaled interest, 5,500 began the course material, 2,500 remained through the first week, and 282 completed it.

WebGL is the technology that made a graphics course of this scale feasible, as teachers evaluated participants' finished WebGL applications using Courseras peer-grading methodology. Without WebGL and its ability to reliably run the same way in any browser, the different operating systems and environment setups of the individuals grading would have confounded the results. Online course platforms usually do try to deal with students having different hardware and runtime environments; Udacity courses have a built in Python

interpreter, and EdX courses have a built in MATLAB interpreter (Zubrycki and Granosik, 2017).

Bourdin (2016) evaluated Computer Graphics MOOCs in general by supplying their own students. They found a number of pitfalls, including poor availability—a total of only three MOOCs exist for Computer Graphics, found on the platforms Udacity, Coursera, and EdX. Their study also found misleading estimates of workloads on course descriptions, and a lack of formal unbiased evaluation of course outcome. Worst of all was an extremely low completion rate, while noting that it "has been known for ages, since Socrates, that one motivation for studying comes with the desire to please the teacher. When the teacher is not there, this motivation fails." They conclude that MOOCs inherently have serious problems, and call upon teachers to build a MOOC "as a complement" to graphics courses instead of as a complete replacement.

We believe The Encyclopedia of Code was designed just as they recommend, having based it on supplemental material after all—demos from traditional University courses. Our design also recaptures the Socratic desire of students to impress the teacher, since our visitors can submit their finished work to be seen by others, or even try to impress the moderators into listing the demo on the main page. We have a formal system in progress for escalating a submitted demo through the ranks towards being publicly listed.

3.1.1.1 Online Encyclopedias

No discussion of the nature of online courses would be complete without bringing up the role Wikipedia has on modern higher education. Since our work directly builds upon the concept, being an alternative online encyclopedia, Wikipedia will be discussed in Chapter 5.

Benefits for Teachers Anecdotally, the creation of supplemental material has had positive effects on the quality of our teaching. Some benefits of putting supplemental lessons online are obvious; it makes the teacher's job easier (or automates it entirely), and makes the material available to more people. But consider another benefit: It generally provides

an opportunity to finalize the course content in optimally presentable, written form, with better-worded explanations of even the minor topics. This has advantages; any experienced instructor might otherwise go off their memory of past lessons, in "autopilot". They might recall a great many examples that worked well in the past and wish to rapidly show them all—however minor or obscure—yet find when presenting them that they no longer succeed without the same execution. Memory is fallible. Finalized content comes with guarantees.

By publishing and finalizing their material, teachers are able to use what should perhaps be called "errorless teaching". This is meant as an analogue to so-called "errorless learning" techniques, such as when music students, for instance, slow down their practice enough to preclude the possibility of errors. Here we mean that the curriculum's pace of expansion, rather than the student's pace of action, should be sufficiently slow that there is zero risk of instructor errors. Errors in curriculum or presentation could be things like awkward transitions or demos that do not work the way the teacher was expecting. In our experience, when these minor stumbling blocks are encountered they cause a shift in student confidence with the material. It bleeds over into the next topic, obscures it too, and accumulates catastrophically.

It is better to prevent errors in the first place by presenting only that which is already perfect—a finalized, published online lesson being the ideal. Our work adds this to a graphics teacher's tool set for guiding both the teacher and student as the material is presented. The benefits are only further compounded by our taking the opportunity to integrate interactive games into our online course. These provide the student with specific yet dynamic visualizations, and can provide answers to their specific questions better than a static blackboard drawing made in haste.

3.1.2 Literate Programming

Donald Knuth did crucial early work on rigorous analysis of programming, and has formalized a large number of programming behaviors that remain relevant today. One of his valuable contributions was the interpretation of code as literature for human consumption. He coined

"Literate Programming" to mean the interleaving of a program's full source code with a textual article explaining the code's intentions and design (Knuth, 1984). A crucial aspect of literate programs is that the document can be compiled into a complete, runnable program. Another crucial aspect is that the source code is stored in the document and is not necessarily in its proper order—the reader sees it in the most instructive arrangement, and a compiler re-orders it for consumption by a machine.

Our online project uses something like Literate Programming by making articles and demos the same thing. Rather than our article being compiled into the demo program, the program both writes the article and produces the 3D demo. We often use such programs to build up our demos in parts, and interleave the article text that explains the building of each part. In those cases we include the 3D drawing canvases that show each intermediate result, as well as the code windows that show the source code so far. The tiny-graphics.js interface provides all of these parts as embeddable page widgets for mixture in any order. The following section describes how.

3.1.2.1 Active Textbooks

To contribute to the world of education in topics of math and Computer Science, we introduce a novel form of online textbooks called "Active Textbooks". These virtual textbooks include illustrative and interactive demos. These appear in place of what would otherwise be static diagrams in a normal textbook's discussion of graphics or math concepts.

Our website contains examples of these Active Textbooks, whether they are visual demonstrations about math concepts or tutorials about how to build a progression of increasingly complex shape buffers, increasingly complex shading formulas, or more. Each interactive illustration area appears in one of our 3D WebGL canvases. They all update according to the visitor's interactions with the web page so far, where the visitor can scroll from one to the next in an exploration of the different concepts in the material being illustrated.

Along with text, our interactive areas are sometimes interspersed with code as well. Whenever the topic is programming, our lessons include the code of the illustrative demo

itself. The fact that the code building the article also builds the demos makes any Active Textbooks about programming a very similar analogue to Literate Programming, while also augmented by the use of 3D visuals. Code editors placed in between the demos allow the visitor to interact with their code directly. These contain hyperlinks to navigate the code better, demonstrating another feature from Literate Programming.

We provide a few examples in Section 3.2 as proofs of concept. These are not just Literate Programming articles that happen to be about 3D computer graphics, but are reinvented by graphics into an entirely new form of interactive visual textbook.

We will use the phrase "Active Textbooks" for online textbooks with this particular form of extra automation. This phrase has some history. Google search reveals patents dating as far back as 1985 for active electronic textbooks (Malvino and Malvino, 1989). There is a very early precedent for animations in textbooks, including the 1993 CD-based textbook by Ross (1995), which was delivered with educational animations for asymptotic notation, recursion, simple data structures, sorting algorithms and their analysis, hashing, binary trees, red-black trees, minimum spanning trees, single-source shortest paths, Fibonacci heaps, Huffman encoding, dynamic programming, matrix multiplication, matrix inverse, convex hull, genetic algorithms, neural networks, and a few others.

One company currently markets a product called active textbooks at `https://activetextbook.com`, where they host a cloud service for textbook PDF files that have been annotated with media or simple UI widgets. But their solution does not go nearly as far as ours, which replaces the bulk of text and image content outright with WebGL games that walk through the same concepts.

3.2 Design and Benefits of the Encyclopedia of Code

3.2.1 Organization of the Online Resources

Visitors to the Encyclopedia of Code website are greeted with a number of tutorials and demo programs. With transparency and open-source code, we make it easier to understand

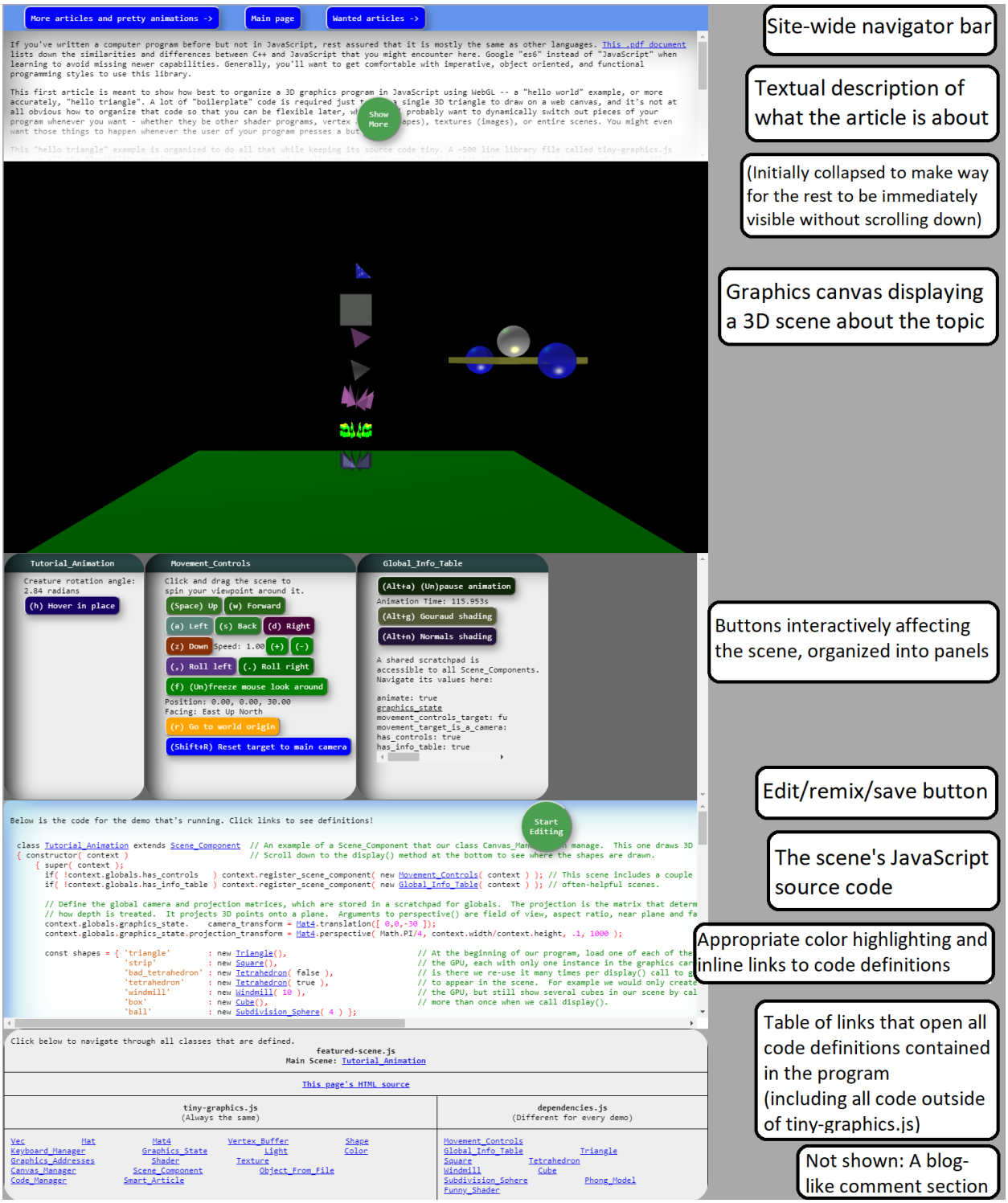


Figure 3.1: Upon visiting any article's URL, the visitor will see these features from top to bottom by default.

how the programs work. Once visitors are comfortable, they can write new programs directly into the interface and instantly post them to be listed online in a standardized way. Their contributed programs can be discovered by anyone and browsed within our larger educational encyclopedia. Their submitted programs also become available for anyone to edit and remix.

All demos on the encyclopedia are built using small pieces of code built on top of our tiny-graphics.js library. Unless otherwise mentioned, all demos (as well as their enclosing human-readable web article) are expressed in code as single small subclasses inheriting from our `Scene_Component` module defined in tiny-graphics.js. Most of our current scene definitions are about a page of source code, but some are a few pages, such as the more content-heavy "Active Textbook" articles that draw more than one demo.

The website's code and our tiny-graphics.js are both designed to be cross-browser and cross-platform. Special measures were taken to ensure that its 3D demos also show up properly on iPhone and Android devices. This increases the inclusivity and accessibility of our educational tutorials. It also opens up the possibility of collaborative coding using its editor while on the go, so developers are not tied to their desk.

The helpful tiny-graphics.js file is shared by all pages on the encyclopedia and can be seen when viewing the source code for a demo. Also seen there are two more files: One called main-scene.js containing only a single class definition that isolates only the code about the demo presently being viewed, and one last file called dependencies.js containing all the remaining code required by the demo that is not universal enough to go into tiny-graphics.js.

Our server does live dependency injection to provide every article in the encyclopedia to which someone navigates with a different custom copy of dependencies.js, containing only the minimal code that the viewed demo needs. These three files are sufficient to make each of the articles displayed on the website work.

Following a brief introduction, the current set of articles will be described. The current articles are of high quality, albeit few in number, and so far all are written by the same author. The project's goal is to grow quickly with enough crowd-sourced content to justify the word "encyclopedia" in its name.

3.2.1.1 Appearance of the Website

The encyclopedia website is shown in Figure 3.1. The encyclopedia can be accessed through its main page (currently located at `https://encyclopediaofcode.glitch.me`), which contains a portal to the articles, or it can be accessed by direct links to individual articles.

3.2.1.2 Definition: "Scene Component" / "Article" / "Demo"

These terms are interchangeable. Because we use something akin to Knuth's Literate Programming, our articles and demo programs are one and the same. Our articles are each named after the source code that writes them (always a single JavaScript class). This is why all articles have underscores in their names—they are the actual names of the code class that creates both the article and the demo. To be precise, all demos (and their enclosing articles) are built by instances of `Scene_Component` from tiny-graphics.js, particularly subclasses of `Scene_Component`.

3.2.1.3 Definition: "Widget"

Widgets are HTML panels containing some content from The Encyclopedia of Code, embeddable onto any website. Any of our JavaScript classes that generates one of these is a widget class.

3.2.1.4 Definition: "Text Widget"

When an instance of `Scene_Component` is viewed through a text widget, the text for an article is extracted from it and displayed. To use terminology from Literate Programming, this "weaves" the `Scene_Component`.

3.2.1.5 Definition: "Canvas Widget"

When an instance of `Scene_Component` is viewed through a canvas widget, a different set of its functions are called to interpret it as a 3D scene to build. To use terminology from

Literate Programming, this "tangles" the `Scene_Component`. Canvas widgets provide the scene with interactive panels of colorful buttons, made from standard web buttons, that provide the visitor control over the 3D animation.

3.2.1.6 Definition: "Live String"

The aforementioned panels of controls in a `Canvas_Widget` also contain continuously-updating text (live strings) in order to monitor the `Scene_Component`s attached to the 3D canvas. The page's event loop will constantly update all HTML elements made this way.

3.2.1.7 Definition: "Code Widget"

Code widgets are HTML panels that provide a special navigator for exploring all the source code available to the running program that the visitor is observing on our website. Our unique code viewer expresses the dependencies between code classes as hyperlinks inserted into the source code wherever one class mentions another.

The user may use the code widget panel to type edits into any currently selected JavaScript class; when they save their edit by pressing the "Run with Changes" button, the program replaces its existing class definition with whatever the user typed. The `Canvas_Widget` on the page immediately resets and re-draws itself, incorporating the user's modification into the 3D scene.

Code widgets allow anyone to save and host new articles. By changing the code in the provided code widget interface, visitors can build several of their own combined programs and articles, each of which is given a permanent sub-address on our website. They can use this to host their own animations. Once assigned a URL, their custom demo can be modified or built upon whenever they like using an authentication system that we designed.

In this way crowd-sourced articles can accumulate on our server. When functioning as a game engine, all code submitted to our platform collaboratively supports a single integrated virtual world, due to our server's shared code environment, single namespace, and code injection. When functioning as an encyclopedia, submitted articles can expand the tutorial

section of our website for the reference of students at universities.

3.2.1.8 Features of All the Articles

Demos have the ability to be displayed simultaneously with others, overlapping in the 3D space, by stringing together their names in the URL. For example, consider two articles hosted on our encyclopedia titled "Billiards" and "Star", which respectively contain demos that show how to code a 3D billiards game and how to code a miscellaneous special effect (resembling an exploding star). They are located at the URLs `http://encyclopediaofcode.glitch.me/Billiards` and `http://encyclopediaofcode.glitch.me/Star`. To view both their scenes together in the same 3D canvas, where the exploding star effect will actually be placed on top of the billiards table, the user can simply visit the URL `http://encyclopediaofcode.glitch.me/Billiards&Star`, which will be automatically parsed as a request for both scenes to appear simultaneously.

Recall that each of our web articles is named after the `Scene_Component` code class that builds it. These JavaScript class names are what are really being appended together in the URL. The server will automatically inject the combined set of source code dependencies of all `Scene_Component`s named together in the URL into the result the client (visitor) sees. The first `Scene_Component` that is named assumes control of the page's `Text_Widget`, but all of them appear in the page's `Canvas_Widget` together. In some cases, this merely results in the drawing of two 3D scenes together. However, in the case of "Secondary Scene Components", one demo might exist solely as an importable tool that many other demos can use by stacking functionality.

3.2.1.9 Definition: "Secondary Scene Components"

A category exists in our codebase of "scene components" that do not draw anything, but instead provide the final scene with extra abilities. They can be imported either from another scene's source code or requested by the visitor when they append the name of that class onto the end of another article's URL, causing our server to stack both components into one scene.

Each time one of these Secondary Scene Components is imported, they provide an extra tool so that the user can, for instance:

- Change the scene's appearance (such as by visualizing coordinate axes)
- Move objects or the camera (such as our `Movement_Controls` scene)
- Build or destroy a scene's 3D objects with a modeling tool (the case of our `Scene_Graph_Tool`)
- Read input (such as for user analytics, as in our `Keyboard_Demo`)
- Measure their scene by displaying measuring tools inside the same 3D space (example: our `Frustum_Tool` from Section 3.2.2.4)
- Open files, extract data, or more.

The buttons of these extra tools, such as camera controls, must not clutter up the web page's elements haphazardly, since it already contains the buttons added by the main `Scene_Component` being viewed. Our solution is to arrange buttons and alerts from individual scene components into separate panels organized per `Scene_Component`.

3.2.1.10 Definition: Smart Articles

The Smart Article concept works as follows: Certain articles on the encyclopedia compute answers to some programming problem during the course of their demo's execution, whenever a visitor (call them Visitor A) loads their URL. Once the answer is calculated, the article (call it the "sender" article) sends this final answer back to our server. Another article (call it the "receiver"), when visited at its URL (by some Visitor B), can read back from our server the answers that were computed by any sender article (such as during A's visit). The receiver uses that answer as a substitute for solving that same programming problem during its execution, as if delegating the work to the sender article and saving Visitor B computation time by offloading the work to Visitor A's CPU, who may or may not still be browsing the

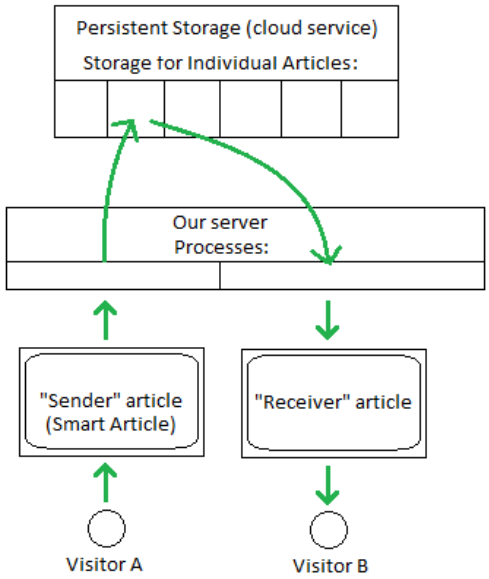


Figure 3.2: A diagram of the flow of information when a Smart Article is read by another article.

sender article. For this process to work, it merely requires any pair of people (or the same person) on earth to navigate to both URLs.

The sender article is what we coin a "Smart Article", an article with extra automation. The name is appropriate because these articles are suitable in some respect for consumption by a computer (the receiver article's code) as readily as for consumption by the human (Visitor A) who reads the textual and visual content of Article A when visiting.

Although simple, the concept has very powerful implications, allowing our executable encyclopedia articles to solve bigger problems than their original scope, greater than the sum of the group of articles involved, by delegating work. An example of grid computing, the concept increases in power as more people visit the page, volunteering their CPU cycles. Some potential applications might require both articles to be open concurrently for continuous communication; others do not.

In our current examples online and under development, the relationship between "sender" and "receiver" is often flexible. Two articles might alternate in their roles as "sender" and "receiver". Perhaps both read and write to a shared scratchpad on the server (a world state), or three or more articles could communicate in this fashion. The "sender" and "receiver" also

do not have to be different articles; they could be two distinct instances of the same article open on different visitors' computers where the combined resources of both visitors' CPUs contribute to the global result. That article likely focuses on a single programming problem, but can now solve a problem that is twice as large. This is due to the final result being able to draw finished calculations from both visitors' concurrent executions of the article code.

Two working examples of Smart Articles (our Visual_Billiards and Ray_Tracer demos) already exist on our website as proofs of concept designed to illustrate the potential of the concept. We describe them below in our listing of encyclopedia articles.

3.2.2 Featured Noteworthy Articles

3.2.2.1 Bases_Game

The `Bases_Game` article is a training tool for students to build intuition about matrix multiplication and change of bases. In this example of Digital Game Based Learning, students use matrix operations to pursue a moving target and, while doing so, learn to get their matrix order right when placing shapes. Each button they press in the game strings another matrix term onto the left or right side of a product. As they do so, their current matrix basis and the target basis are both visualized as a color-coded drawing of axis arrows, which smoothly moves to the new place with each button press. Students are challenged to reach each target using only one action or matrix operation each. The product matrix is shown numerically and the resulting coordinate frame is drawn. Students can use the game to visually test their hypotheses, or as an experimental testbed to plan out matrix operations in a graphics program.

One major course topic for graphics students to understand is that there are significant differences when they pre-multiply rather than post-multiply—i.e., when they introduce new matrix terms on the side farthest away from where they are used to, farthest in the equation from the 3D points upon which their product will be used, now affecting those points in the most indirect way. It is notoriously difficult for students to master the implications of left-to-right order in matrix formulas, but teaching that is where this game excels.

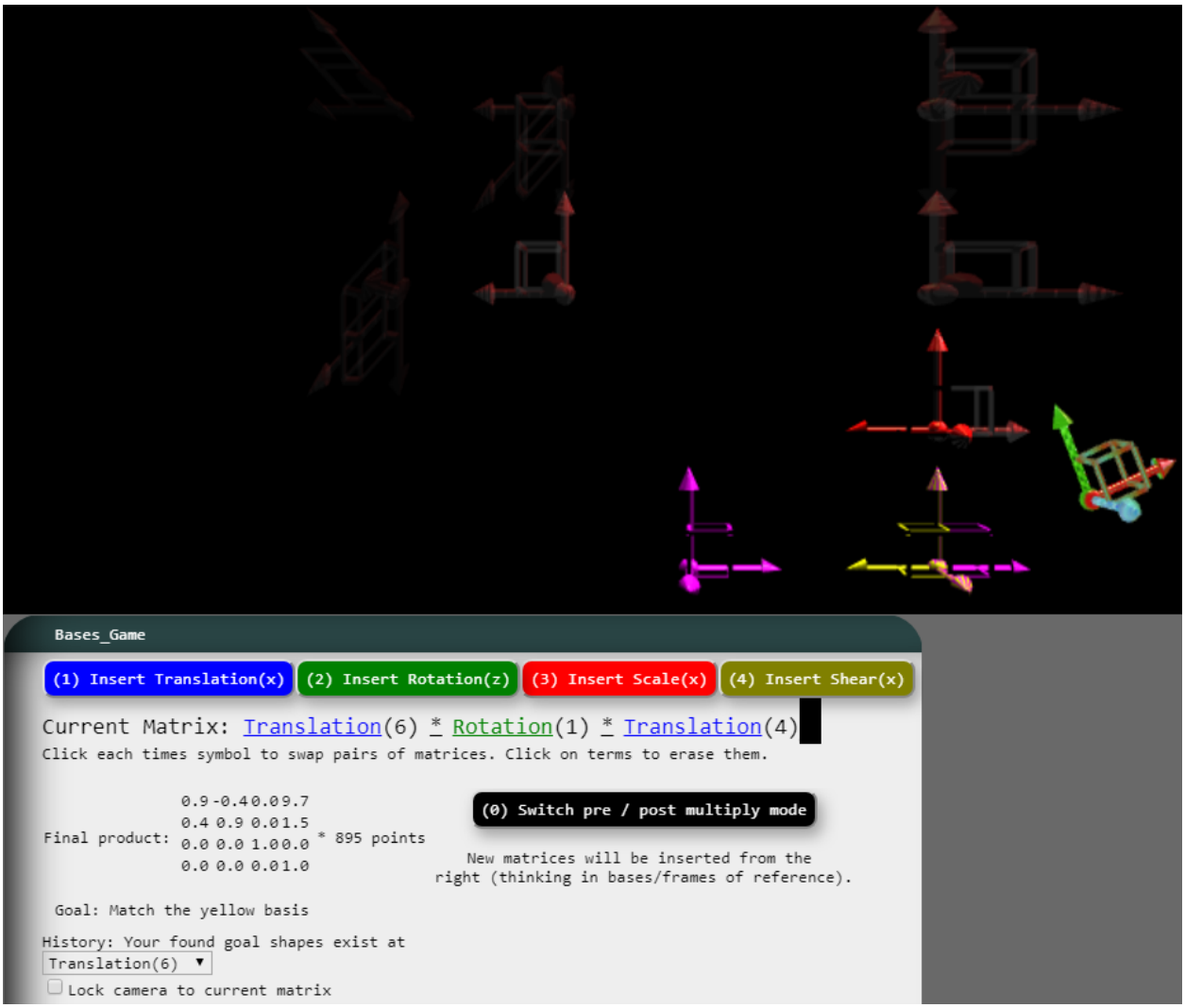


Figure 3.3: The colorful buttons control the bases game. The game gives 3D feedback and also shows the matrix product mathematically in two different ways.

From experience, this game's material replaces one or two class meetings of an introductory graphics course. Particularly, in the UCLA graphics course *CS 174A*, matrix order was the topic of the hardest long-form questions on both the midterm and final exams. Teaching assistants gave students dozens of examples like those exam questions, especially during review sections as the exams drew close. The examples that the lecturer delivered generated questions that could not be answered effectively without cluttering up the chalkboard. These hand-drawn examples could not measure up to the guaranteed correctness, interactivity, repeatability, and smoothly animated feedback the students get from the virtual educational toy featured in the `Bases_Game` article.

Our Computer Graphics students come from a variety of backgrounds including art and film, and may not have a math background. Our game helps these students to grasp matrix concepts visually. As they use the interface to manipulate matrices, the graphics update, making the student immediately aware of the mathematical consequences of their actions.

The "levels" of the Bases Game are modeled after particular past exam questions where certain "gotchas" of matrix order must be understood to answer correctly. Some of these lessons are more intuitive when delivered visually. One such lesson involves noticing that the lengths of axes of a local coordinate frame can change even without applying a new scale transform. Instead, applying a new rotation transform can counter-intuitively be what triggers such a re-scaling effect, provided there were prior scale matrices to the left in the chain.

A number of extra user-interface features are built into `Bases_Game` to enhance learning, such as the ability to use mouse clicks to rearrange or delete the terms of the existing product, and the re-sizing and truncation of text to keep the summary of the whole product in view at once. Other user-interface features actually introduce new topics, such an option that allows the student to follow the reference frame of their current basis by "locking" the camera to it while it moves. This allows them to explore the resulting inverted effects of their buttons on the shapes drawn, and the rules of matrix inverses of products, which are crucial to understanding the behavior of cameras in graphics. This feature allows the student to visit another popular exam topic: How drawing a basis is conceptually inverted compared to expressing points in a basis, requiring the opposite actions in the opposite order (just like when inverting a matrix product).

For students who are brand new to the concept of why matrix order matters, a supplemental document is provided that goes over how to combine matrices in practice, and the implications of writing code that builds matrix products starting from the left versus the right. Another document breaks down the camera matrix and other special parts of the final product used by graphics cards to get shapes to draw on-screen.

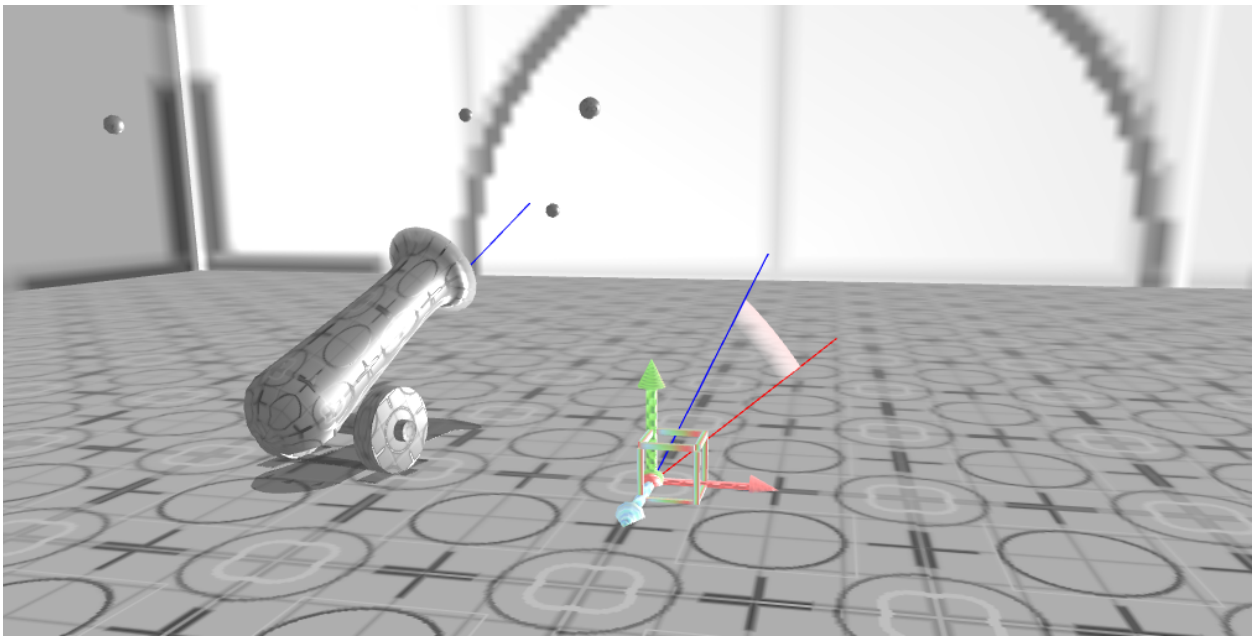


Figure 3.4: The main illustration panel of the Dot_Products article shows the angle of a cannon's vector while it spins around due to user interactions. The rest of the article uses the dot product math operation to calculate this angle.

3.2.2.2 Dot_Products

This example is our main proof-of-concept of an Active Textbook. The `Dot_Products` article resembles a single page of a typical math textbook, where the focus is on using illustrations to introduce dot products to a beginner. Dot products are used throughout 3D graphics to compare angles of vectors and to compute matrix products.

Since this is an Active Textbook, what appear between the text sections are not mere illustrations but interactive WebGL canvases. They show animated 3D scenes that portray several mathematical rules about dot products. In a connected way, all these active illustrations reflect the user's interactions so far.

A visually appealing scene is drawn on the main canvas, showing a cannon that is aimed by dragging the mouse. The angle of the cannon updates a vector plotted at the coordinate origin, marked by a drawing of each axis. Releasing the mouse fires the cannon, triggering an audio-visual experience as the cannonball strikes walls, bounces, falls, and rolls in the box-shaped room. Angles are highlighted in the drawing based on the cannon's vectors, and

in other canvases on the document the dot product of these same vector values are used to plot trigonometric functions, ultimately recovering the correct value of the cannon angle. A different illustration on the page visualizes the numbers of vectors as they rearrange and combine to calculate the dot product itself. The latter visualization is used in a companion article `Matrix_Multiplication` which is linked as a followup to `Dot_Products` for visualizing the same concept on matrices. A matrix product comes from simply doing a row-column dot product to compute each cell of the resulting matrix.

As the user adjusts the scene in one canvas, the other canvases update. The user will recognize their own footprint as being in common across all illustrations, making it more clear to them what control they have so they can better explore the scene. This article has no emphasis on coding and no coding widgets are shown here. It is not as related to Knuth's Literate Programming as some of our other examples, as much as Knuth's concept was the inspiration for this interactive re-interpretation of a math textbook. The article could instead serve someone in a high school level math class with no programming background just as well as it could serve our own university students. This reflects the ambition of the Encyclopedia of Code also to encompass the explaining of non-programming topics to a general audience.

3.2.2.3 Movement_Demo

This is another Active Textbook example, demonstrating the `Movement_Controls` tool as featured in Figure 3.5. This article introduces a "Secondary Scene Component", a type of demo that helps other demos by executing simultaneously and stacking additive functionality. In this case, the added functionality is the ability to smoothly change the global camera's vantage point using keyboard and mouse controls. Besides the camera location, `Movement_Controls` also provides code for smoothly changing the drawn location of objects with user input—third person control as opposed to first person. As the visitor scrolls down the `Movement_Demo` article, they are first taught about the workings of the first person controls, and then a second canvas shows an "out of body" demo where the same controls work in third person due to using the `Movement_Controls` class with a different settings flag (for

inverted matrix operations).

`Movement_Controls` was created to fulfill student demand for flexible manual camera controls in their 3D scenes. In the UCLA course *CS 174A*, most students historically created their term project (a custom short film) while being unable to see their scene from multiple vantage points in real time. Although they were required at least to hard-code a pre-made "look at" camera matrix into their code, most stopped there—live interaction using key or mouse controls was much harder and usually not done. Students who did try to implement it each did so in different incompatible and verbose ways.

Prior to the existence of tiny-graphics.js, back when the course projects were still done in C++, the instructors attempted to alleviate this by introducing a small camera matrix library called "ArcBall" into the template code provided to students for the term project. The ArcBall controls were mouse-only and were not capable of first person movement as provided. They simply allowed the user to spin the scene around its origin on two intrinsic axes via clicks and drags. Despite only adding that trivial functionality, the library filled the student codebase with difficult quaternion math far beyond what was necessary for its limited value added. It required importing six new files (of 653 lines of code, at 17.2KB) plus several calls added to the student's own file within their I/O callbacks.

Upon inheriting this code template from past UCLA instructors, we first replaced the entire Arcball library with two lines of code, to little distinguishable effect on the application's controls. This quickly confirmed that Arcball was a poor solution for UCLA's needs. The two lines manipulated the camera matrix on two intrinsic axes based on mouse drag position, pre-multiplying two new rotation matrices onto the camera's stored inverse matrix.

Students primarily wanted to be able to explore their scenes in first person, getting close to objects of interest from a desired angle. Code for first person controls (rotation and translation of the camera) proved slightly more complex, necessitating `Movement_Controls` as a distinct class.

Our final result for `Movement_Controls` first-person translation and rotation is accomplished with several button elements drawn on the page, which can either be held down

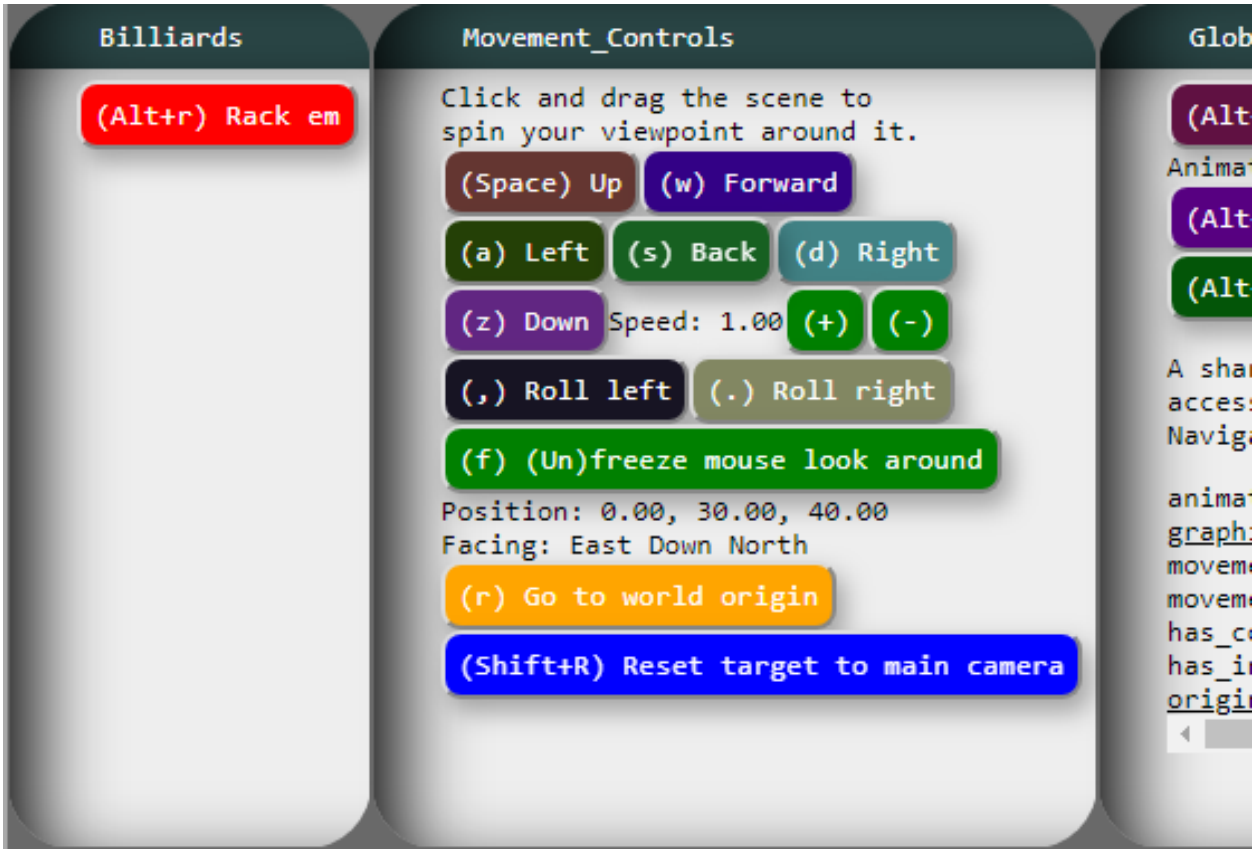


Figure 3.5: The movement controls.

with the mouse or fired with keyboard shortcuts that are printed on the buttons along with descriptive labels. See Figure 3.5. Multiple movement buttons can be held down simultaneously for fluid movement, and the buttons dynamically animate and change color to show viewers which buttons the user is pressing in case the user has an audience or a demo is being shown to a class.

In a small package (about a page of simple code), the class includes many more novel features beside its base behavior of first person movement. The first-person camera rotates towards where the mouse is hovering over the canvas. This steering behavior is toggled by a button, locked by default to prevent accidental hovers. Other buttons apply smooth translation motion in the six cardinal directions when held down, which stops upon heir release. The camera speed (the magnitude of rotations and translations) is adjustable via two more buttons for fine-tuned movement. Another button resets the camera matrix to the identity.

Third person movement is not only possible, but separated into two different concepts. Clicking and dragging the canvas still "rotates the scene" for easily viewing it as a unit although this is now done using first person paradigms, in the local coordinate frame relative to the camera and irrespective of the scene. It uses a "guess" as to how far in front of the camera the scene and center of rotation should be; the camera speed buttons adjust this guess.

True movement in third person, the movement of an object instead of a camera with all the same buttons for rotation and translation described so far, is now possible. As mentioned at the beginning of this section, this has been conceptually separated off into a separate camera class as in the "out of body" demo, with inverted matrix operations. To manage many objects that may compete with the movement controls along with the camera, pointers to different "graphics state" objects are held by each candidate and alternately referenced by the `Movement_Controls`, switching off whenever more "take control" buttons are pressed by the user. A skilled user can rearrange the entire scene's objects as well as the vantage point by simply switching between the objects that should have the movement controls' attention.

This control scheme might be an unprecedented design across the literature, the web, and the games industry considering its multiple behaviors, animated buttons, the switching of control, and its peculiar nature as a Secondary Scene Component.

The `Movement_Controls` class in our code affects our users even while they are not navigated to the article about it due to its inclusion in most other demos. It is automatically imported by the code of scenes such as `Tutorial_Animation`.

3.2.2.4 Frustum_Demo

Frustums are pyramids pointed at you with their nose cut off. They are used throughout Computer Graphics to define view volumes, which are the somewhat box-shaped regions in 3D that represent the space visible onscreen. Generally speaking, if a point in 3D happens to fall inside the current view volume, the graphics card will attempt to draw it; otherwise not. All points in the volume get projected onto its nearest plane (the "image plane" or "near

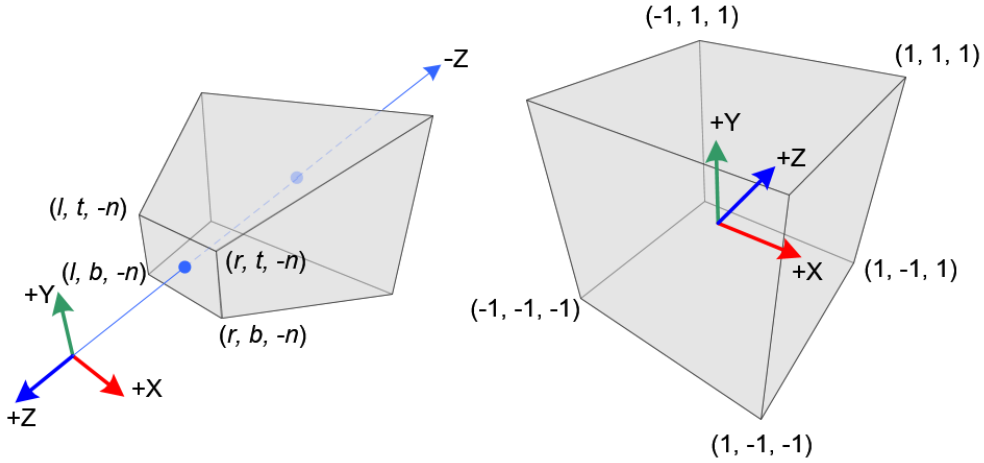


Figure 3.6: A diagram of a perspective frustum on the left, and Normalized Device Coordinates (NDC) on the right. From (Ahn, 2009).

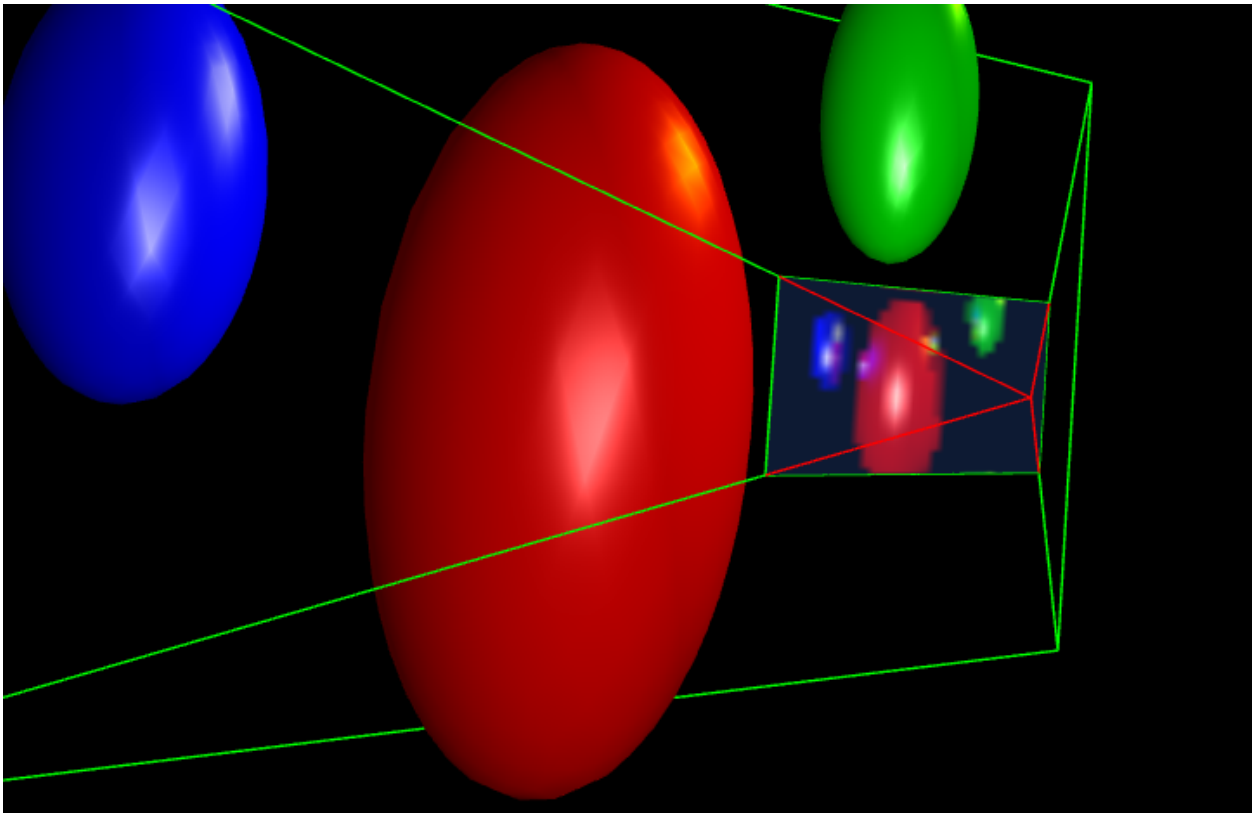


Figure 3.7: The Frustum_Tool shown drawing a view volume, embedded in a demo where the image is also shown that would be projected from that view volume onto the near plane.

plane") to generate the final planar on-screen drawing. The green line drawing in Figure 3.7 shows what a view frustum and projected image theoretically looks like.

Our students rarely find any reason to directly work with view volumes, or to change theirs from the default in the program template we provide them. Nonetheless, their understanding of view volumes is crucial toward gaining a simple mathematical grasp of the underpinning of most graphics programs: the stages of the 3D graphics pipeline. The "pipeline" is a topic that is brought up throughout Computer Graphics textbooks to explain the math done by the graphics card (Angel and Shreiner, 2014a). The pipeline is an imaginary multi-stage process, mapping a shape's points from wherever they exist abstractly in 3D object space onto the correct pixels shown in 2D screen space.

The `Frustum_Demo` article illustrates the concept of view volumes to graphics beginners, using demos with the help of what we call `Frustum_Tool` objects. These objects draw a plot of lines in the 3D scene, located at the exact edges of a given view volume shape. This is defined by providing a camera matrix and projection matrix. If those match the ones that are currently being used globally for drawing everything, then the plotted lines will be barely out of view, as their projection (from 3D) lines up precisely with the rectangular boundary of the 2D canvas. Otherwise, the user will see either a rectangular box or an angled frustum (depending on whether an orthographic or perspective projection matrix was passed in to initialize the tool), and the frustum shape will appear precisely in front of the location of the provided camera matrix passed in to initialize the tool.

This tool can help programmers diagnose issues with graphics projections. As a Secondary Scene Component, `Frustum_Tool` stacks alongside other scenes and can help visualize the view frustum being used globally by other scenes. This works because the default behavior is for the global *projection* matrix to be passed in to initialize the tool, whereas the *camera* matrix is different from the one globally used. As the global camera matrix is moved around by the user as they fly around to explore, the drawn frustum stays in place due to its own matrix's location. The drawn frustum disassociates from the actively used one and can be viewed out-of-body.

The inner workings of the `Frustum_Tool` are extremely simple. The code begins with the canonical view volume (a cube extending one unit in each direction) and then undoes the projection transform via inverting the matrix. The code for this is just a few lines long, applying the inversion procedure to the points of the cube. No further analysis of the matrix is necessary, such as determining its type (orthographic or perspective).

Our inversion procedure uses a mathematical trick independently discovered here that may be useful to other programmers and researchers dealing with projection matrices. Understanding this trick depends on familiarity with the perspective division operation. This division operation is unique for being the one step in the graphics pipeline not expressed as a matrix (because it is non-linear). Instead what it mathematically does given a coordinate point with four components (x, y, z, and w) is to simply divide all four by w. The perspective division broadly helps with a transition from 3D to 2D for screen display purposes, and it also achieves effects such as vanishing points.

We discovered that any projection of points onto the near plane can be undone by basically re-projecting them again. Namely, apply the linear projection matrix first as always (except use its matrix inverse instead), and then repeat the perspective division operation exactly, dividing by the fourth coordinate. One might naturally expect something different, like having to reverse the steps instead to undo them—to multiply by w first instead of dividing by w last—but then there would be nothing to multiply by at first. Our canonical cube's fourth coordinates are all equal to one. Projections by their nature wipe out a coordinate in a non-invertible way, but we somehow need to recover all coordinates from our post-projection cube anyway.

Due to some coincidence of math, using our method not only successfully reverses the division in a sign-correct way but also correctly recovers the Z coordinate from the post-projection points. According to the author's own automated testing this technique works works on general 4x4 projection matrices irrespective of projection type (perspective or orthographic) or parameters such as field of view and vie plane locations.

The `Frustum_Demo` article itself illustrates view volumes by rendering a scene containing

small colorful balls (See Figure 3.7). These are randomly scattered in view or just out of view. Exploring the scene dissociates the real frustum from the drawn one and the visitor sees a line plot of the original view volume, and they will see how it encloses precisely the set of balls that were initially in view and no others. Shadows of the balls that are out of view are still drawn so the student can be aware that they exist barely out of range. This is done via multiple rendering passes, some of which send a wider view volume (one that does capture all the balls) to `Frustum_Tool` instances.

Since `Frustum_Tool`'s algorithm is agnostic to whether a perspective or orthographic projection was used, this means our code process can draw either type and, via a button, seamlessly switch from an orthographic matrix to perspective and back. The viewer is able to instantly see the difference between the two view volume shapes.

Our `Frustum_Demo` is inspired by a demo created by the makers of the three.js engine, located on the three.js examples website under "camera" (Cabello, 2013). Both our demo and theirs illustrate for students the difference between an orthographic and perspective projection by allowing them to toggle between the two. Both show a line plot of the frustum alongside a scene drawn using that view volume. Compared to three.js's demo, our viewers are able to understand the differing view volumes' implications better by moving the camera to line up with them and noting which objects enter and leave the view.

3.2.2.5 Ray_Tracer

Ray tracing is a challenging alternate method of drawing 3D scenes onto a computer screen compared to the normal method of "rastering" geometry. In the normal approach, thousands of triangles are projected one by one onto a canvas where they are in-painted, each one independent of the other. Ray tracing, however, must consider all the geometry together to figure out where rays of light will travel, bounce, and absorb color to decide each pixel's result. Ray tracing a frame of an animation involves a computationally intensive search process through the entire set of objects in the scene to ensure the correct one is struck by the light ray. This search process and the ensuing vector math must be repeated for every

single pixel and every ray of light.

Notwithstanding simplified examples like ours, ray tracing is typically not possible in real time. It is hence usually not seen in the games industry (where images must be computed fast enough to be shown in immediate response to player input), though it is the algorithm of choice in the film industry (where the images of a movie may be slowly computed offline for playback later). Implementing a ray tracer is a common assignment in beginner graphics courses, and is frequently the final assignment of UCLA's CS 174A.

`Ray_Tracer` is presented as an Active Textbook for teaching students the algorithm of ray tracing piece by piece using a buildup of partial demos. We finally implement a real time ray tracer that manages to run in a website in a single JavaScript thread. It smoothly animates camera movement through a scene of arbitrarily placed, stretched, and rotated balls. It features shadows, reflections, and refractions.

`Ray_Tracer` contains a number of features that make it an excellent visualization of the ray tracing algorithm compared to any prior (typically very simple) online demo. The same scene is drawn with and without ray tracing simultaneously to show students the difference. The ray traced version (see Figure 3.8) is executed from many vantage points in the scene, and all resulting images from these vantage points are shown to the student, embedded in appropriate places in 3D within the non-raytraced visualization of the scene.

In textbooks and lecture slides and on Wikipedia a common illustration found for explaining ray tracing shows a ray striking an object, branching, and bouncing to the next object. Since these are usually hand-drawn, the student cannot visualize the emergent flow of all rays throughout their custom scene, or see the scale of just how many are used, nor see any live response to movement of their scene or vantage point. In contrast, in our example the colorful rays themselves are drawn embedded in the scene, showing which objects are struck by them. Students can directly observe the consistency between the colors of the rays of light and the objects they are striking (shown as non-raytraced geometry). Rays drawn can include primary rays as well as secondary rays (reflections, refractions, and helper shadow rays). While programming a ray tracer for their assignment, this feature can help students

directly diagnose wrong paths taken by any single ray.

Our real-time ray tracer is built out of customizable parts. To generate the vantage point(s) the rays are emitted from one or more frustums that are each configured in code using a typical camera matrix and projection matrix. In our demo the frustums themselves are drawn using our `Frustum_Tool` Secondary Scene Component. The final ray traced image may be drawn directly onto the near plane (image plane) of the drawn frustum to show exactly where points within that frustum have been projected to. One subclass of `Frustum_Tool` adds this image-generating and drawing behavior, and in addition to this image embedded in 3D (using a special texture), also inserts the image directly into the web document in a way that can be saved to a file. This is not the only way to display the result of ray tracing; another subclass of our `Frustum_Tool` uses the same stored projection and camera matrices to place a different kind of visualization of a viewer, displaying the rays' colors as colored cones of a virtual eyeball. Either display method can show the rays bouncing through the scene as well, piercing through either the image plane in correct places or through the cones. The concept of a frustum located somewhere and being the origin of the rays remains, but the decision of how to display the ray traced result is flexible.

Rays need not always be cast outward in only a traditional grid pattern to sample colors in the scene. Instead, custom distribution patterns called `Sampler`s are used. They map the rays, arbitrary vectors in 3D (usually defined in spherical coordinates), to pixels. Pixels are arbitrarily located on the the near plane of the view frustum, that is, the image plane corresponding to the final image drawn onscreen. We provide a variety of samplers, both stochastic and regular, including ones that get denser toward the center of view like the human eye does.

Different types of frustum drawings and `Sampler`s can be mixed and matched arbitrarily to create a number of effects, such as a bio-mimetic drawing of a functional eye. Our biologically inspired eye shown in Figure 3.8 has a lifelike pattern of hexagonal receptor areas forming spiral towards a denser central fovea (the highest resolution region of the eye). Our demonstration of foveated visual sensing was inspired by Terzopoulos and Rabie (1995), wherein the researchers simulated the vision of artificial fish. They rendered the scene with

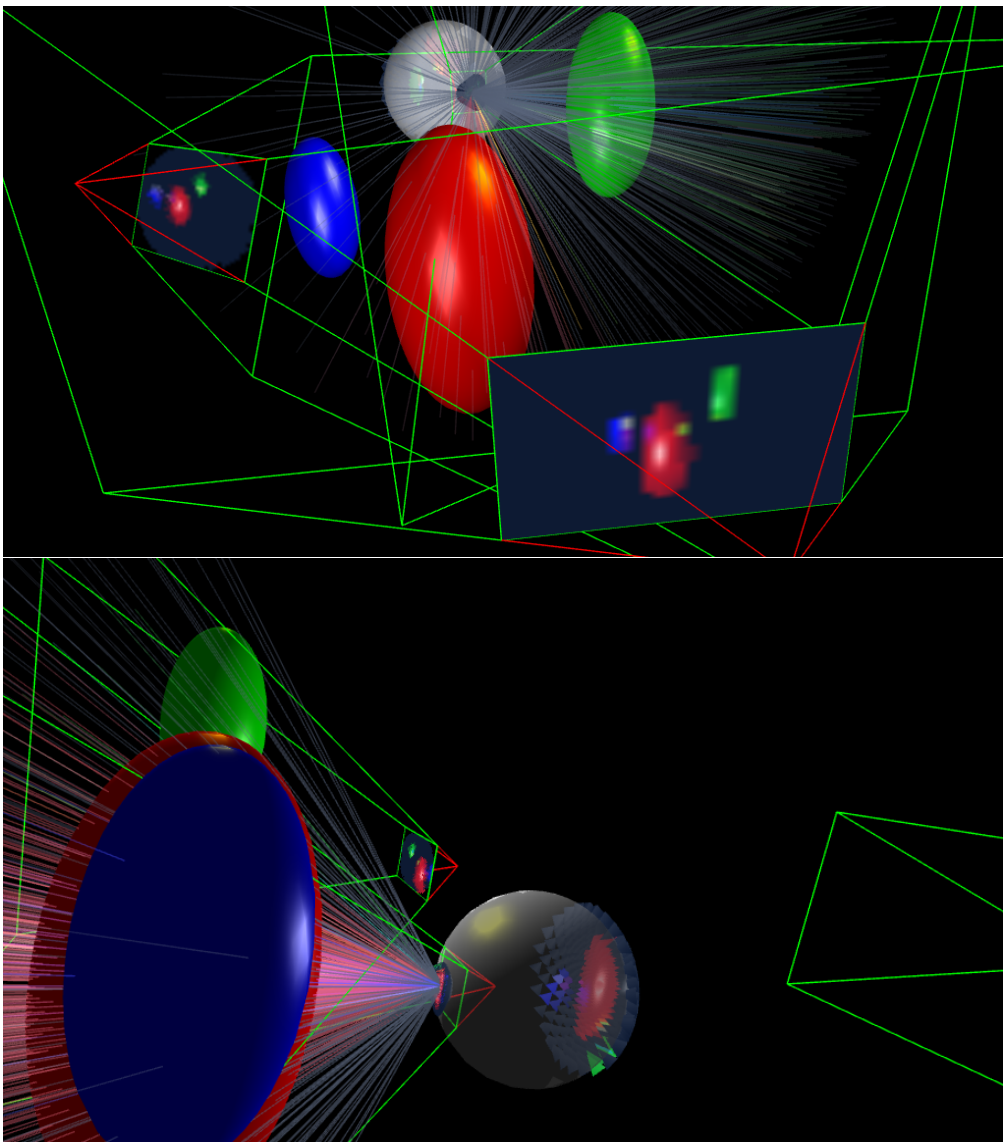


Figure 3.8: The article shows a ray tracer employing multiple subtypes of Frustum_Tool. Besides to the standard flat image one, this also includes a simulated eyeball. These have a dense sampling fovea, image projection onto the retina, and individual primary rays visualized.

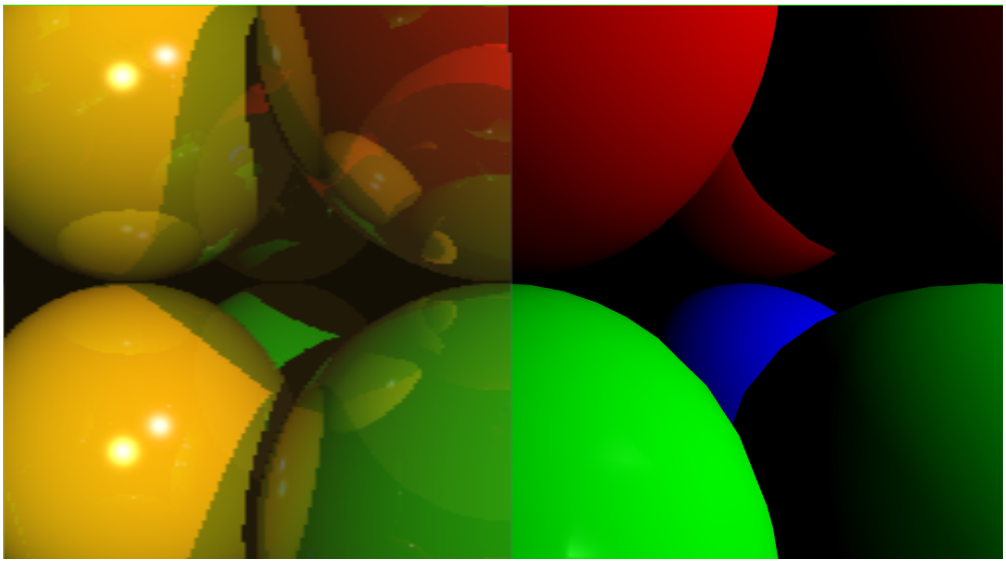


Figure 3.9: The ray tracer shown up close (from its own view volume's perspective). Scan lines are progressing across the image from the left, showing the ray traced image result with reflections, refractions, and shadows. Meanwhile, the right half of the image has not been ray traced yet and still shows the traditional triangle-based geometry behind the image. The difference in quality is evident.

several coaxial cameras that increase in resolution towards the center, just like in biology. The fish then adjusted their swimming so that interesting stimuli became centered in these maximally sampled vision regions.

Applications of this flexible sampler system exist in education about ray tracing concepts as well as for research in simulations involving the eye and considerations of the behavior of light entering the eye.

3.2.2.6 Visual_Billiards

`Visual_Billiards` is our proof-of-concept example of a Smart Article, themed using a game of tabletop billiards. Recall that a Smart Article, defined in Section 3.2.1.10, leaves an unfinished computation task on our server for completion upon subsequent visits to our articles, essentially delegating work to the future (and potentially to better-specialized articles).

To exhibit the Smart Article concept, `Visual_Billiards` runs its billiards-relevant code and then communicates its finished calculations to our server for a companion article

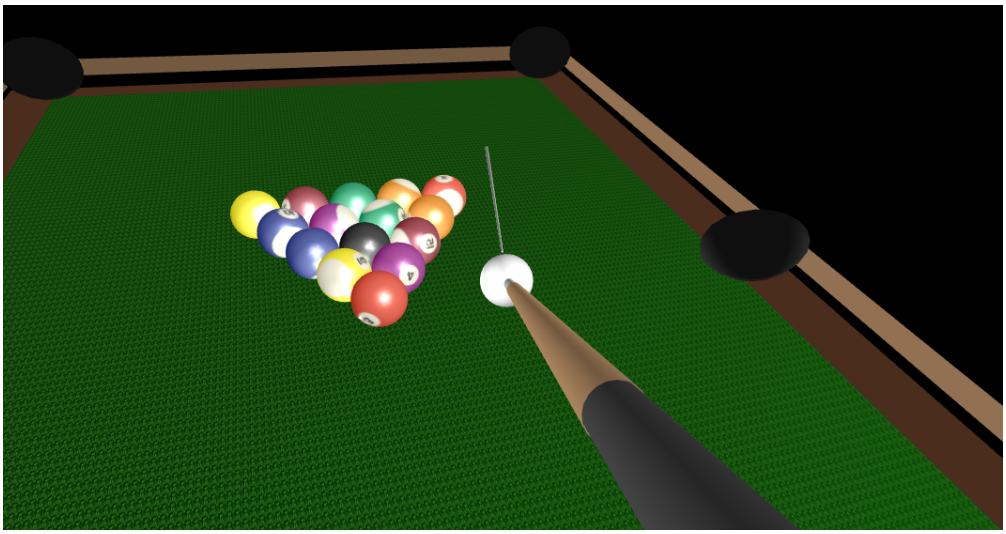


Figure 3.10: The Billiards demo. The white line is for aiming.

(`Ray_Tracing_Tool`) to use, which sends its own sort of calculations back. The back-and-forth and delegation of work to outside code about a different topic (ray tracing) enables new features in the demonstrated game—the drawing of rays onto the billiards table as a helpful visual for players to improve their aim.

Billiards The `Visual_Billiards` builds upon a demo called `Billiards` (See Figure 3.10), which is our main example of a game implemented using tiny-graphics.js, in addition to being a showcase of our `Simulation` class and the concepts of timestepping and collision physics. The game shows a minimalistic billiards table, balls, a cue stick, and a shadow of the cue stick representing the stick trajectory. The player drags the mouse to aim the cue stick, and clicks to "fire" it into the cue ball at a speed proportional to its distance from the shadow stick. The balls respond physically (and with audio) by bouncing off the walls or partially disappearing into the table's pockets. A special camera system is used that constantly seeks the position requested by a `Movement_Controls` instance, adjusting this in consideration of the stick position and the top of the table so that the player's mouse drags always aim the stick towards the cue ball and maintain a reasonable camera angle where the bottom of the table is never seen.

We added basic networking code to the `Billiards` game so that it can send and receive

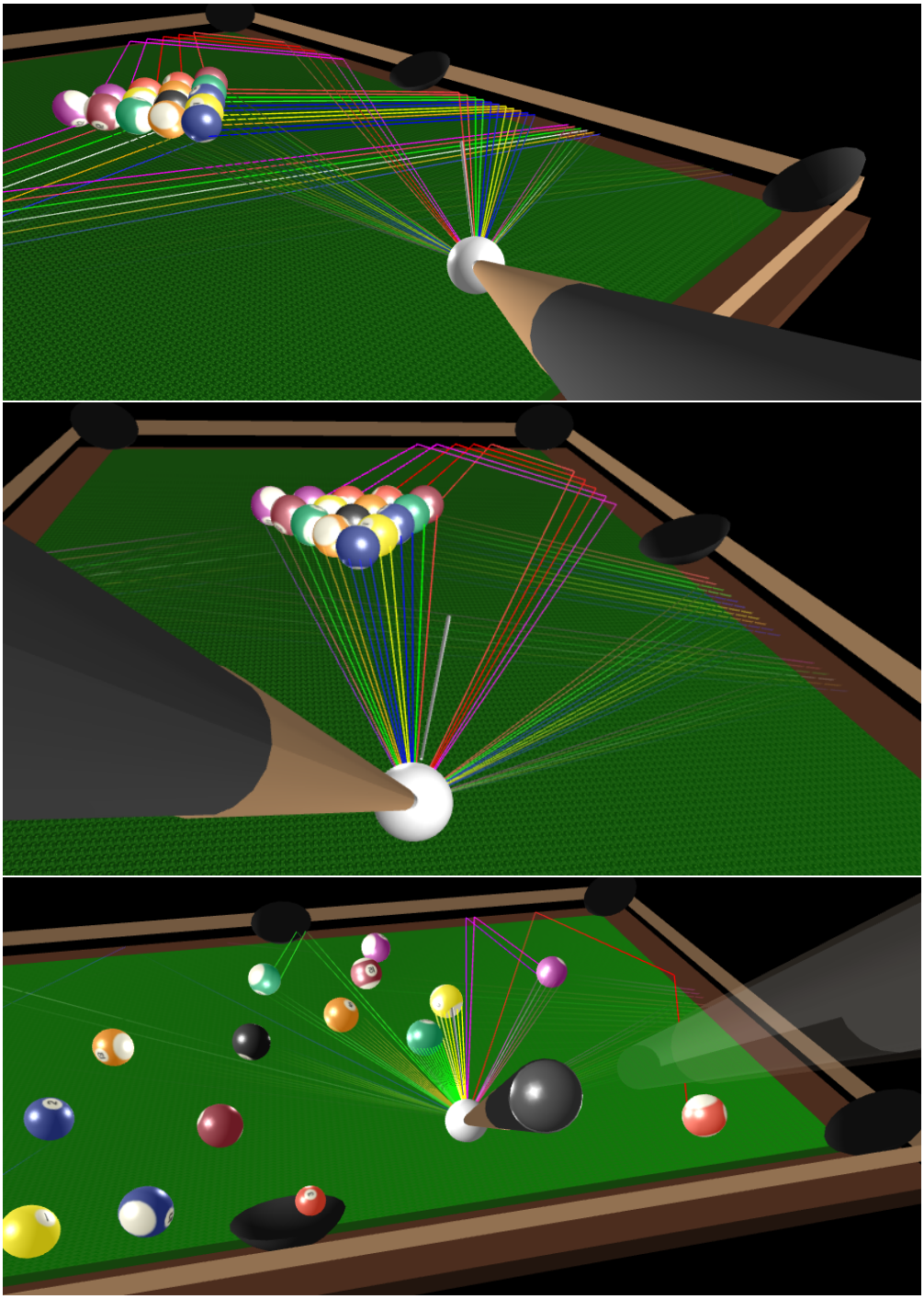


Figure 3.11: Visual_Billiards, showing colored rays to guide the player. This is possible due to the assistance of a concurrently open page navigated to the Ray_Tracing_Tool article.

JavaScript data structures. This functionality appears in derived classes of `Billiards` so as not to clutter up the main code. In nested, derived classes we add one feature at a time. One such addition enables multiple players to participate in the pool game by taking turns. Another enables a single player to save the game's state and recall prior configurations of the table. Sounds and lights are used indicate to a player when their turn starts. We continue wrapping the main class in more networking code to progressively build up to the more advanced functionality of `Visual_Billiards`, our Smart Article proof of concept.

Making the Article Smart To show that we have a Smart Article, we must now make the billiards game and running ray tracing program benefit each other while running, as though they were stopping to consult each others' differing content—as though they each were reading the other article to learn more. This means two articles will help each other to solve specialized problems they cannot solve alone. Separately, the `Billiards` program only "knows" the nuances of simulating and stepping physics as it applies to billiard balls and resolving their collisions, but it can hardly do anything else. The unrelated `Ray_Tracer` article reports the paths of light rays as they bounce in a scene with balls, forming an image, and can hardly do anything else. By saving their world data to a common place in the network, these two articles can interact. They will pair up to solve a bigger problem.

To do this, `Visual_Billiards` adds a twist to the game. When a player finishes their turn, this triggers a signal to be sent to a different web demo called `Ray_Tracing_Tool`—assumed to be running at the same time, so someone must have an open window of its URL on a computer somewhere. Its running process does some ray tracing work. Then, the ray traced result will be beamed back to all the instances of the `Visual_Billiards` article that different players have open. Each communication follows the flow of information we explained for Smart Articles with in Figure 3.2. Since the series of communications is back-and-forth, `Visual_Billiards` and `Ray_Tracing_Tool` alternately assume both roles in the diagram: "Sender" and "Receiver".

The ray tracing is done in a way that answers some particular questions about the state of the Billiards game. Namely, `Ray_Tracing_Tool` renders the scene from the cue

ball's point of view in 360 degrees, tracking light rays that bounce across the billiards table and reflect upon reaching the table boundaries as though they were mirror covered walls. Since these reflections are similar to prospective paths a ball would bounce off of cushions, they represent prospective paths of the cue ball that could hit another ball. All of these represented prospective paths are literally visible to the ray tracer, which sees them as colors reflected in walls, colors matching the ball that would be hit. The client running `Ray_Tracing_Tool` displays the rays onscreen.

More importantly, the ray information is helpfully sent back to each browser client running `Visual_Billiards` so the players can see the colorful rays too. Once `Visual_Billiards` receives the rays back, it draws them directly on the table, emanating from the cue ball and bouncing off the cushions. This can be seen in Figure 3.11. They are colored to show which ball would be struck if the cue ball is launched at the cushion at that angle, so players can better line up their shots.

The final ray information shows up for all players whose client pages had the `Visual_Billiard`s page loaded. The information is only available because of what `Ray_Tracing_Tool` knew to do with ball positions and rays of light, that the Billiards article itself did not know how to do. It is important to maintain perspective that each of these articles is a self-contained, minimal program about its topic—the source code running the Billiards article only has things in it that are pertinent to Billiards and physics, to easily let a student or programmer fully understand the code's small scope. Yet due to the small amount of networking code included, each minimal program can do things that are not in its own source code by collaborating with other articles over the network. It is as though one of our articles designed this way is somehow being read and understood by a computer instead of just a human.

3.2.3 More Summaries of Current Articles

3.2.3.1 Main_Page

`Main_Page` is a splash page showing new visitors a description of the website using similar language as that found in Chapter 1. A selection of the best demos taken from other articles are drawn in the 3D HTML canvas, with a simple "previous" and "next" button interface for cycling through them. The page invites new visitors to click through links to the following articles as a beginning tutorial.

The tutorial's progression has branches depending on the needs of the visitor (if they need math review, they are directed to the `Dot_Products` article first, for instance). Or, if the visitor wishes they can proceed through all the articles in a numbered ordering where each builds upon knowledge from the prior ones.

3.2.3.2 Environment_Setup

This article prepares the student for downloading our demos for local editing and debugging, allowing them to work on their assignments or personal projects in a more serious environment than what our in-page code widgets can provide. The instructions provided show the various features of the Google Chrome browser for assisting programmers.

In general, we find that UCLA's required Computer Science courses do not sufficiently prepare new programmers to use debugging environments to monitor program state, nor where to find all the other tools available to them that can help empirically diagnose issues with their code. This leads to countless hours being wasted throughout the students' graphics course (and potentially thereafter) with each new misunderstanding of their own programs. This article is an attempt to alleviate that by walking students through the unusually thorough debugging features of Chrome, and the ability to set up an environment in Chrome that maps its file browser and editor to their local filesystem so that they can turn in their finished assignment as a traditional bundle of files.

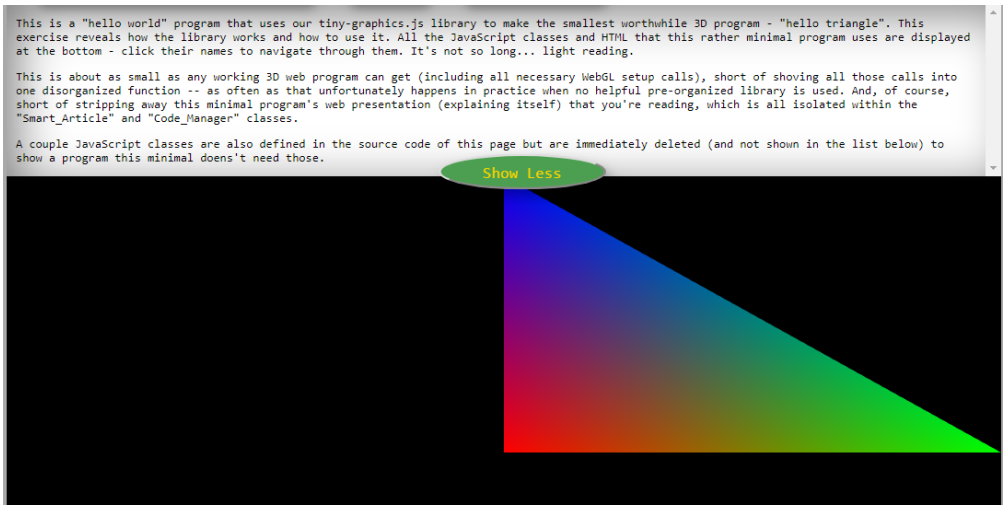


Figure 3.12: The top of the Minimal_WebGL_Program article, with the explanation section widget's text already expanded.

3.2.3.3 Minimal_Webgl_Program

`Minimal_WebGL_Program` shows the literal smallest program that can be made using tiny-graphics.js. Its purpose is to help users familiarize themselves with the library's parts and basic usage thereof. It implements a trivial shader, shape, and scene. The scene draws a single colorful triangle onscreen. Many classes within tiny-graphics.js are unused by this simplistic demo. A total of around 65 lines of code are added to tiny-graphics.js to make this smallest possible working usage of it.

3.2.3.4 Transforms_Sandbox

`Transforms_Sandbox` is meant to give users a first taste of graphics programming by encouraging the changing of the matrices used to draw an extremely simple scene of two boxes and one ball. Setup code is hidden away in a superclass, and anyone who modifies this class only sees code directly related to the matrix transforms without any other clutter. Using the provided code interface the user can immediately see their changes update in the drawing.

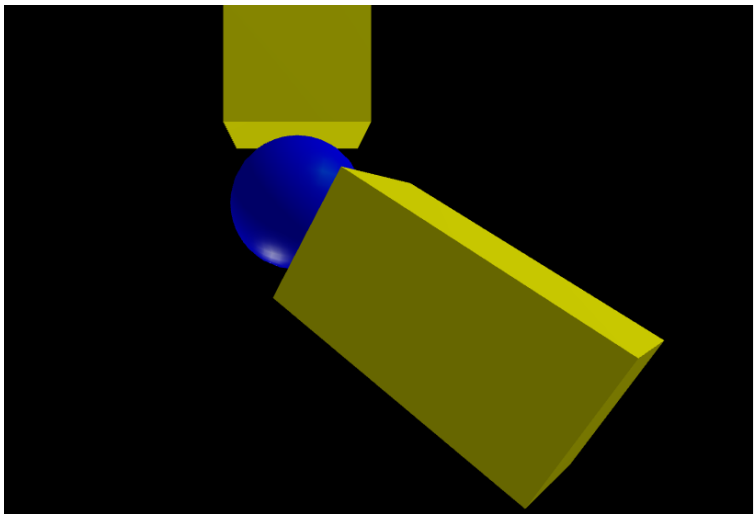


Figure 3.13: The Transforms_Sandbox demo for newcomers to graphics programming to modify and experiment with.

3.2.3.5 Bases_Game_Level_2

This game in Figure 3.14 is a follow up to the `Bases_Game` described in Section 3.2.2.1. This additional "level" of the game teaches how to animate movement. The primary difference compared to level one is that the targets the player pursues and tries to match are now moving—that is, translating and rotating according to simple linear or sinusoidal functions of time. The buttons that the player uses to chain more matrices onto the product are also new (shown in Figure 3.15), and now apply several new matrix values that are functions of time; these allow the user to exactly match the movements shown. The progression of goals in this level is the same order of transformations that a programmer would use to cause two box volumes to perform a hinge movement, hooked up precisely at their corner edges. If the player were to stop at some of the intermediate goal coordinate bases and draw boxes there, they would accomplish this effect. To illustrate this, transparent boxes are shown in addition to the target coordinate axes that must be reached on the way to the box bases. A player who can hinge two moving boxes together in exactly this way has demonstrated competency at using matrix order to place shapes.

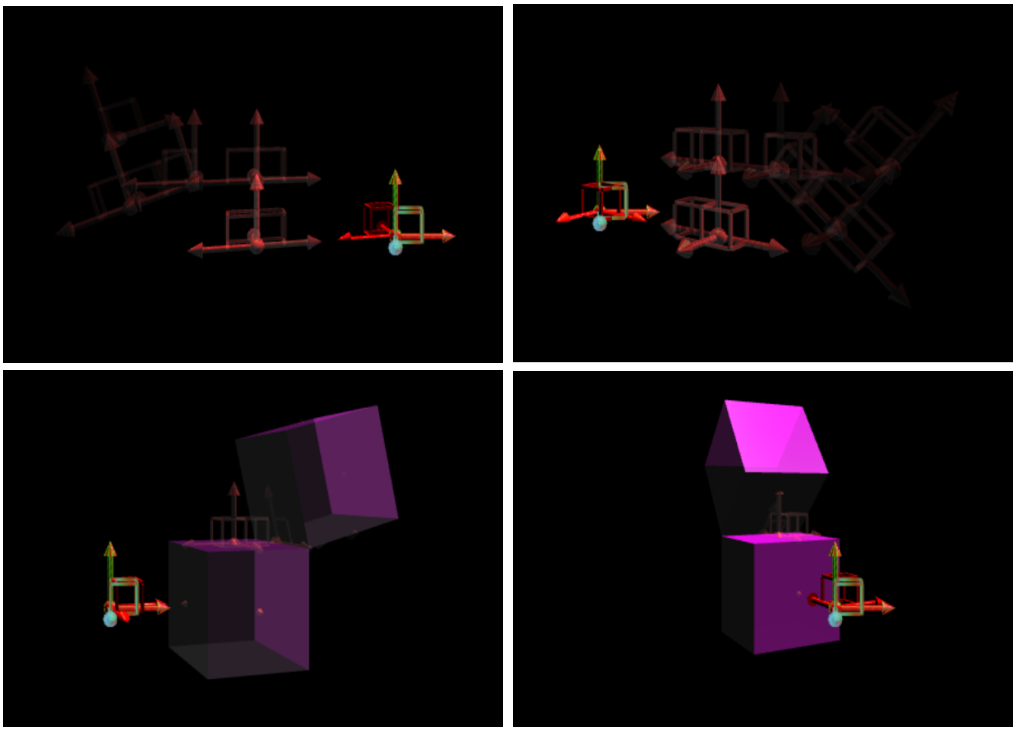


Figure 3.14: The second level of the bases game, with targets (axis arrows) whose transforms oscillate as functions of time. Their placements correspond exactly to the matrix bases that a programmer would iteratively make in order to draw the two hinged cubes shown, which revolve around the Y axis.

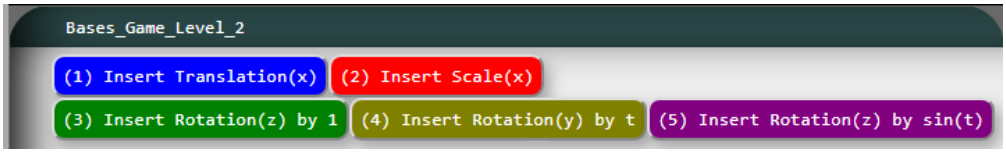


Figure 3.15: The modified controls of the second Bases Game level, using functions of time.

3.2.3.6 Phong_Shading_Demo

This is a demonstration of Phong lighting. One component each of the final calculation (such as normals, ambient, diffuse, and specular) from the Phong Refection Model are drawn. It is an Active Textbook page, so these appear in separate illustrations using multiple canvases, between textual explanations of the calculations that are being done. Each illustration is accompanied by the code that made it, which progressively gets longer and more complex as the article adds pieces of the full Phong algorithm. This goes on until the complete `Phong_Shader` class is built, containing all lighting components. The full code is then shown

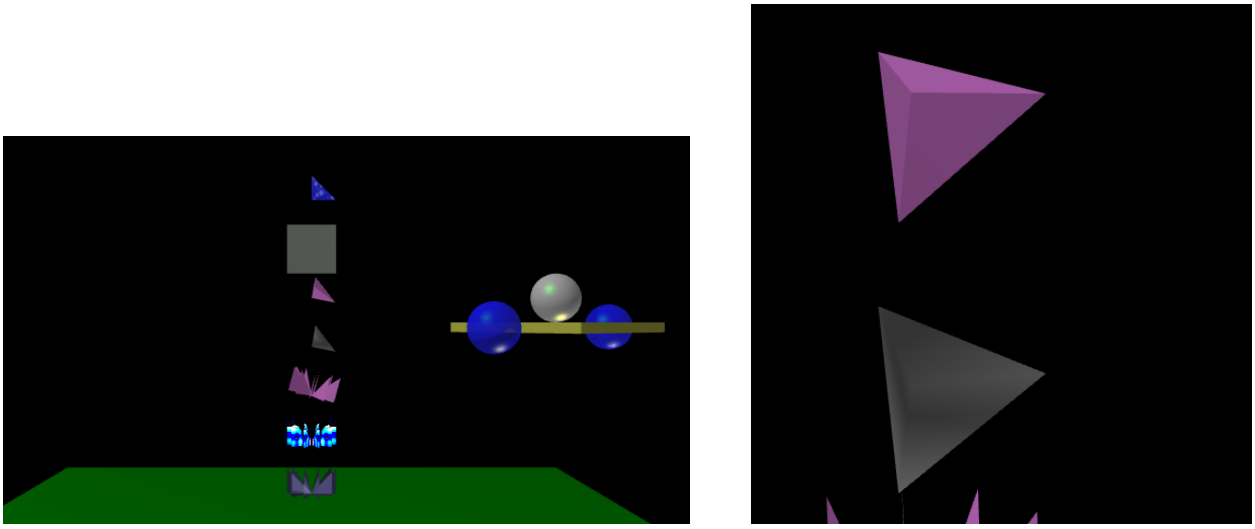
along with the visual result of Phong lighting. The final code builds the article itself, making this an example of Knuth's Literate Programming concept.

The final canvas shows `Fake_Bump_Map`, a slightly modified version of class `Phong_Shader` that demonstrates a form of bump mapping, to demonstrate that particular widely-used concept in graphics to students. The effect is exemplified in Figure 3.18a. We made a simplified form that does not use an actual bump map; the texture image itself is used to perturb surface normals. Nonetheless the effect is convincing, visually appealing, and requires barely any extra code, so `Fake_Bump_Map` is used in place of `Phong_Shader` in many of the other demos we are describing.

Definition: Materials Using lighting in other WebGL frameworks like three.js involves learning about and managing very complex objects called "Materials", but our analogue for this is comparatively simple. In our system, materials are just plain JavaScript objects—namely, tables of a few Phong parameters with values

More generally, a `Material` is a JavaScript table collecting all the stored numeric values that are bound for the shader program in the GPU. A material is associated with a particular 3D object drawn in the scene. Other shader-bound values might also exist that are associated with the whole scene rather than any particular shape; a separate but analogous data structure from `Material` called a `Graphics_State` is used in our program to pass those scene-wide values onto the shader as well. Whereas a `Graphics_State` stores values like the camera and projection matrices, lights, and time measurements, a `Material` stores values like the reflectively, color, or texture image of a particular object.

Every possible `Shader` in our system provides their own definitions of this table, returned by calling a function called "material". If `Phong_Shader` is the shader subclass that is used, then the particular parameters stored will include a base color, ambient brightness, diffusivity, specularity, shininess, and an optional `Texture` object. Otherwise, in cases where a different custom type of `Shader` is used, the parameter names and values stored could be anything desired. A compact notation is provided for properties of material objects to be individually overridden to derive new materials.



(a) Zoomed out.

(b) Close up of the two shapes demonstrating flat versus smooth Phong shading.

Figure 3.16: The Tutorial_Animation demo

3.2.3.7 Tutorial_Animation

`Tutorial_Animation` is the superclass of `Transforms_Sandbox` and therefore shows the previously-hidden setup code of simple scenes. This includes the declaration of shapes and materials (wrappers of lighting parameters, as discussed above) and the setup of the camera and projection matrices.

`Tutorial_Animation` depends on a few other classes that in short code snippets define how to draw basic minimal shapes (trivial ones that a beginner might make, such as a triangle or square). Via our server's dependency injection, the source code of this article automatically includes those shapes, and so does the code navigator that visitors use. The user may use the navigator to reveal the shapes' definitions in source code, clicking through to read them one at a time in order of increasing difficulty. This demo therefore serves as a good first introduction to reading our tiny-graphics.js library and building shapes from vertex arrays. It also demonstrates storing lights and materials (see previous section) and shader programs, matrix transformations, and how to tell the difference between when a shape is flat shaded or smooth shaded.

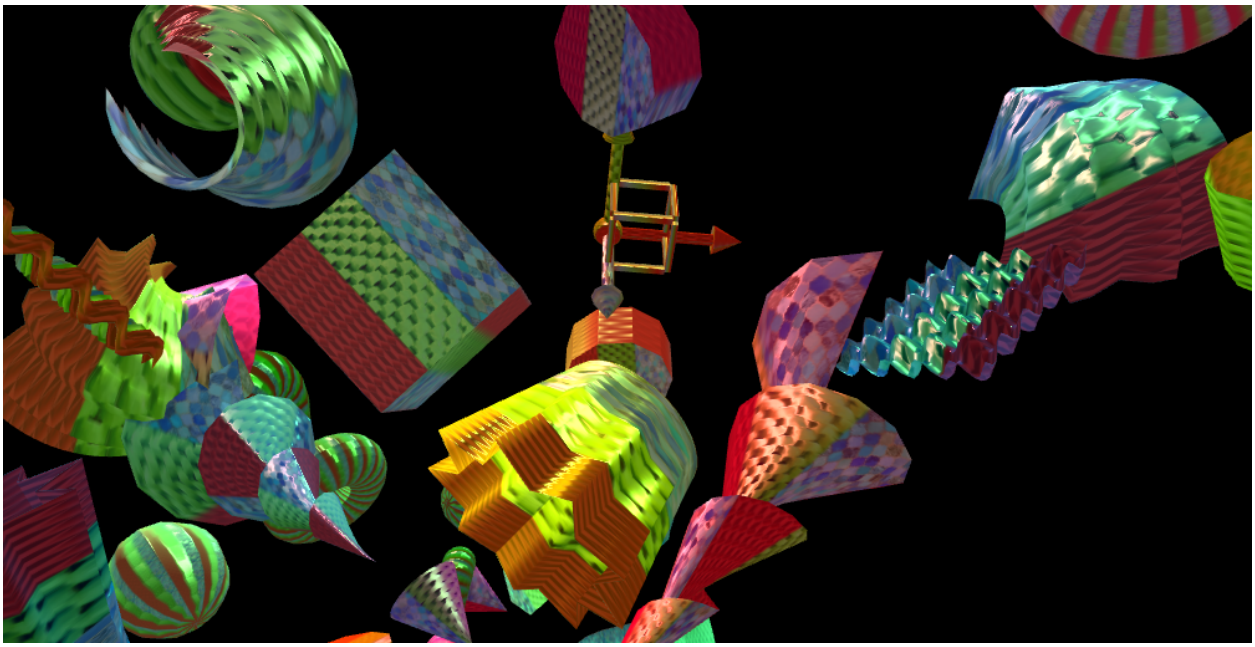
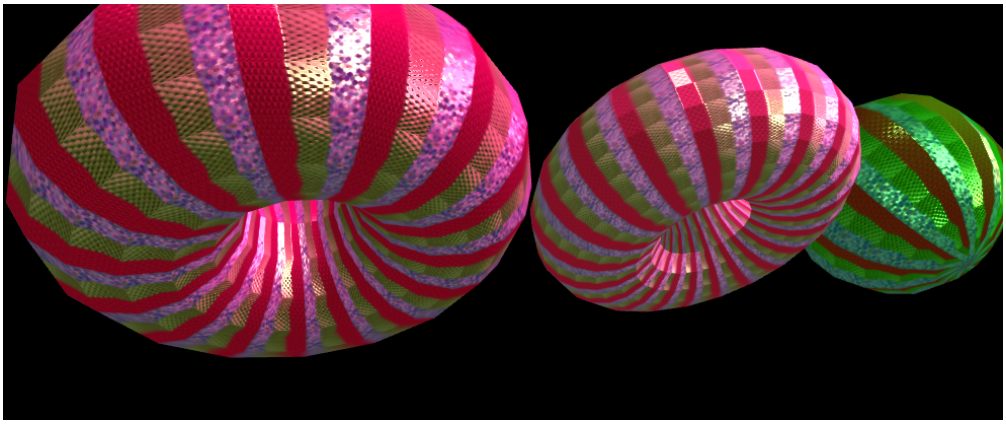


Figure 3.17: The surfaces demo.

3.2.3.8 Surfaces_Demo

`Surfaces_Demo` is a demonstration of how to make a diverse set of shapes using the least amount of code. It contains examples of shapes (custom vertex arrays) including cylinders, cones, spirals, and more that all derive from a common technique. As already introduced in Section 2.2.2, its programming takes advantage of JavaScript language constructs ("arrow functions" which are callbacks defined inline) to great effect, creating a novel shape generator not yet seen in other graphics engines. Arbitrary operations—matrix transforms or otherwise—are allowed during a traversal process between the points of the shape being generated.

Very small pieces of code are responsible for generating each distinct shape in our demo. This brevity was made possible by the help of two classes: `Grid_Patch` and `Surface_Of_Revolution` (a special case of `Grid_Patch`). `Grid_Patch` works by generating a tessellation of triangles arranged in rows and columns, connecting the points by generating a certain predictable pattern of indices. `Grid_Patch` produces a deformed grid by doing user-defined steps to reach the next row or column, defined by two callbacks supplied from outside. Depending on the shape designer's goal the callbacks' operations could either be



(a) Figure 2.2 came from Surfaces_Demo. Here it is in a final render with a bump map and texture.

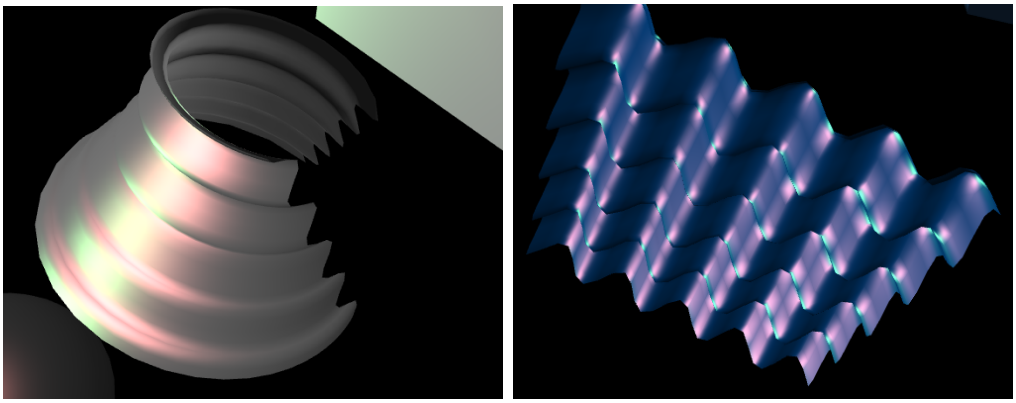


Figure 3.18: Close-up shots from surfaces demo.

functions of the previous point, pure math functions of the row or column index, or functions that sample provided arrays of points and interpolate curves in between them.

This demo draws spheres, domes, cones, polygons, other surfaces of revolution, and other arbitrary surface patches such what is shown in Figure 3.19: A shape made by blending one curve into another while also transforming it incrementally through space.

All shapes in `Surfaces_Demo` are generated as a single vertex array each, sometimes by using class `Shape`'s compounding feature; this practice speeds up performance considerably. One shape, shown and described in Figure 2.1, is a set of three axes arrows drawn by compounding together many primitive shapes such as cylinders. It appears in a variety of other demos (such as the `Bases_Game` to demonstrate the location of local coordinate frames.

Our students often make use of our axis shape when building a scene to stop their

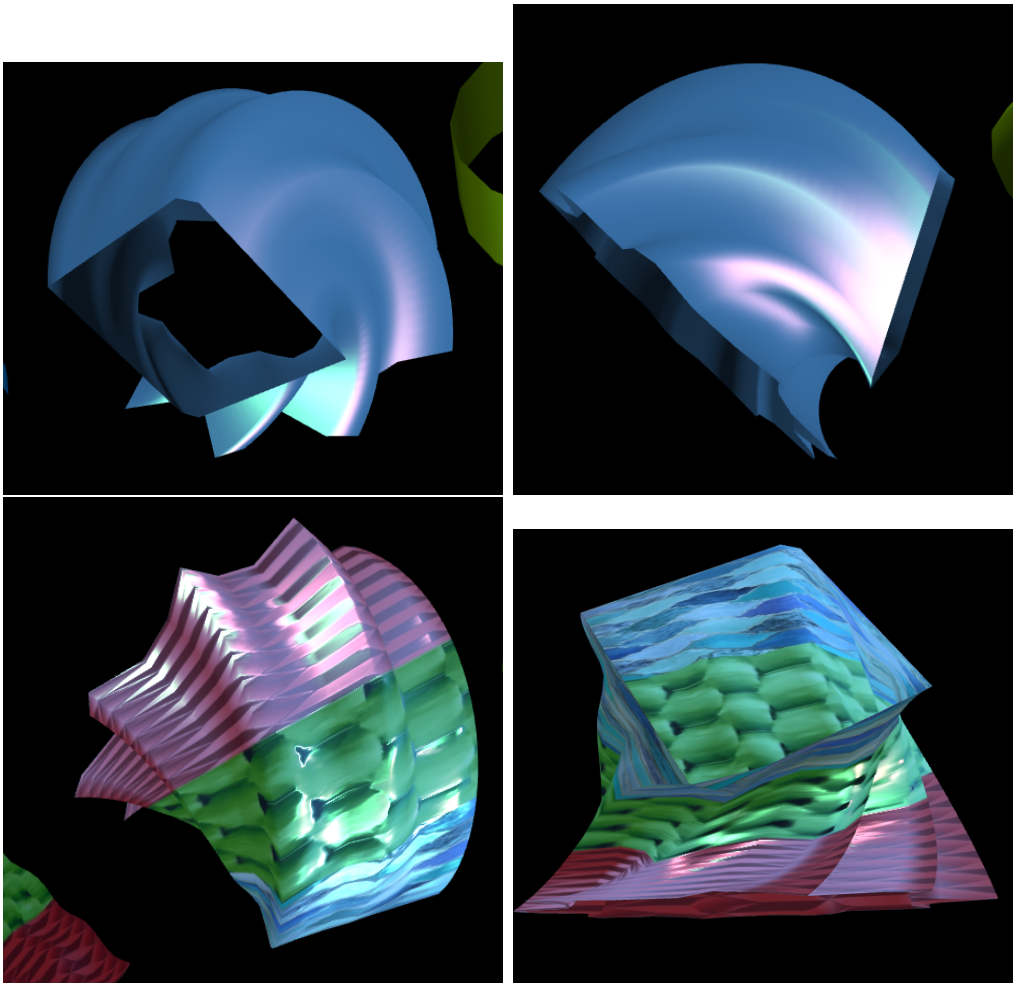


Figure 3.19: Two curve arrays (a square and a many-pointed star) are blended together here into a surface patch. Rather than using the typical linear blending method of surface patches, instead incremental transforms are used to smoothly rotate and translate from one curve's orientation to the next. This is only possible due to allowance of arbitrary operations between points (such as array sampling and incremental movement) by our Grid_Patch code.

flow and draw their local coordinate system whenever desired. This explicitly causes their intermediate steps on the way to their final basis to be drawn, greatly helping them with diagnosing transform problems. Our `Bases_Game_Level_2` article, as described above, also presents this same thought process as an educational game. Axis shapes are used to animate motion using transforms, including shapes that hinge off of one another in precise ways. Our visualization for students is similar to drawings of Denavit-Hartenberg (DH) parameters that are used in engineering (Denavit, 1955), an intuitive way of modeling the end effector of a linkage in terms of the rotations and translations leading up to it.

3.2.3.9 Star

The demo `Star` shows a simple special effect and suggests how to make similar effects. In this technique simple primitives are drawn in mass numbers at slightly different matrix transform offsets depending on the indices of programming loops. A versatile and beautiful kaleidoscope effect is the result.

3.2.3.10 Many_Lights_Demo

This is another simple special effect suggestion, this time explaining how to get around a limitation in shader programs such as our `Phong_Shader` class. Shader programs have a somewhat rigid structure since they execute in parallel and the hardware only has a limited number of data pipelines per thread. This means that the amount of light sources that contribute to the Phong reflection model's lighting calculations must be fixed in size as well. `Phong_Shader` only allows two light sources due to this small number of hardware data pipelines, and also due to the need to save computation time in the fragment shader, which has to run many thousands of times per frame.

`Many_Lights_Demo` shows how to make the illusion that there are many lights (despite the shader only being aware of two) by overwriting the lights between shape draw calls. This makes each given light configuration only affect certain shapes, such that only two lights are influencing any given shape at a time. For this to look right, it helps for shapes to be aware of which lights are nearby versus which are too far away or too small for their effects to matter, so the best pair can be chosen. In our demo scene, the camera moves past a grid of buildings in a city skyline scene. One light exists per row and one per column, and a box simply looks up the lights it is sharing a row or column with.

3.2.3.11 Simulation

Our helper class `Simulation` shows a good way to do incremental movements, which are crucial for making objects look like they are moving on their own instead of following a

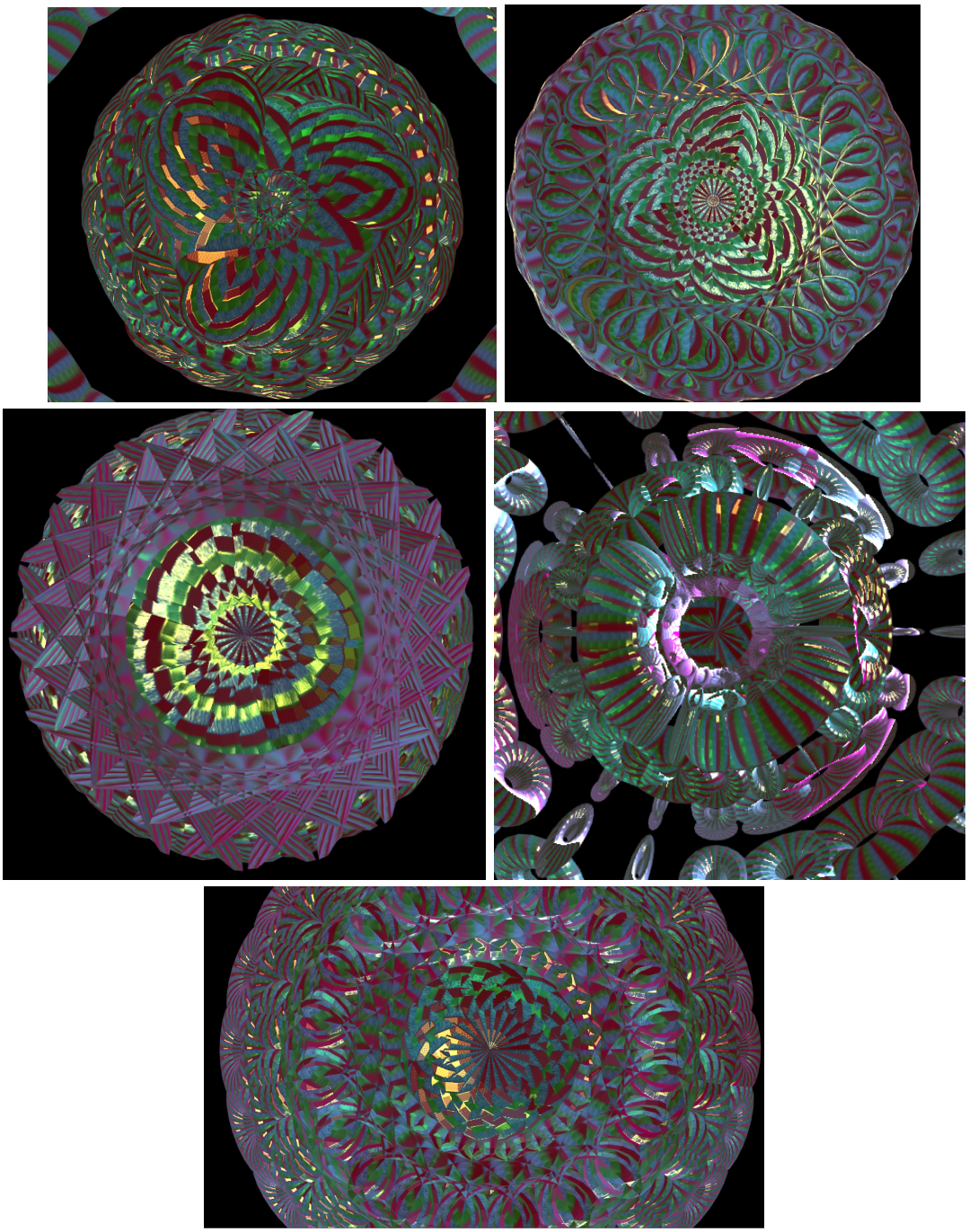


Figure 3.20: The Star demo.

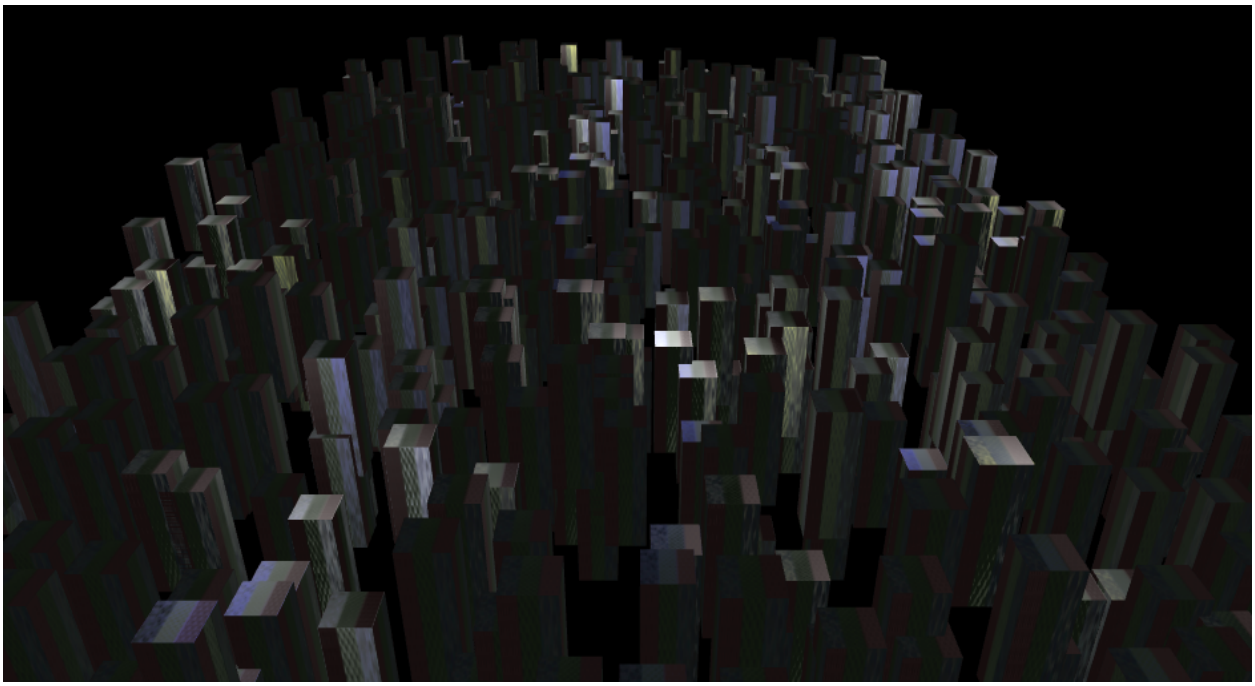


Figure 3.21: The Many_Lights_Demo.

pre-determined path. Animated objects look more real when they have inertia and obey physical laws, instead of being driven by simple sinusoids or periodic functions. `Simulation` is the underlying tool used to make the demos shown in Figures 3.22 and 3.23.

For each moving body, we store a model matrix somewhere persistent and give it a velocity to track over time, which is split up into linear and angular components. The latter is modeled as an angle-axis pair so that we can scale the angular speed how we want it. The forward Euler method is used by `Simulation` to advance the linear and angular velocities of each shape for each timestep (each unit of simulation time). The techniques described so far are the simplest possible way for a beginner to start showing moving rigid bodies. What makes `Simulation` stand out is instead the industry-based lessons it teaches on how to handle time.

This `Simulation` superclass helps other demos that feature physics simulation by carefully managing their stepping of simulation time. It totally decouples the whole simulation's movement of bodies from the frame rate, advancing the effects of physics in fixed time increments following the suggestions in the blog post "Fix Your Timestep" by Fiedler (2004).

Buttons allow the viewer to speed up and slow down time to create the illusion to the simulator that the display frame rate is running fast or slow, independent of the simulation time step size. The simulation's answers will be unaffected. Our code squeezes in as many simulation steps as possible per second of the program's execution time until we're caught up for the next frame. It takes extra measures to avoid what Fielder calls the "spiral of death" by setting a hard limit on the amount of time we will spend computing during any timestep if display lags. Here we also store an interpolation factor for how close our frame fell in between the two latest simulation time steps so that we can accordingly interpolate the drawn positions of bodies by blending the two latest states and displaying the result.

A readout shows how many total simulation steps have been calculated. By slowing down the simulation time enough, the reader can verify that less than one timestep is occurring per second even though the slowed-down movement remains fluid and continuous, demonstrating that interpolation is being used in between simulation steps to come up with intermediate states of the drawn result without having to run the potentially expensive simulation code more times than necessary.

3.2.3.12 Inertia_Demo

This is a minimal demo built on top of class `Simulation` to show its principles of incremental motion at work, including the buttons it provides for manipulating time. The demo shows colorful bodies of various shapes, that each respond to physical forces. It applies random initial velocities and simply lets momentum carry the bodies until they fall and bounce. The velocities are not subject to any forces but have a downward acceleration. Collisions of the geometric volume of one shape into another have no effect. Velocities are constrained to not take any objects under the ground plane.

3.2.3.13 Collision_Demo

`Collision_Demo` detects when some flying objects collide with one another, coloring them red and stopping them in place when they do. To keep things moving, the objects are

Figure 3.22: The Inertia_Demo.

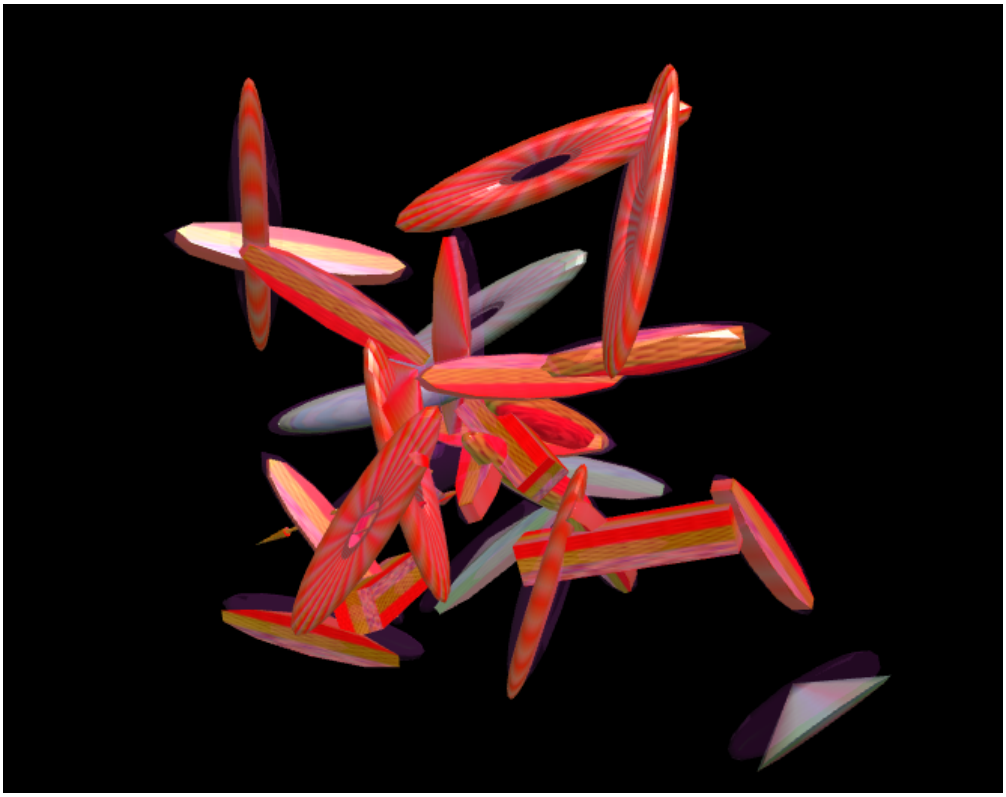
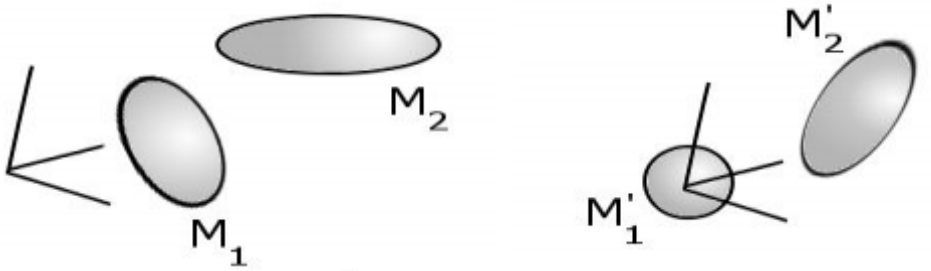


Figure 3.23: A number of moving rigid objects, sticking together when intersections are detected in the Collision_Demo. Objects stuck together form a structure or pile (red), while fast-moving free floating objects (white) have not yet collided.



(a) Before change of bases (b) After change of bases

Figure 3.24: Our trivial collision method. We perform a change of bases into a more convenient space where one of the objects conforms to the non-transformed unit sphere as much as possible. In that space, points too far from the origin can be disqualified as colliding.

occasionally assigned random velocities, and a force pulls them towards the center leading to orbital motion. Special volumes that are used for testing collisions are drawn in translucent purple alongside each shape's true drawing.

Detecting intersections between pairs of volumes under arbitrary linear transformations can be difficult, but it is made easier by being in the right coordinate space. Our collision algorithm for beginners treats every shape like a ellipsoid roughly conforming to the drawn shape including its rotation, due to having the same transformation matrix applied.

Our particular collision method uses only a few lines of code and is shown in Figure 3.23. In essence it combines a change of bases with a vector norm used on several points sampled from a sphere. The simplicity of our volume intersection test allows the student to focus on the structure of a collision algorithm and considerations such as how many possible intersection pairs are tested, and how to reduce that search space and computational complexity class. The algorithm has problems, though. Making every collision body a stretched sphere is a hack and does not handle the nuances of the actual shape being drawn, such as a cube's corners that stick out. Looping through a list of discrete sphere points to see if the volumes intersect is even more of a hack (there are perfectly good analytic expressions that can test if two ellipsoids intersect without discretizing them into points, although they involve solving a high order polynomial). On the other hand, for most non-convex shapes a real collision method cannot be exact either, and is usually going to have to loop through a list of discrete

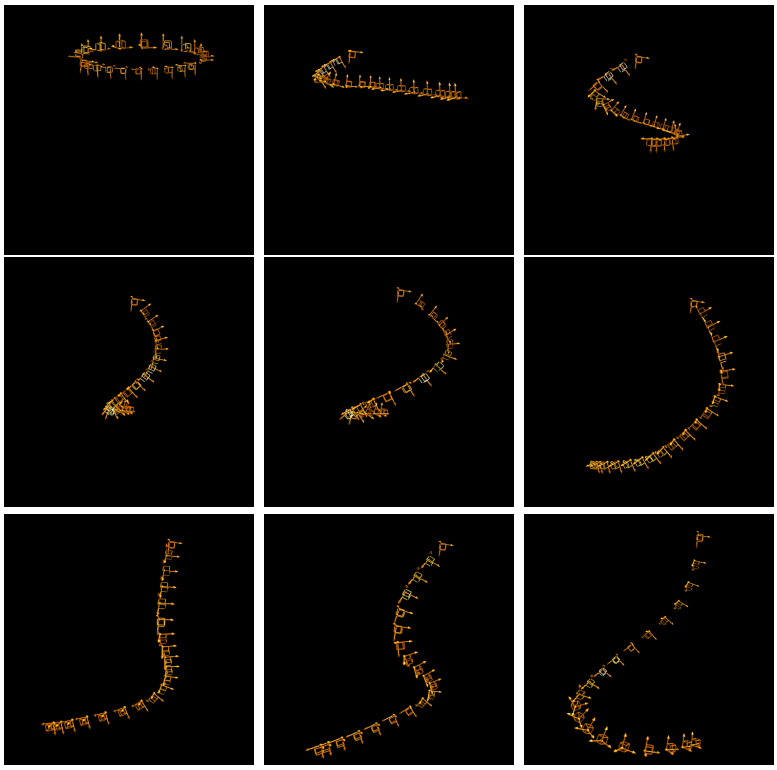


Figure 3.25: A falling string in the Springs_Demo.

tetrahedrons defining the shape anyway.

3.2.3.14 Springs_Demo

This demo illustrates more applications of physics simulation, Hooke's law of springs from physics, and the concept of numerical instability. It shows nodes of a falling 1D string. It steps a simulation of Hooke's law in fixed time increments using the forward-Euler method. Using similar mechanics as `Simulation`, the user can press buttons to create the illusion to the simulator that the display framerate is running fast or slow, independent of the simulation time step size. No matter the amount of slowdown, the simulation's answers will be unaffected and the slow count of timesteps computed so far will show that the smoothness of animation does not depend on new potentially expensive timestep calculations.

Parts of this demo are left for the visitors to fill in as an exercise, such as adding in a small non-zero rest-length and a series damper to each spring segment or altering which nodes are constrained in place. Another suggested exercise is to reduce the fixed timestep length in

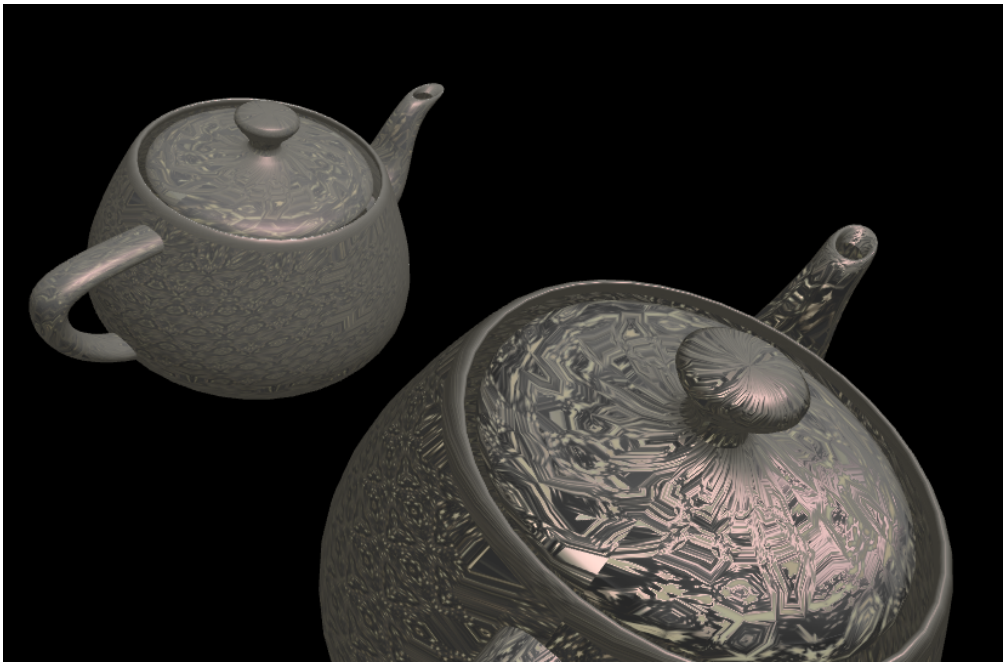


Figure 3.26: Two teapots imported from text files by the Mesh_Loader_Demo. One is shown with bump mapping and one without, to further help educate viewers about the difference. They have differing light response while they move in the animation.

the code from .002 to .0002 seconds, causing more discrete physics steps to be computed. The student can observe that the framerate slows down but the simulation's answers are not noticeably different. Lastly, the student is encouraged to see how many nodes they can add to the string before the forward-Euler method breaks down and numerical error of their added damping term catastrophically accumulates, exploding the position values to infinity.

3.2.3.15 Mesh_Loader

Students often want to import pre-made shapes into their animation project rather than making all shapes in their scene from scratch. These shapes from outside modeling software or from downloaded files containing famous models of widely recognized characters. Our `Mesh_Loader` class and demo article provides a simple way to do that, using a copy of the online library "webgl-obj-loader.js" (frenchtoast747, 2015) that is massaged into our code library. Particularly, our modified copy uses a more generalized interface for reading strings from files so we can re-use it in other code.

3.2.3.16 Keyboard_Demo

As described in more detail back in Section 2.2.3, our `Keyboard_Manager` class in tiny-graphics.js duplicates all features of the "shortcut.js" library by Binny V A (Abraham, 2012). The utility adds keyboard interactions and shortcuts to any website, which is especially useful for interactive editors and games. Ours tracks combinations of keys the user has held down and binds them to arbitrary behavior. As a single standalone class it can be immediately beneficial for all other JavaScript programmers, so we present `Keyboard_Demo` as a splash page for it.

The `Keyboard_Demo` is a Secondary Scene Component that uses several `Keyboard_Manager` objects to show off their capabilities. Keys can be captured from the entire document or from child HTML panels, with separate `Keyboard_Manager`s for each that fire their mapped callbacks if their paired document element is the target of the key press (because the user has clicked it into focus). The regions are shown to not interfere with typing into text input boxes. A readout reports the set of keys that are considered pressed down in each region. One example `Keyboard_Manager` bound to the document cancels key events from triggering the default browser keyboard shortcuts, while another `Keyboard_Manager` passively monitors the events without cancelling; instead, it logs them for building a record of a user's experience. As a Secondary Scene Component this logging behavior can be stacked alongside any other of our demos when this class is requested.

3.2.3.17 Text_Demo

As discussed in Section 2.2.2, there is no trivial way provided by WebGL to render text in the 3D coordinate space using the built-in interface. This is because all drawing routines of WebGL assume shapes will be made of points, lines, or triangles. The designers of WebGL probably observed that text can simply be provided by the surrounding web document, or if a heads-up display is desired, even displayed on top of the canvas using offsets. This still leaves open a common student question during free-form animation projects: How to show text statically on 3D objects. We therefore seek to give programmers back the text rendering

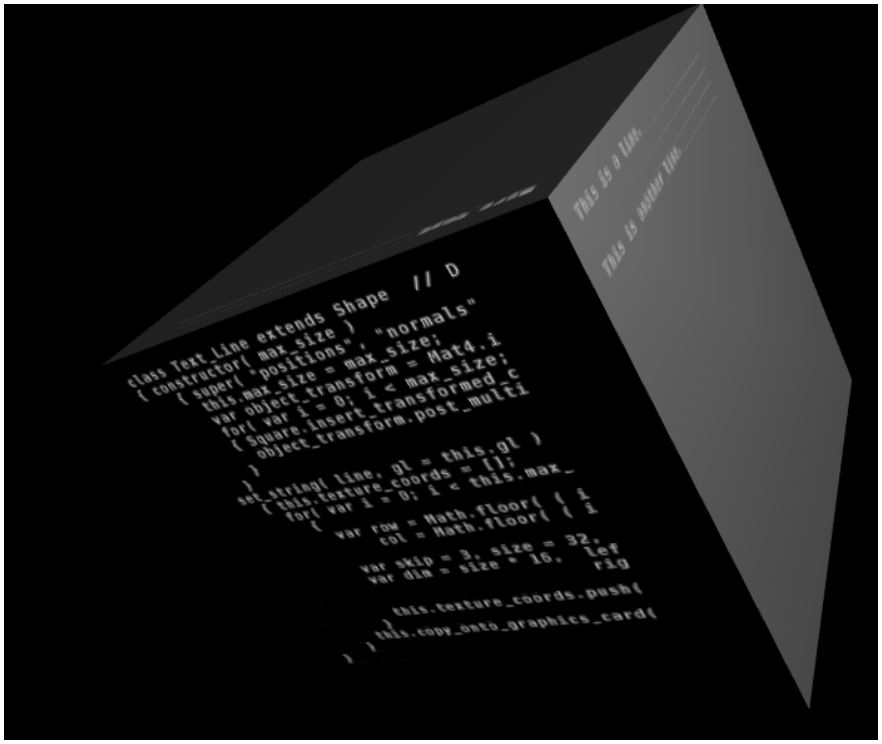


Figure 3.27: The Text_Demo.

functionality that all OpenGL editions once had.

In a sub-type of class `Shape` called `Text_String`, we design a traditional planar shape that can show text on a transparent background. `Text_String` uses texture mapping and carefully aligned pieces of an image of the ASCII character set to do this. Our article `Text_Demo` explains how to use this class and displays several text strings on the sides of a gray cube, including multi-line strings.

3.2.3.18 Scene_Graph_Tool

Scene graphs are a major topic of Computer Graphics, with an entire chapter devoted to them in Angel's textbook (Angel and Shreiner, 2014a). They help students to not only organize and re-use parts of their scene, but to also conserve the amount of calculations their program has to do.

A scene graph is a tree data structure that represents the spatial layout of a scene and logical relationships between its parts. The nodes of a scene graph tree include the drawn

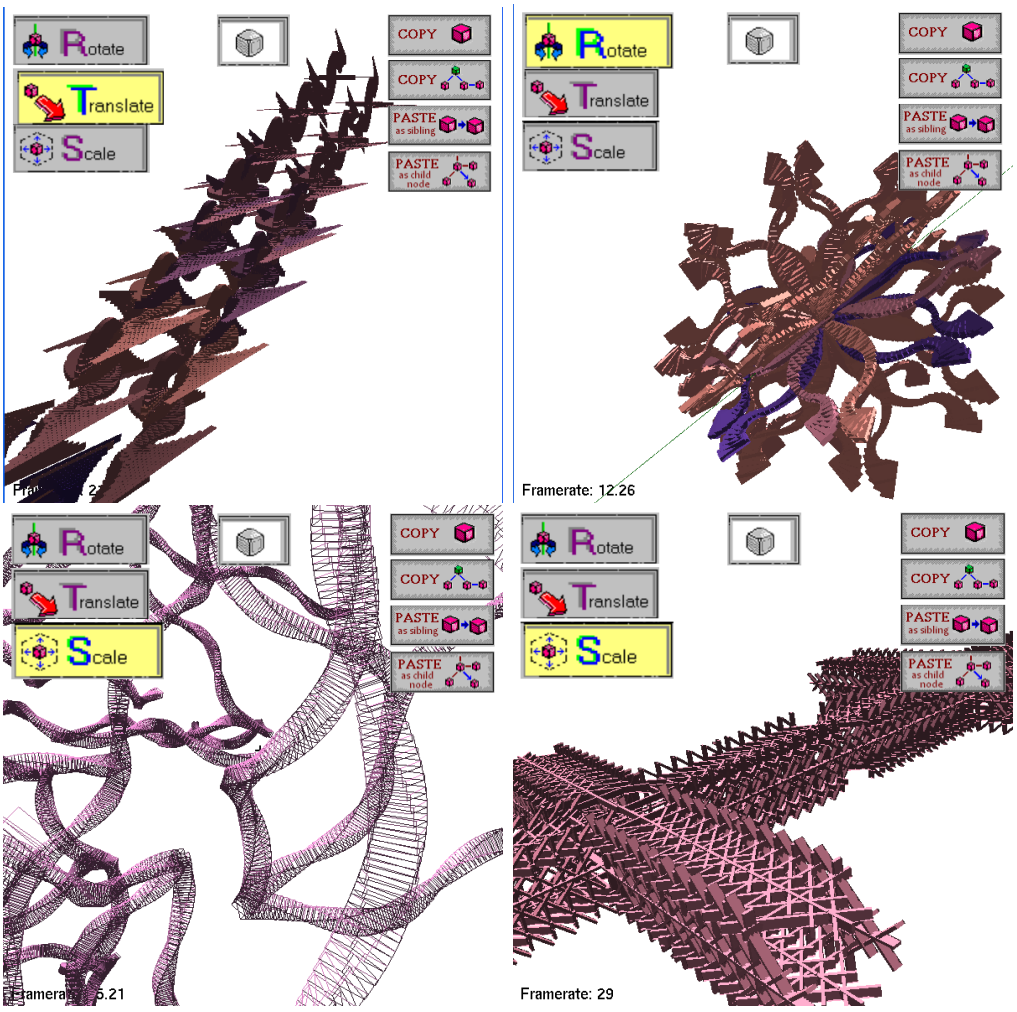


Figure 3.28: The Scene_Graph_Tool. Used here to model the above shapes by cutting and pasting branches of the graph, or repeating parent-child relationships and transformations in a pattern. These highly organized shapes were generally made in a blindly improvised fashion with few expectations about the outcome, by tapping the UI buttons for a minute or so.

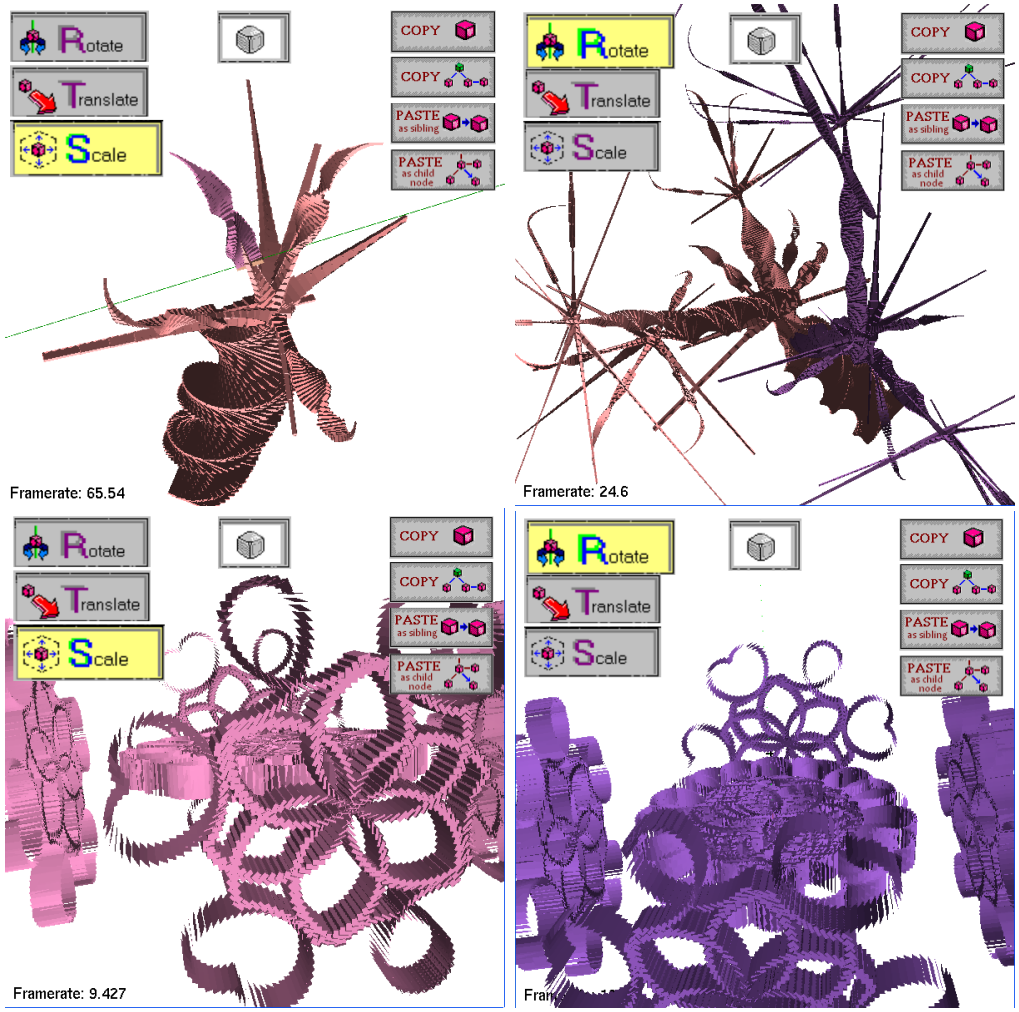


Figure 3.29: The Scene_Graph_Tool (Continued). Used here to model the above shapes by cutting and pasting branches of the graph, or repeating parent-child relationships and transformations in a pattern. These highly organized shapes were generally made in a blindly improvised fashion with few expectations about the outcome, by tapping the UI buttons for a minute or so.

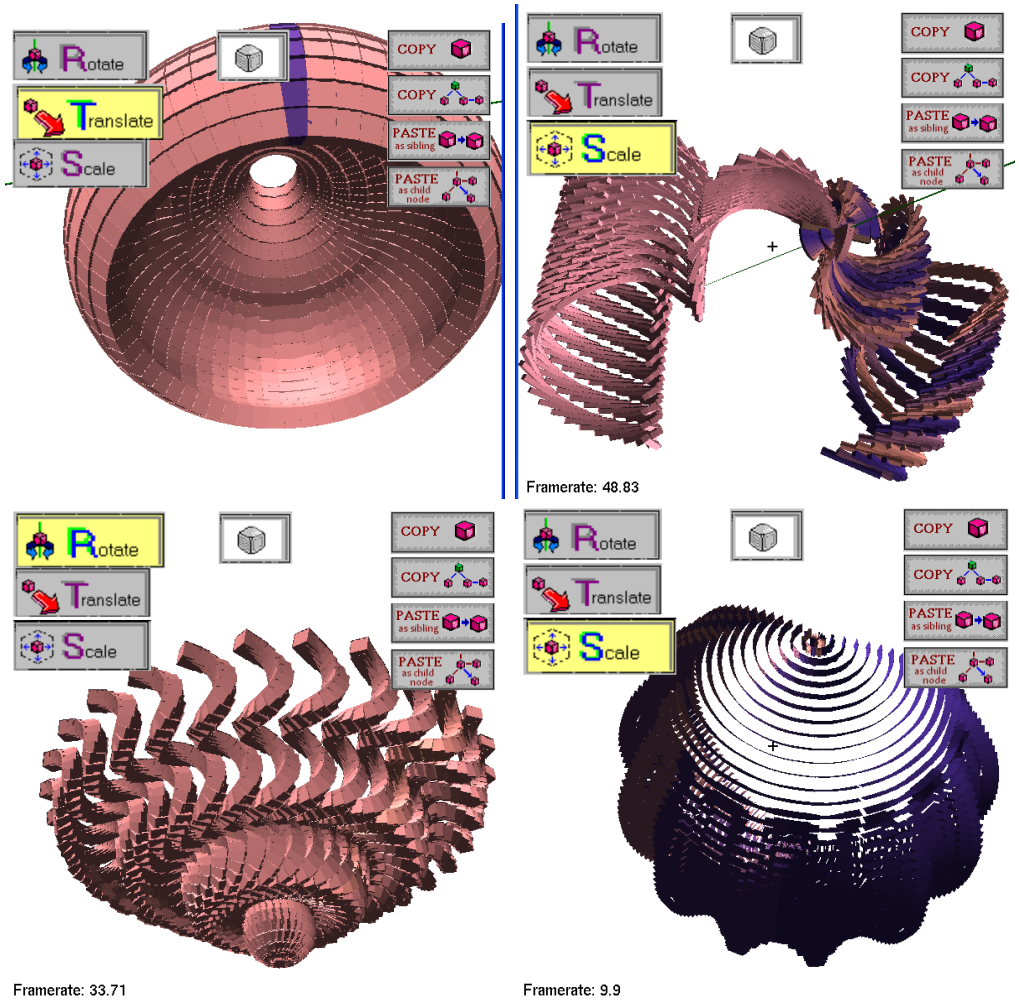


Figure 3.30: The Scene_Graph_Tool (Continued). Used here to model the above shapes by cutting and pasting branches of the graph, or repeating parent-child relationships and transformations in a pattern. These highly organized shapes were generally made in a blindly improvised fashion with few expectations about the outcome, by tapping the UI buttons for a minute or so.

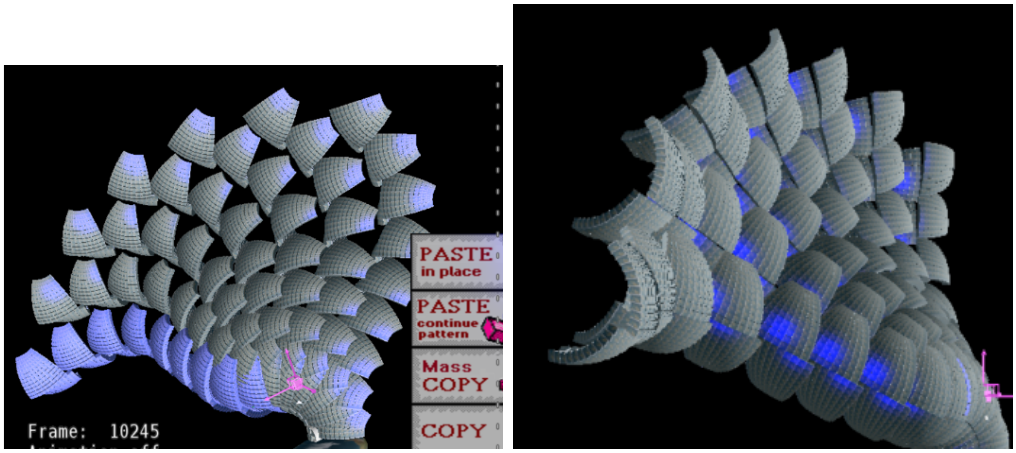


Figure 3.31: The effect of changing the scale matrix of the root node of the Scene_Graph_Tool is shown here. Observe how this also grows the gaps between individual cubes (whether they are touching or not), yet avoids any undesired shearing behavior since the rotation matrices down the tree are kept from being affected.

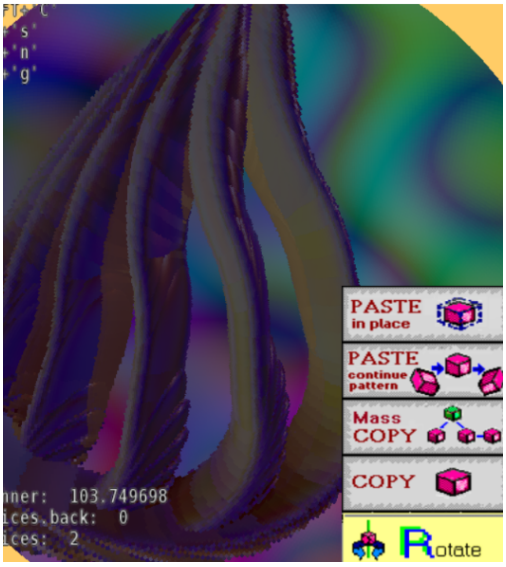


Figure 3.32: An example of our Scene_Graph_Tool being used as a Secondary Scene Component. This image of a shape modeled using a scene graph is actually ray traced. The primary demo running is the ray tracer, while the Scene_Graph_Tool stacks alongside it and contributes its usual user interface tools for growing shapes out of nothing. The primitives composing the shape arrive inside of the same array that the rays are tested against. Rays that struck nothing show a sinusoidal color pattern.

shapes and other entities (such as camera locations) with relations to a scene's particular parts. A node knows its own location or position in space. This location is a transformation matrix stored as an offset relative to its parent node—local to the parent node's coordinate frame. The relative properties of nodes accumulate down the tree; any matrix operation performed higher up in the tree dynamically propagates its effect downward through the branching "child" relationships to all the leaf nodes below. This happens by affecting the final matrix product as multiplications accumulate while the tree is traversed.

Among scene graph implementations, ours is noteworthy for pairing the scene graph with a user interface that emphasizes immediate feedback. It allows live editing of the graph's nodes. It especially encourages re-use of nodes and entire branches, allowing the automated repetition of nodes with the touch of a button. Adding nodes to grow the branches of the graph implies that the transforms get repeated in a pattern as well, sending the nodes down a path that might curve and spiral as the repetition goes on. The tool can create highly symmetric shapes by procedurally re-attaching a scene graph's entire branches to other parts of a tree, or duplicating them around the tree. See Figures 3.28, 3.29, and 3.30 for examples of interesting shapes made this way. Our novel tool builds upon industrial modeling tools, as discussed further in Section 5.6.

Button or key controls let the user either grow the scene graph or modify the current node's current transformation matrix freely, allowing the user to choose any axis and then rotate, translate, or scale the local coordinate frame. The current graph node's shape immediately redraws, along with any child nodes' shapes farther down the branch. Changes to the scale matrix of a node cause all gap sizes between shapes throughout the branch to immediately change, providing striking visual feedback and unexpectedly interesting new shapes.

Our `Scene_Graph_Tool` stores a traditional scene graph with some caveats. Rotation matrices are automatically unaffected by scales higher up in the hierarchy to prevent unwanted shear effects, but translations are handled separately and do get the scale effects so that gaps between shapes correctly get bigger as the scale of a graph branch increases. Secondly, unlike the Angel textbook's suggestion of implementing a scene graph using a pointer-based

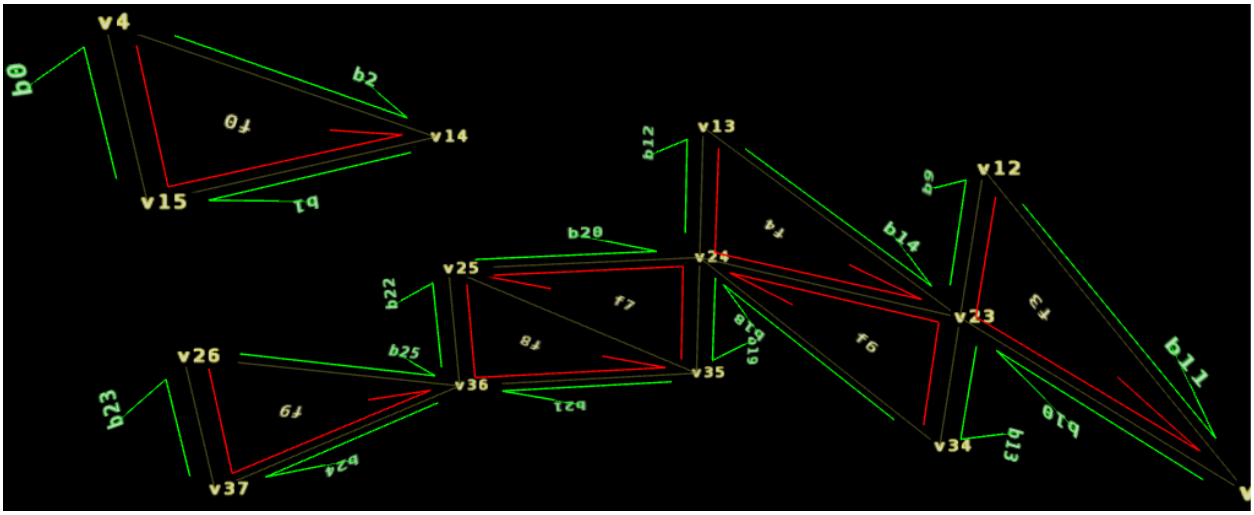


Figure 3.33: The Half_Edge_Demo. This shows our labeling conventions that illustrate the orientation of each interior half-edge (red) and border half-edge (green), to ease manual checks of the mesh neighborhoods.

binary tree, we instead use a simpler data structure made of dynamic arrays nested within one another, resulting in a much shorter program.

`Scene_Graph_Tool`'s accompanying article for students reinforces concepts about matrix ordering. It explains why a scene graph must build its matrix product using post-multiplication in order to correctly place child objects local to the coordinate system of their parent. Post-multiplication is the natural choice when modeling shapes, since most objects exhibit a parent-child hierarchy of sub-parts (such as a tree with a root and branches, or a human with a torso and limbs). It would be difficult to build hierarchical structures using pre-multiplication instead, because the intermediate products on the way to the child objects would be meaningless for drawing purposes. With post-multiplication, however, the intermediate products equal the parent matrices and can be used to draw the parents on the way to computing the final child product, requiring only one matrix multiplication operation instead of many to compute each object's final matrix while the tree is traversed.

3.2.3.19 Half_Edge_Demo

In traditional polygon based graphics, triangles are usually processed separately from one another on their way to being drawn. The triangles are stored as triples of integers (indexing

into positions in the vertex lists), and each triplet entry is separately unaware of the others. No other information is stored that would be helpful for queries such as what triangles are neighbors to each other in the larger shape. A mesh data structure attempts to alleviate that deficiency by adding additional data to the description of the mesh of triangles so that its neighborhoods or borders can be traversed using constant-time operations. These structures can be generalized from 2D triangles to 3D tetrahedrons. Mesh data structures can speed up a graphics program dramatically. They can provide immediate constant-time answers to ray-volume or volume-volume intersection queries by simply walking the neighborhood of each volume, in place of linear or logarithmic time search algorithms. We discuss our plans to explore this in Chapter 6.

`Half_Edge_Data_Structure` is an implementation of the array-based mesh data structure described in Tyler Alumbaugh's thesis (Alumbaugh and Jiao, 2005). This array-based method represents the whole pointer-free mesh as a few flat integer arrays for easy serialization to multiple processors.

Our flat integer arrays are stored compactly with JavaScript's newer raw array types such as Int32Array. Our version of the data structure includes a feature shared with the *El Topo* mesh library (Brochu and Bridson, 2009) wherein vertices are allowed to be non-manifold, for example during intermediate construction of a single triangular face, or when faces are added or removed in non-contiguous order.

`Half_Edge_Demo` shows an animation of `Half_Edge_Data_Structure` at work as it progressively integrates new faces into an initially empty triangle mesh. Each face and vertex are identified using in-world labels with the help of our `Text_String` class described in Section 3.2.3.17. A special arrow notation shows the ordering of half-edges of each face implicit in the data structure, and also the relationships of boundary edges to their opposing interior half-edges. A modern JavaScript language feature called generators allows the complex mesh-construction functions to be stopped in between operations and drawn at the intermediate stages, or paused if a problem is encountered for easy diagnosis at the moment the mesh's indices are detected to no longer encode the correct connections.

CHAPTER 4

Results

We explore the value of our contribution to date in a university setting. Later we will also address its potential value given our plans to scale it up into a true encyclopedia by crowd-sourcing both new articles and the CPU time needed to run Smart Articles.

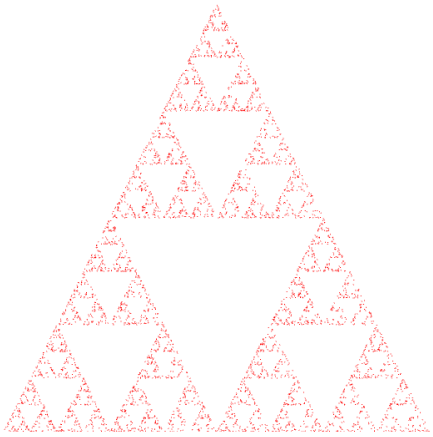
Our more practical measurement of today's benefits begins with a side-by-side comparison of Angel's supplementary web demos held up to our own improved results when building similar demos using our framework. We show that armed with our tiny library, we successfully built demos that are more advantageous to show in a university course.

We will then show the results of a case study at UCLA involving student usage of our tiny-graphics.js library and the online resources and tutorials on the Encyclopedia of Code.

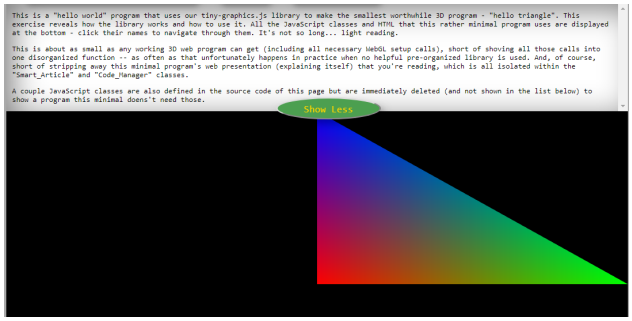
4.1 Comparison to Angel's Online Examples

Angel's textbook is supplemented by a large number of online demos broken down by chapter (Angel and Shreiner, 2014a). In his textbook he occasionally references these demos to illustrate a particular bit of knowledge, directing students to visit the demo's URL. Although there are some extra topics covered (especially relating to 2D image construction), his online demos largely have the same topic coverage as many of our own online demos covered so far from the Encyclopedia of Code, and share the same goals.

Here, we will show a one-to-one correspondence between several of our demos and Angel's, and make the case for ours being an improvement both in terms of what they illustrate and in terms of the usability and simplicity of their code. The latter is especially important because we observe students typically begin coding their projects on top of an already working demo



(a) Angel's "gasket1" demo (Angel and Shreiner, 2014c) (Chapter 2) (Angel and Shreiner, 2014a).



(b) Most similar to: Our Minimal_WebGL_Demo article.

Figure 4.1: Angel's "gasket1" demo.

rather than wholly from scratch in empty files.

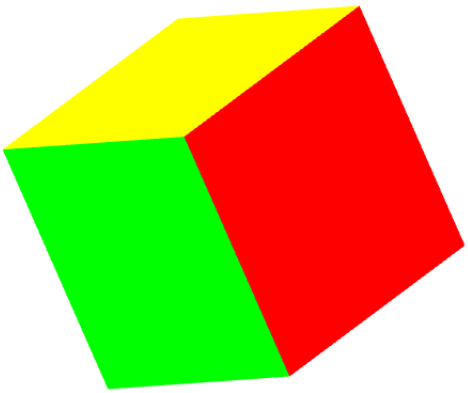
Besides the individual comparisons, keep in mind that Angel's demos all have the inherent limitation of existing on a separate medium from the textbook. Students must interrupt their reading and switch to a computer to use them, and therefore they will be tempted to view just one or the other. Our website seeks to get around this structural restriction. Our "Active Textbook" articles insert our demos right alongside the text and illustrations they are designed to explain, providing more value and convenience to students.

Because we used tiny-graphics.js, our results include improved educational demos that are more useful code-wise for students to build upon compared to using only Angel's bare bones library. Our interactive page also encourages live code editing and sharing, which gives another advantage over Angel's demos even if their educational value was equal.

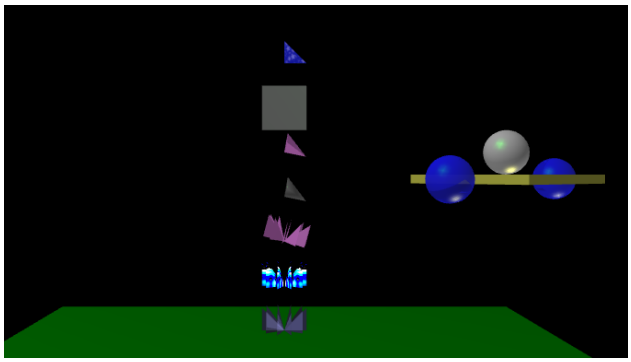
Our comparison excludes demos on Angel's website that currently appear to be non-working.

4.1.1 Angel's "gasket1" Demo

The first full program that Angel's textbook offers to students draws a Sierpenski Gasket fractal, using a canvas, a WebGL context, and simple JavaScript loops. He spends much of



(a) Angel's "cube" (trackball) demo (Angel and Shreiner, 2014m) (Chapter 4) (Angel and Shreiner, 2014a).



(b) Most similar to: Our Tutorial_Animation article.

Figure 4.2: Angel's "cube" (trackball) demo.

Chapter 2 of his textbook incrementally presenting this demo, explaining each line so that, by the end, the student fully understands one complete minimal WebGL demo. Itself an admirable goal, the incrementalism results in a program structure that does not separate common code from the uncommon into flexible, reusable components. Due to his book predating modern JavaScript, it also includes ill-advised steps like embedding shaders in the HTML file.

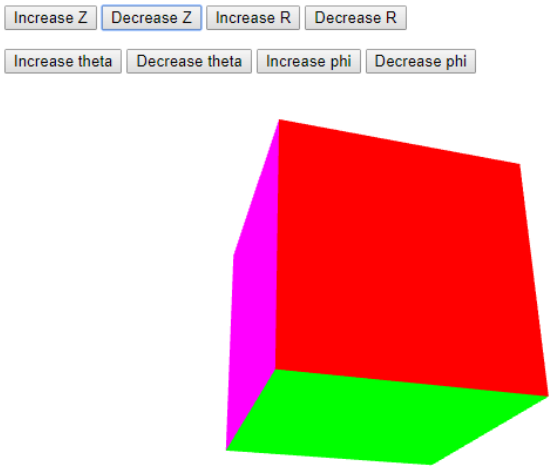
In contrast, our `Minimal_WebGL_Demo` uses a subset of tiny-graphics.js to do the same job, which as detailed in Section 2.2 organizes the boilerplate code of WebGL anticipating later reuse. This means that programmers starting from our demo may dynamically switch between vertex arrays (shapes), shader programs, textures (images), and entire scenes within each canvas. Our minimal shader is also more useful and draws triangles instead of points as in the gasket; Angel's usage of a gasket fractal does not add much to the tutorial besides a primer to JavaScript, which we cover elsewhere on our website.

4.1.2 Angel's "cube" (Trackball) Demo

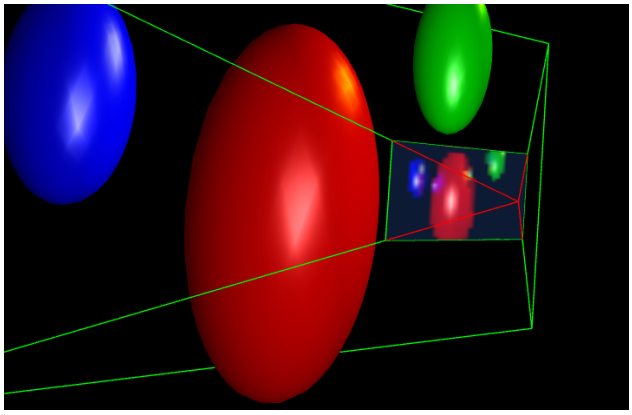
This simplistic cube with flat coloring and no lighting happens to be Angel's first example program where a matrix is used for drawing. As such, before tiny-graphics.js existed most of our students started with this to base their term project on rather than manually going through the steps of his Chapter 2. Here Angel provides them with working code for passing their matrix through to the graphics card to calculate final point positions. Angel's sub-example called "trackball" additionally provides mouse control over the camera matrix in the same spirit as our `Movement_Controls` tool.

Our `Tutorial_Animation` demo gives students a much more flexible foundation to build their project off of, since we observed a trend of that happening with the equivalent Angel demo. Compared to Angel's cube demo `Tutorial_Animation` comes with more demonstrated shapes, Phong shading, our helpful `Movement_Controls` secondary scene component, and more. Its reduced version, `Transforms_Sandbox` (from Section 3.2.3.4), does all that as well while maintaining a very small total footprint of code that students need to worry about. The section where students add code to build matrices and place shapes is segregated away from all other code, which they can either ignore or learn at their own pace, after they already have understood the basic flow of a graphics program and how to work with matrices.

The shapes displayed in `Tutorial_Animation` are arranged in code in the same order and presented as a progression of how to model increasingly complex shapes, interwoven with detailed code comments explaining the craft of 3D modelling to students. The shapes display onscreen in the same order. Shape modelling is the primary gateway to graphics programming offered by `Tutorial_Animation`, serving the same role as Angel's demo. Although Angel's demos explain shape modelling before matrices, we prefer to introduce our `Transforms_Sandbox` demo prior to this one, with the idea that matrix order concepts can be an even gentler initial gateway to graphics programming.



(a) Angel's "perspective1" demo (Angel and Shreiner, 2014f) (Chapter 5) (Angel and Shreiner, 2014a).



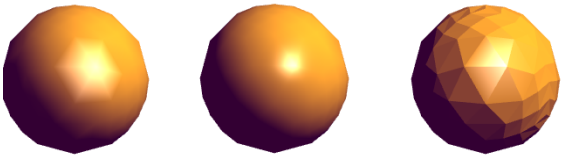
(b) Most similar to: Our Frustum_Demo article.

Figure 4.3: Angel's "perspective1" demo.

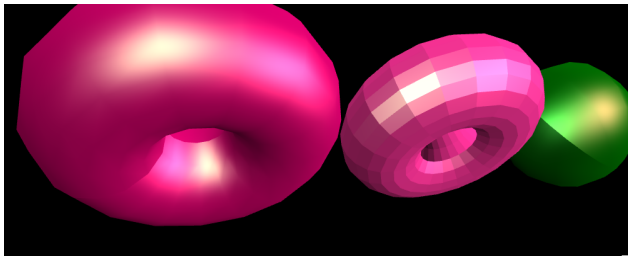
4.1.3 Angel's "perspective1" Demo

Angel's "perspective1" demo is another flat colored cube. This demo lets students see perspective foreshortening for the first time, wherein the cube's parallel edges do not appear parallel due to converging towards vanishing points. Besides showing this, there are no other clues provided as to what view volumes are. The cube jarringly vanishes before the student's eyes when it passes through the near or far plane using Angel's controls, but since they cannot see the volume directly, students who are trying to visualize a frustum for the first time here may have trouble.

Our `Frustum_Demo` quite succinctly shows students both the perspective effect and the view volume that caused it, in a superimposed drawing. Further demos using the `Frustum_Tool`, namely `Ray_Tracer` (from Section 3.2.2.5), illustrate in simple terms that the contents of the view volume are projected onto the volume's near plane, forming a 2D image. The code behind it illustrates the process of deconstructing any projection matrix using the canonical view volume. These concepts can either take a whole chapter of a textbook to explain, or can be done in just one interactive moving picture if given to the student to play with.



(a) Angel's "shadedSphere" demos (Angel and Shreiner, 2014i)(Angel and Shreiner, 2014j)(Angel and Shreiner, 2014k) (Chapter 6) (Angel and Shreiner, 2014a).



(b) Most similar to: Our Phong_Shading_Demo article.

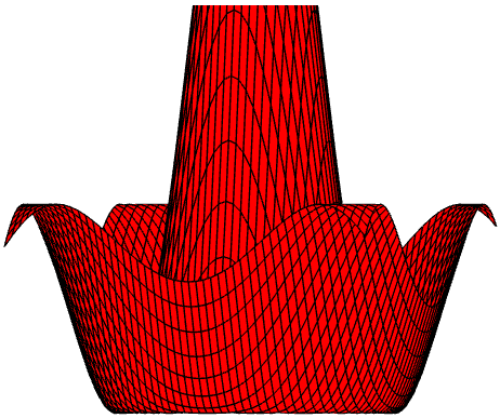
Figure 4.4: Angel's "shadedSphere" demos.

4.1.4 Angel's "shadedSphere" Demos

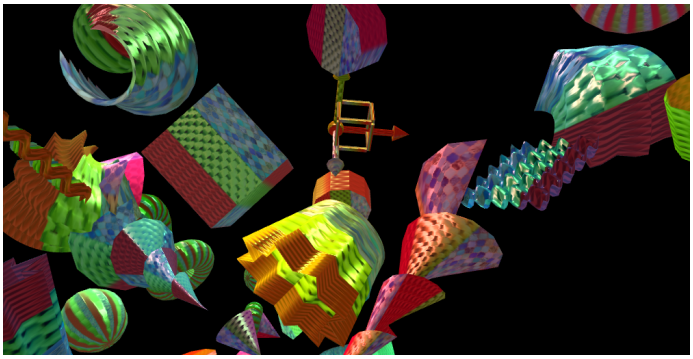
In three separate demos, Angel illustrates the differences between the flat, Gouraud, and smooth techniques of shading under the Phong reflection model. We alternatively direct students to our `Surfaces_Demo`, which demonstrates these three concepts in one central place. Compared to Angel's code, our `Phong_Shader` class's lighting calculations are decoupled from our shader code; this means it can be used in either the vertex shader or the fragment shader, producing smooth or Gouraud shading, respectively. Students can see the difference with a touch of a button and observe why it works in the code. Our Shape class's ability to automatically produce flat shaded versions of complex student-designed shapes both illustrates the concept of flat shading and takes the tedium out of designing a shape's triangulation to be flat shaded.

4.1.5 Angel's "hat" Demo

Angel moves to the topic of surface modeling with his hat demo, which shows students how to draw their first complex continuous shape. It is an arbitrary 2D height map function. Although those are convenient, most shapes a graphics programmer would like to draw are not mathematical functions or simple height maps. Even shapes that qualify as piece-wise functions might not have formulas that are known analytically. Angel's code can be adapted into new shapes by plugging in new functions of x and z (outputting y), but it is otherwise



(a) Angel's "hat" demo (Angel and Shreiner, 2014d) (Chapter 5) (Angel and Shreiner, 2014a).



(b) Most similar to: Our Surfaces_Demo article.

Figure 4.5: Angel's "hat" demo.

too specialized and would have to be reworked for making the 2D surface patches Angel describes in the final chapter of his textbook (Angel and Shreiner, 2014a).

Angel's code for this demo devotes fully 70 lines to nothing but defining the hat shape itself. A vertex array is constructed that has only one data field, position. The 70 lines of code do not separate any logic off that could be easily re-used for other similar shapes that also only have a position field.

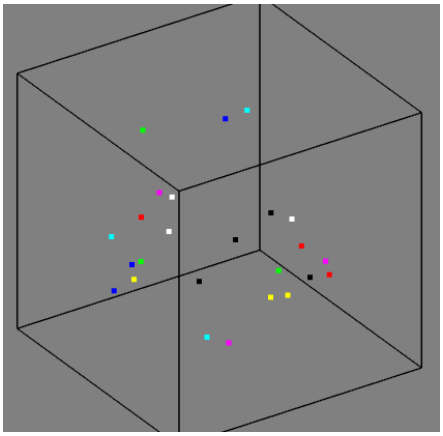
Our `Surfaces_Demo` is a far better introduction to drawing smooth shapes, and unlike the above it is even suitable for drawing the arbitrary surface patches that Angel devotes an entire chapter to. With the help of our approximately 50-line `Grid_Patch` class (from Section 3.2.3.8), students can quickly set up a variety of other custom shapes that need not be mathematical functions. For details on how "quickly" students can accomplish this, refer to Table 4.1.

4.1.6 Angel's "particleSystem" Demo

Angel shows particles responding to gravitational acceleration, Lennard-Jones attraction and repulsion forces, and velocity constraints to prevent passing through walls (Angel and Shreiner, 2014a). Our `Inertia_Demo` does the same things save for the Lennard-Jones forces,

Name	# Lines	Name	# Lines	Name	# Lines
Cylindrical Tube:	2	Open Cone Tip:	2	Torus:	9
Grid-based Sphere:	8	Regular 2D Polygon:	4	Closed Cone:	5
Closed Capped Cylinder:	6	Axis Arrows:	17	Five unusual surface patches:	22

Table 4.1: The amounts of lines of code required to create the various shapes in our Surfaces_Demo article. Each shape's code makes calls to our Grid_Patch class to generate surface patches. Students can see these highly optimized examples and learn to make their own shapes, with more functionality and less complexity than with Angel's closest example.



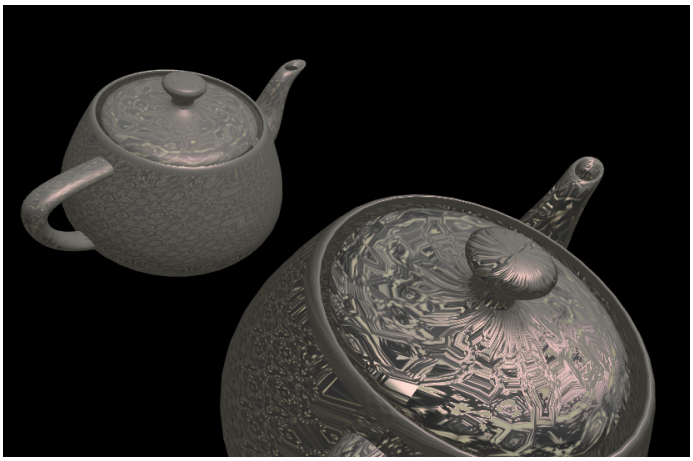
(a) Angel's "particleSystem" demo (Angel and Shreiner, 2014e) (Chapter 10) (Angel and Shreiner, 2014a).

(b) Most similar to: Our Inertia_Demo article.

Figure 4.6: Angel's "particleSystem" demo.

but with one more important difference: Angel's objects are not stored as matrices. They have no rotation or scales of their own, and as such, angular momentum is impossible to track. While rotational motions are difficult to track correctly, our demo assigns random angular velocities and still looks far more convincing to the untrained eye than if the same complex models were drawn with unchanging rotation as they fly around. As such, this allows our demo to illustrate physical forces with more complex objects than Angel. This benefit applies to students who re-purpose `Inertia_Demo` as well. Of the two, only our demo allows students to integrate physics effects into their own animations without being constrained to simple non-rotating dot shapes.

(a) Angel's "teapot5" demo (Angel and Shreiner, 2014l) (Chapter 11) (Angel and Shreiner, 2014a).



(b) Most similar to: Our Mesh_Loader article.

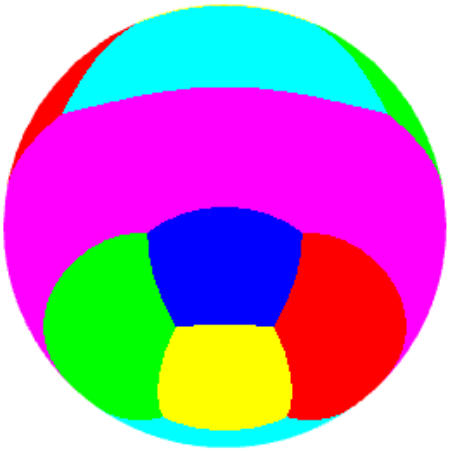
Figure 4.7: Angel's "teapot5" demo.

4.1.7 Angel's "teapot5" Demo

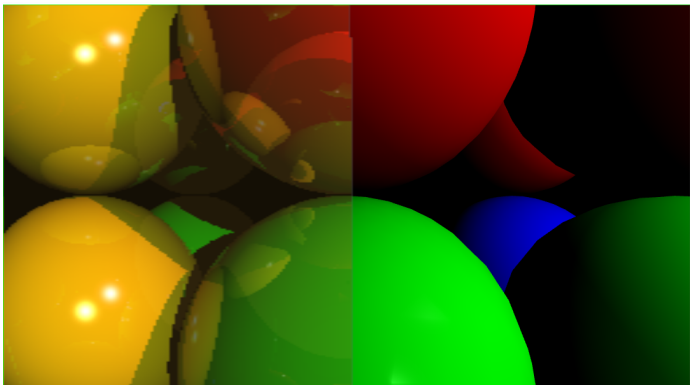
Angel demonstrates a teapot model not to show students how to import arbitrary objects such as the famous Utah teapot, but as an example in a chapter about surface patch generated shapes. The teapot's point data is hard-coded into the JavaScript code he provides, with no generalized means to import any other models. As discussed in Section 3.2.3.15, there is a student demand for placing pre-made shapes from the internet into their animation projects. Our `Mesh_Loader` class and demo article provides a simple utility class that can do that, allowing them to not only draw the same Utah teapot as Angel in this demo, but any other model from an .obj file as well.

4.1.8 Angel's "reflectionMap2" Demo

Angel's reflection map demo and our `Ray_Tracer` are similar in that they both require code that maps a rendered image into texture space for display on the surface a shape embedded in 3D. In Angel's case, the image is a simple reflection map, a re-rendering of the scene from a different perspective than the camera using traditional triangulated geometry. In our case, we use a much more intensive ray tracing process that allows us to draw a more



(a) Angel's "reflectionMap2" demo (Angel and Shreiner, 2014g) (Chapter 7) (Angel and Shreiner, 2014a).



(b) Most similar to: Our Ray_Tracer article.

Figure 4.8: Angel's "reflectionMap2" demo.

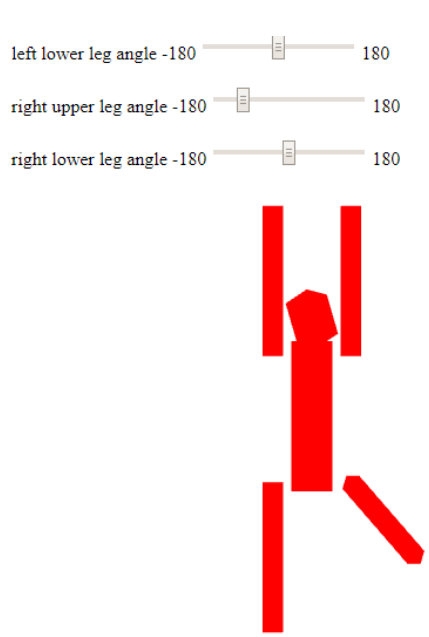
compelling effect that can include deeper levels of reflection, refraction, and shadows cast by other objects.

4.1.9 Angel's "figure" and "robotArm" Demos

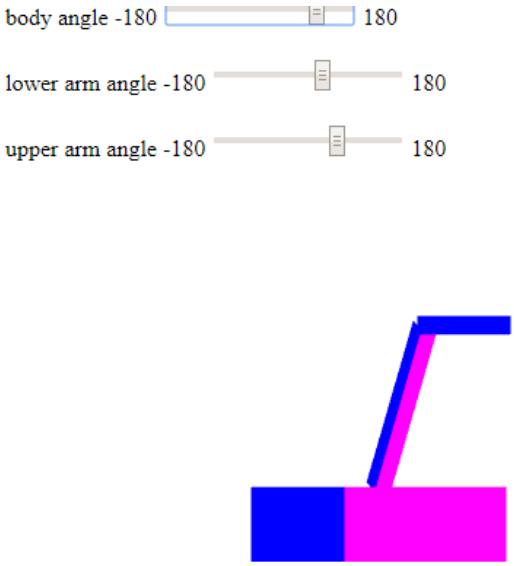
Scene graphs were introduced in Section 3.2.3.18. Angel's "figure" and "robotArm" scene graph demos show the basics of one-level and two-level hierarchical objects, respectively.

Due to the low quality of these demos it is not even initially apparent that the objects are 3D. The fact that the solid-colored shapes are not flat but are in fact cubes is only visible after a specific sequence of control inputs that achieve rotations. This must be done manually on multiple axes applied relative to the parent node in the scene graph. Angel's scene graph goes at most two branches deep and cannot be dynamically edited.

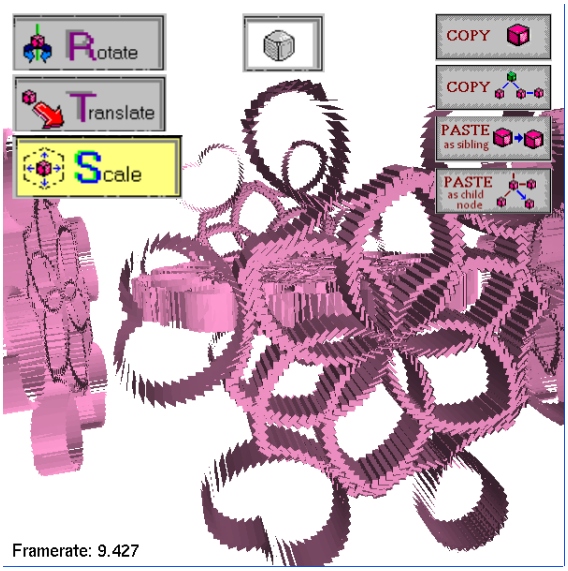
This demo does not make a very convincing case to students that they need scene graphs as opposed to just manually maintaining a current transformation matrix in easier ways. Normally a student could achieve their matrix hierarchy by simply entering and exiting scopes of JavaScript functions and passing the current matrix. Here each deeper call would serve to travel down the graph branches their scene is conceptually divided into. Even when



(a) Angel's "figure" demo(Angel and Shreiner, 2014b).



(b) Angel's "robotArm" demo(Angel and Shreiner, 2014h).



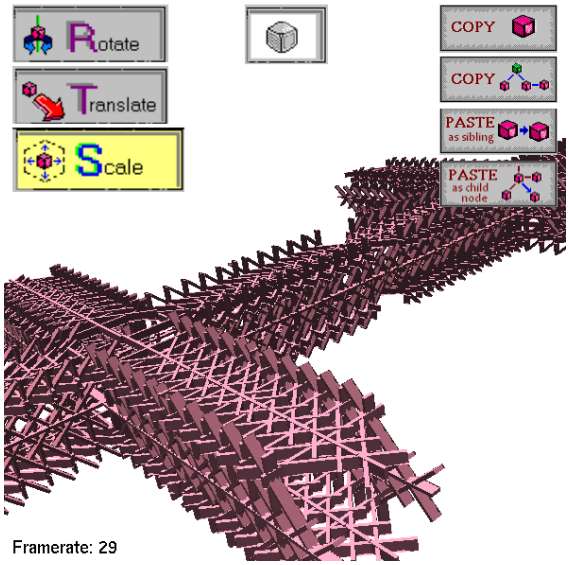


Figure 4.9: Top: Angel's two "scene graph" demos (Chapter 9) (Angel and Shreiner, 2014a). Bottom: Our Scene_Graph_Demo article.

students must store their matrix history, this only requires manipulating a stack array, not a whole scene graph.

Our `Scene_Graph` demo solves all these problems and many more by allowing arbitrary and fast graph edits through a novel user interface. The visually compelling shapes that result are more rewarding and informative. The `Scene_Graph` demo is one of many tools we provide for building intuition about both matrix order and how to store and manage the many matrices of a scene.

In conclusion, our demos are each similar to Angel's but go farther to help students. We now move on to our observations when using tiny-graphics.js in a real class.

4.2 Case Study: A Graphics Course Before and After tiny-graphics.js

We observed the UCLA Computer Science class 174A "Introduction to Graphics" before and after introducing tiny-graphics.js and its demos. We recorded differences in the outcome by comparing the success of the term project assignment, based upon quality of the animations that were turned in. Directly comparing the projects is difficult because the criteria that constrained the projects were subjective; most of the grade for the animations came from scores for creativity, complexity, and quality as deemed by the teaching assistants.

There is no easy measurement of the success of the class project assignment, that's comparable from one particular offering of the course to another. Averages of grades assigned would not be a good measurement; grades were subjective judgments of quality. In the first place they were often provided by the author themselves rather than some source free from bias. Instead we ask the reader to compare for themselves screenshots of the top student submissions before and after (Figures 4.10 and 1.1), hosted on the course page with permission from the students.

What is shown above in Figure 4.10 are the top ten projects from 2007 as selected by the teaching assistants at the time, representing only the best, so they are of higher quality than

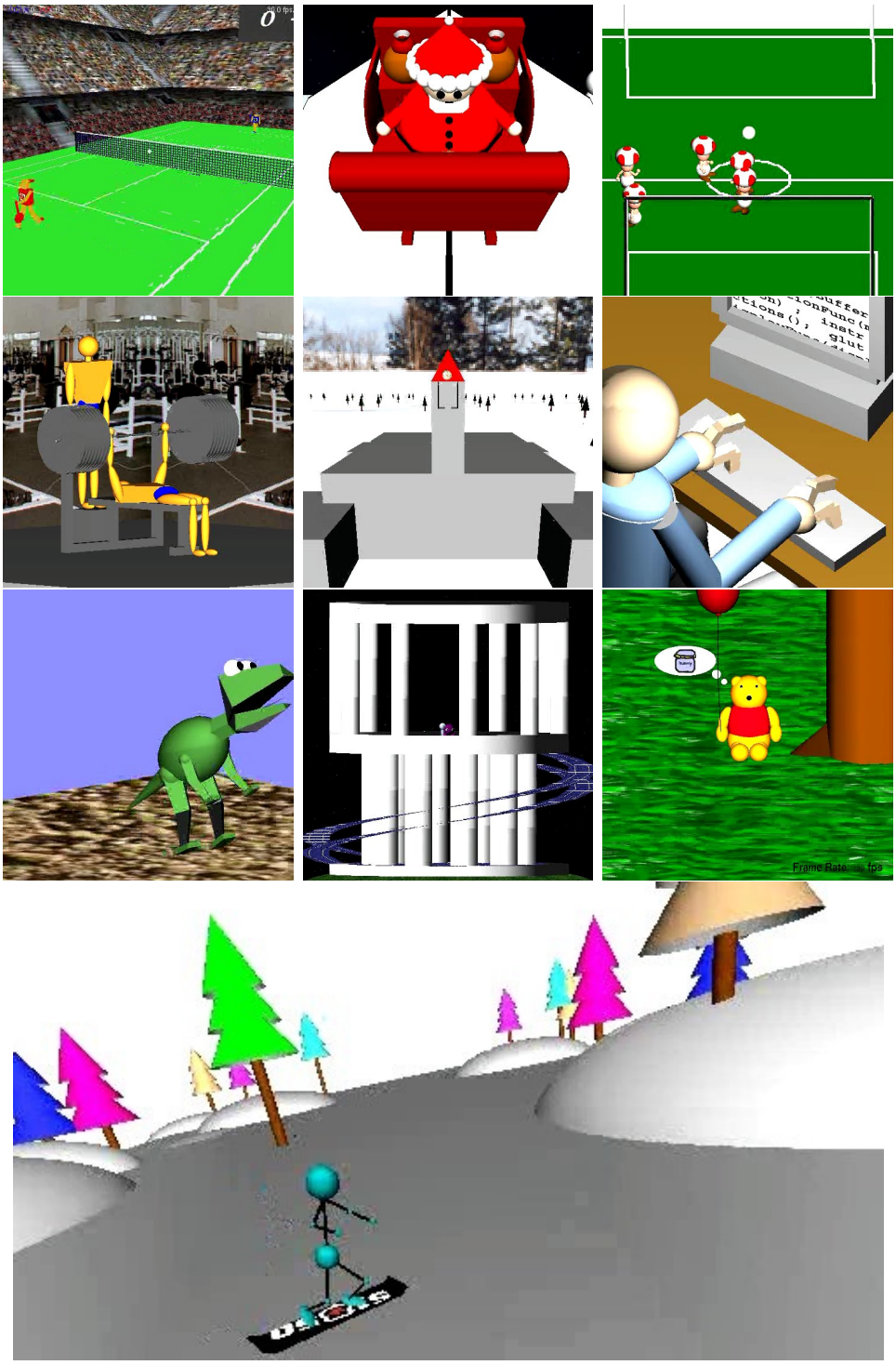


Figure 4.10: These animation projects were the top 10 selected from a 2007 offering of the UCLA course 174A: Introduction to Computer Graphics. At bottom is the classmate-elected winner. These are hosted by Professor Terzopoulos on the course page (Terzopoulos, 2017). Compare these to Figure 1.1.

an average sample would be. The 2007 batch was chosen simply based on what is available with permission. They should be representative of prior projects because the assignment, code template, and lectures remained largely unchanged for the decade prior to the creation of tiny-graphics.js.

Our newer students' work was shown near the beginning of this paper in Figure 1.1; compare it to the older ones above. Note the differences in average shape complexity, as well as the small number of shapes onscreen at once in older projects. Computers at the time could handle many more shapes, so this was more a matter of the difficulty of placing shapes and working with matrices in the given math library from Angel (Angel and Shreiner, 2014a). That process is what changed. Our library also demonstrated more examples of custom shapes. In 2007, the built-in shapes we offered for free use in projects had been limited to the few simple primitives (sphere, cylinder, cube, cone) included in the C++ library "glut" that we packaged with the project code.

Very few custom shapes are seen in the 2007 batch compared to those in Figure 1.1. On average, projects went from using fewer than one to much more than one complex custom polygon, and their complexity rose as well from nearly trivial shapes (of a dozen triangles or so) to more elaborate ones. The difference comes from the overhaul our provided code base gave to its treatment of shapes, and an increased capacity to use automation when building them.

Due to subjectivity in our project rubric, it is useful to compare how projects went *beyond* the rubric. In the stated requirements one custom shape was required, but how many students made non-trivial ones? How many included a variety of closed volumes with many triangles versus a single planar shape with a few triangles? Likewise, interactivity was not required in the projects, but what proportion of projects added it anyway, whether via keyboard, mouse, or HTML controls? These differences can be readily measured.

One such metric comes from the early 2015 run of our course. This was just prior to the creation of any parts of tiny-graphics.js or their usage. Just 2 out of 68 students in the 2015 batch decided to add interactivity to their term projects by designing their code to

handle input from the user. In contrast, the latest library version appeared in late 2017; out of our newer batch of students, 18 out of 54 (fully one third) of them independently decided to introduce interactivity to their project. To explain this eagerness we attribute the better integration between our library and HTML; the improved ease provided by each new version tiny-graphics.js when it comes to making interactive key-triggered buttons in the web interface and connecting those to the 3D scene. The presence of interactive controls also was likely helpful for students in the development process of their projects, allowing them to control which parts or moments of their scene they are looking at dynamically for ease of fault diagnosis.

One more metric that is available comes from the class's supplemental online forum (hosted by Piazza, a popular choice for university courses). This forum's statistics can indirectly help quantify how fast the development is of this educational code base. Changes to the library code are driven primarily by demand, especially whenever sources of confusion are identified during interactions with students on Piazza. The high rate of these questions and answer posts is revealing.

We will consider the 47 day period from October 3rd to November 19, 2018. At the beginning of this period, the latest version of the library was released to a new batch of UCLA students. A peak of 164 unique users per day was measured on the class Piazza during this time period. A total of 346 total message threads and 1592 total comments were made over 47 days. These typically consist of specific questions of implementation about our assignments and the open-ended term project. 422 of these comments were the author's and included long-form answers addressing the points of confusion. 342 of the message threads have been marked as viewed by the author.

Every point of confusion that is identified on Piazza leaves a record. These records are always harvested prior to the next offering of the course, resulting in a wave of code fixes. We have done this for at least seven iterations of the course so far. The library is rewritten as a new draft as each new course begins, while consulting these logs of conversations and any notes we took about them. The more records there are, the more the library evolves to smooth over rough edges and reduce all potential stumbling blocks the code itself might add

to the education experience. Our average of 33.9 online comments per day when homework is assigned is enough to provide significant and constant feedback to us. This statistic makes our educational library and project very sensitive to the needs of its student stakeholders.

4.2.1 Study Limitations

To date there have been no formal, quantitative user studies on the benefits of tiny-graphics.js or our website. For that to be possible would probably require that a University course be built centered on the library, for the purpose of gathering such information. Historically, the tiny-graphics.js library and all references to the Encyclopedia of Code have been provided by a teaching assistant, who only has so much authority to organize such an effort.

Another option is to conduct student interviews about features to measure usability. Similarly, we could include metrics into the website itself to track the usage of its features and how long people spend on coding. Our `Keyboard_Demo` Secondary Scene Component was built with this application in mind. It can log metrics of a whole open website window's keyboard usage. Analytics on our website could also be used to compare slightly alternate forms of the same article or user interface to measure differences in correct usage, or version popularity.

An analytics based approach might include setting up some formal coding environment for a study group to compare how long people spend on other online resources such as Google. The user study by Brandt et al. (2009) had these considerations and could serve as a good model for us. They tasked participants with designing a web chat room application and then monitored their foraging through online help resources while programming. They distinguished whether students were using the web to learn brand new concepts, versus refreshing their memory of specific details. Our study could make similar distinctions upon users of our website who remix existing programs that we host.

CHAPTER 5

Comparison to Related Work

We have described two projects: the tiny-graphics.js framework and the Encyclopedia of Code. They add learning resources to the current toolkit of educators, programmers, and researchers. Our project builds upon a backdrop of prior work. To show what is new that we have added to these existing efforts, we now compare our projects to the most related projects in the literature and industry.

We present a convenient chart in Table 5.1 that compares which features are present across all popular related projects, and then ours. This chart highlights the platforms that overlap our features the most. It also shows which of the features we offer are most unique, versus ones that appear more commonly in existing platforms. We found our most unique contributions to be the following:

- Our Smart Articles (from Section 3.2.1.10) seem to be unique. Recall that Smart Articles use "work delegation" by way of leaving data for remote processes that can run separate, heterogeneous programs. Recall that they use "grid computing", a distribution of concurrent processes across volunteers' machines. These are the two concepts we have broken down Smart Articles into as separate rows in our Table 5.1. Smart Articles intersect these features in a way none of the compared projects do.

- Our Active Textbooks (from Section 3.1.2.1), which also combine two rows of our Table 5.1 in a way no other listed projects do. In this case, the two rows are specific emphasis on 3D animation, and using something very close to Knuth's "Literate Programming" to deliver articles and programs that have that emphasis. These are again combined in a way no other listed projects do.

	Our Project	Wikipedia	GitHub	d.mix	Glitch	p5.js	Dwitter	ShaderToy	D3.js	CodePen	three.js	BabylonJS	TWGL.js	Maya	PlayCanvas	Code.org	@Home	npm
Runs in Web Browsers	●	●	●		●	●	●	●	●	●	●	●	●		●	●		
Offers Educational Articles to Non-Users	●	●	●		●	●						●				●		
Industry and Academic Graphics Methods	●	●	●			●	●	●	●		●	●		●	●	●		
Draws in an Audience with Games	●		●		●	●	●	●			●	●			●	●		
Dependency-Free	●	–	–		–	◐	●	◐	◐		◐	◐	●		◐			
Open Source / Documentation	●	●	●	●	–	●	●	◐	●	–	●	●	●		◐	–		
Small Source Code (1000 lines or less)	●						●			●			●					
Only A Single Code File to Read	●						●			●								
Enhance JavaScript and WebGL APIs	●								●									●
Neatly Embeddable Into Other Sites	●					●	●	●	●	●								
Live Coding Environment	●		●		●	●	●	●		●		●		●	●	●		
Collaborative Work / Remixing	●	●	●	●	●		●	●		●								
Hosting Service	●	●	●		●		●	●		●								
Crowdsourcing a Single Shared Codebase	●																	●
Grid Computing*	●																●	
Delegation to Heterogeneous Programs*	●																	●
3D Emphasis	●							●			●	●	●	●	●	●		
Literate Programming Articles	●					●												
Has Academic Publication		●	●	●					●							●	●	
Supported by an Online Community		●	●		●		●	●	●		●	●		●	●	●	●	

*Our "Smart Articles". Legend: provided (●); not provided ( ); partial/unknown (◐); not applicable (–).

Table 5.1: Our feature comparison to similar projects around the web.

- Being dependency-free, which is more ideal for education as discussed in Section 2.1. Our project has lowered the bar of entry for programming on the web, as it bypasses learning how to use frameworks, libraries, package managers, and organizing the associated files (since tiny-graphics.js fits in a single file). We instead deliver enough functionality to the coder in very small JavaScript classes using automatic server-side injection. In the end, the coder sees as little complication as possible: The contents of a few code classes.

Ours seems to be the only web-based editor with convenient links automatically embedded in each source code class that can be clicked through to all other code classes. Advanced traditional code editors like Microsoft's IDE do have this feature, but it is hidden behind a menu or key shortcut.

The Case for Teaching Computer Graphics with WebGL: A 25-Year Perspective (Angel, 2017)
Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code (Brandt et al., 2009)
Literate Programming (Knuth, 1984)
Teaching Robotics with Cloud Tools (Zubrycki and Granosik, 2017)
MOOCs in Computer Graphics (Bourdin, 2016)
Why Do Developers Use Trivial Packages? An Empirical Case Study on npm (Abdalkareem et al., 2017)
A new way of teaching programming skills to K-12 students: Code.org (Kalelioğlu, 2015)
Programming by a Sample: Rapidly Creating Web Applications with d.mix (Hartmann et al., 2007)
Students' use of Wikipedia as an academic resource—Patterns of use and perceptions of usefulness (Selwyn and Gorard, 2016)

Table 5.2: The main academic works our comparison will rely upon are found in these citations.

Lastly, not all of the projects we surveyed can host code. None besides Wikipedia use our practice of sharing a single namespace for all their crowd-sourced submissions. We do so with the goal of creating a giant single code ecosystem that can be easily verified to cover every topic (covering every name) just like an encyclopedia.

The projects we will review fall under the topics of: 1. Online encyclopedias and course materials 2. Digital Game Based Learning (DGBL) 3. Educational web demos and visualizations 4. The current ecosystem of similar graphics demo hosting and remixing websites 5. Collaborative work and software development 6. Frameworks building on WebGL, and 7. Industrial tools for graphical effects.

5.1 Wikipedia

Wikipedia may be today's largest and most recognizable reference. As of now it is the sixth most used website in the world. It commands figures of 4.6 million English articles, 287 other languages, over 18 billion page views and 500 million unique visitors each month (Selwyn and Gorard, 2016). Wikipedia is mentioned in literature across most disciplines, with descriptions of online collaborative encyclopedias often mentioning Wikipedia alone, taking it for granted as the only noteworthy one (Staub and Hodel, 2016). Wikipedia's direct predecessors were Encarta (an offline hyperlink-based encyclopedia distributed on disks) and Nupedia, both now defunct due to Wikipedia's popularity (Contributors, 2018). The same goes for Google's more recent failed competitor attempt called Knol (Anderson, 2009).

Many researchers acknowledge the impact of Wikipedia upon education (Selwyn and

Gorard, 2016) and on the literature itself, such as in the area of natural language processing (Daxenberger and Gurevych, 2013).

The Encyclopedia of Code is meant to exist alongside Wikipedia, putting a different spin on the idea of an collaborative encyclopedia with a different impact in mind. Wikipedia already does a fine job of cataloging the world's knowledge in an easy to use form. It does its job well of educating the next generation. But it has an audience of humans, and only humans. The articles do not educate the computer that loads them, or change their execution or state in any way at all that varies by article topic. It does not provide an interactive demonstration of the topic for a human viewer, or use code about the topic to augment a running computer program's capabilities.

As a reading resource Wikipedia is more of a database than a program. Its articles cannot perform topic-related actions because each article does not have any code or instructions for the visiting computer to run pertinent to its article topic. Despite the vast and complete knowledge it holds as an encyclopedia—seemingly sufficient for recreating modern civilization from scratch—Wikipedia is unable to reflect upon what it knows in each article via a running program, or perform different actions based on the article's contents. It cannot ask questions or delegate those questions to other articles. If it could do that well, collectively the whole program would be as smart as all of humanity's knowledge combined.

The primary difference between this project and Wikipedia is that Wikipedia is not alive in a certain sense. Our project asks what it would be like if Wikipedia articles could communicate with each other, while running simulation programs about their topics from within each visitor's computer. Our Smart Articles defined in Section 3.2.1.10 do just that. Inter-connected Smart Articles drop off messages for one another, which queue up work requested of specific articles. The work is farmed out to execute on the computer of the next visitor to the article, and can make use of that particular article's programmatic capabilities. When articles are visited concurrently the communication is real time.

In Section 3.2.2.6 we showed one of our articles, with a demo specialized only in geometry collision and physics, delegating work in real time to another article with a demo specialized

only in ray tracing to accomplish something it otherwise would not know how to do, as if its program had read the article like a human would. Specifically, it took advantage of a small amount of extra code supplied in a wrapper class to help it read and act on data left for it on our web server.

Our project also asks, what if this concept was taken to its logical extreme? Could Smart Articles about high-level and low-level programming concepts (graphical or otherwise) be made to communicate, and start delegating questions to one another? One example of this might be a high level article (about programming language types, or how to structure libraries, or more) that is designed to cooperate with Smart Articles about basic data structures, adding automation to some effort to implement each basic data structure in each language. How far can the automation and delegation go when making these?

This ambition could also be applied to high-level concepts besides programming, like physics, while still producing a program. Smart Articles about different physics timestepping methods could be made to communicate with articles about different physical materials or objects, allowing any combination thereof to be viewed in a simulation. Could Smart Articles communicate about topics as far reaching as Wikipedia articles cover, such as on modeling social phenomena, or on algorithms based on reasoning that come from the study of philosophy and logic? If so, programmers could even manually delegate lofty questions about these out from their code. Eventually any topic covered on Wikipedia could conceivably fall under the scope of what topics a Smart Article can tell another program about, for it to act upon.

Many research projects boil down to the novel combination of two or more things in ways that have never been tried, like mixing a little of one test tube with a little of another and finding out which pairings work. The most recent example of this that we have seen (out of many) comes from machine learning research. Ha and Schmidhuber (2018) combines one program that brainstorms virtual worlds with another that brainstorms solutions to virtual combat in 3D worlds. The two unrelated processes help each other out in a co-evolutionary way when put together; the whole is greater than the sum of its parts. Likewise, for our project to allow smart encyclopedia articles about projects to talk to one another—by being

in the same source code ecosystem—it formalizes this process and lets web visitors easily come up with new pairings and run the resulting programs.

5.2 Education in Computer Science and Math

One movement found in the literature that overlaps our goals is the search for exceptional Computer Graphics assignments (Duchowski et al., 2017). A 2017 SIGGRAPH panel issued a call for submissions in order to build a collection of such assignments, seeking to arrange them by where in the curriculum of a graphics course they would appear. This structure would resemble the current selection of graphics articles initially created for The Encyclopedia of Code, which correspond in a similar fashion to certain chapters in a standard graphics course textbook.

Refer back to Section 3.1.1 for a discussion of prior online graphics courses and their impact.

5.2.1 Literate Programming

Our sample encyclopedia articles offer something like Literate Programming, where we interleave a program's full source code with text explaining the code's intentions and design. Refer back to Section 3.1.2 for our application. Others have also attempted to use WebGL for Literate Programming. Chang et al. (2017) did so (with a focus on audio, not graphics) and used a dedicated language for Literate Programming as well as library-free (vanilla) JavaScript. Further online applications related to Literate Programming will be introduced in a moment in our discussion of p5.js.

5.2.2 Project Jupyter

Jupyter Notebooks, named after three of the programming languages they support (Julia, Python, and R), are a means of publishing a computational method which can be readily read and replicated using a web browser (Perez and Granger, 2015). They include code,

prose and results, very similarly to our Active Textbooks model. These Notebooks contain embedded panels called "cells" within the HTML document that can show code editors or graphical displays.

One especially promising feature of Notebooks is the ability to display lines of code out of order in the editor, an integral characteristic of Literate Programming as it accommodates human readers. Our "Code Widgets" panels could benefit from having similar functionality, or at least the ability for each panel to show a selected range of line numbers. Another interesting feature of Notebooks is that one type of cell can be interacted with as a command line, or can include arbitrary JavaScript for full control over the surrounding page.

While very analogous to our Active Textbooks, what we propose goes farther in a few ways. Firstly, Jupyter Notebooks are hosted by individuals (such as on GitHub) and collected manually as links. There is no central repository that allows users to generate new hosted Notebooks themselves easily for purposes of remixing, or experimentation. Secondly, Notebooks are often graphical in nature but this is usually 2D, by plotting charts and graphs. They do not generally use 3D or involve the graphics pipeline or GPU in any way, nor create 3D HTML canvas contexts (WebGL). These are thus not suitable for teaching 3D graphics or helping others develop low-level graphics programs, engines, or games.

Finally, and most importantly, their features that resemble an Active Textbook (live, interactive code editors) do not seem to be available to casual visitors. None of the Jupyter Notebooks checked were capable of live editing in the Chrome web browser without additional preparation; all were read-only code panels and the graphical plots were static as well. It seems that the Notebook "cells" can only be experimented with by power users who have pre-installed a particular Python package, whereas our Active Textbooks can benefit all web visitors, including non-programmers, instantly without installing anything.

5.2.3 Educational Web Demos and Visualizations

There are very early projects that attempted the ambitions of this work. One early exploration was by Brown and Najork (1996). Brown published animations of how various

programming algorithms work, in much the same spirit of the Encyclopedia of Code. Their "electronic textbook" consists of a set of web pages (analogous to our active textbooks). Many of the same ideas are found, such as control panels and multiple simultaneous views of the running program as it presents an algorithm to the user.

Brown's work predates Windows 95 and even the emergence of support for JavaScript in websites. Without the ability to have their page run any sort of code, Brown envisioned "active objects", regions of the page that are drawn by programs dynamically loaded through the Web—but static once rendered to the client, along with any other media files they initially loaded onto the page. Little did they know that years later JavaScript would revolutionize web pages into dynamically interactive documents, much less with WebGL and modern shader-assisted graphics. Their project existed in a backdrop of the Windows 3.1 era (although they used the UNIX-like X Windows OS); in a lot of ways our project is a more modern take on it now that better technology is available.

More recently, others have made educational web tools that visualize data using WebGL. Rego and Koes (2014) used WebGL to visualize large molecules and their data. It includes a sharing system and a code library for developers. Sherif (2015) sought to make it easier to view huge distributed data sets about brain imaging. Their work saves the viewer from having to install any software or transfer all the data onto one machine. This allows scientists to build guided hypotheses "after analyzing the mass of available data". The web interface allows them to publish and share data with one another.

Heer and Bostock designed a software library for online visualizations and present a series of informative demos about how to use them, in much the same spirit as our project. Visualizations include Voronoi Diagrams, Chord Diagrams, Node-Link Trees, Sunbursts, Geographic data, and Circle packing graphs. These make use of Bostock's popular D3.js library for the interactive visualization of math and data. On the academic side they experimentally assess the design of their visualizations using Mechanical Turk, a form of crowd-sourced survey (Rosenberg, 2015).

5.3 Collaborative Software Hosting and the Body of Open Source Software

This project benefits the open source community by providing an alternative collaboration platform. The open source movement spreads the notion of collaboration by creating code that is packaged with permissive licenses. The related free software movement rejects the idea of proprietary software altogether and maintains the vision that all software is a part of freedom of speech (Warger, 2009).

Open source software is often collected in repositories and automatically harvested as needed by software package managers. These have search-able indexed databases of entire libraries, preserving the entire file structure and various versions of each. The leading JavaScript package manager is npm (Wittern et al., 2016). Package managers can read metadata about library dependencies and help programs to load the correct code from disparate sources. The end program has effectively delegated to outside code on the web.

The package manager "reads" metadata of heterogeneous programs and uses the right specialized one at the right time. In a way our Smart Articles exist in the same spirit as these. Smart Articles are also readable by computers for delegation of specialized tasks. They manage not just code files but heterogeneous *running processes* over the web, thereby employing grid computing (distribution of work over distant machines). Smart Articles are even more computer-friendly than the packages managed by the likes of npm as they can not just use metadata to give problems new capabilities but also share the load of other programs when needed.

The Free and Open Source movements have an effect of collectivizing code for the good of society; it "demonstrates how labour can self-organize production, and, as is shown by the free operating system GNU/Linux, even compete with some of the worlds largest firms" (Söderberg, 2015).

5.3.1 GitHub

GitHub is the largest host of source code in the world (Gousios et al., 2014). Based on the Unix program git, it manages versions of user-submitted software and other documents. It is the most popular repository of open source code next to its competitors SourceForge, Google Code, Bitbucket and Gitlab (Finley, 2011). Like the Encyclopedia of Code, GitHub is capable of running certain programs submitted to it directly by visiting certain URLs.

The organization of our own code repository under the encyclopedia model makes it a very different place from GitHub. Our encyclopedia only allows one article per topic. This means the redundancy found on GitHub, multiple code projects accomplishing the same task in different ways, is not present. As a tradeoff, this instead gives us a single body of code with an unusually full namespace, with the aim of full coverage of topics and a complete set of tools for the programmer. Users would easily be able to tell if a topic is already covered or if a tool is already available by simply typing in its name and checking for a matching class in our codebase.

Not only does our codebase lack any duplicates, but it automatically presents itself to the programmer in a minimal form—a process not built into GitHub. Dependency injection from our server is used to present visitors with only small parts of our giant combined codebase at once. They see the parts that the article they are viewing immediately needs, and no others. The source code stays minimal and readable, and wasteful amounts of data are not transmitted. Users of the Encyclopedia of Code can code as though they could call upon a massive, fully featured library of all the demos ever made, or if they want, as though they had imported tiny-graphics.js alone if that is all their code uses. By having this advantage over GitHub it fulfills a different niche, encouraging the design of one giant all-purpose codebase that explores every educational topic.

Lastly, GitHub programs also have no "Smart Article" analogue; they have no built-in way to leave jobs for one another to execute, when they are run the next time (or in real time if run concurrently). GitHub as a platform does not have a back-end like ours that naturally promotes Grid Computing, the crowdsourced volunteering of CPU time.

5.4 Similar Efforts Around the Web

This section explores projects around the web that are relevant to this work. These all use collaborative coding done as small "re-mixable" widgets and demos rather than large projects.

We do not directly build upon any of these websites, in the sense that our tiny-graphics.js wholly improved upon Angel's codebase (see Section 2.2.2) and other current tiny WebGL libraries. Rather the Encyclopedia of Code is meant to serve alongside these other tools as they each teach programming using their own strengths.

5.4.1 d.mix

The d.mix project (Hartmann et al., 2007) was an academic attempt to "democratize" application development via collaborative coding. Like our project, it was an earlier exploration of what happens when web visitors of any skill level can host pages and remix each others' pages into novel creations.

The idea of d.mix was to allow novice programmers not familiar with particular web programming APIs to nonetheless build pages that use them to show live content from major sites around the web. The researchers experimentally tested a system where users build "mashups" by scrolling through and choosing from a selection of interactive elements, obtainable from search results on popular websites as of 2007 (YouTube, Google Maps, Yahoo Search, Flickr, Amazon and the Java Developers Almanac). Users of d.mix could select associated code that generates the particular elements or operations the programmer wants to combine into their own website. The results could be shared and remixed by others. Unlike our web tool, d.mix must be installed to be used.

5.4.2 Glitch

Glitch (Software, 2000) is a code remixing website with very similar educational goals to this project. The makers of Glitch designed it as a resource for web programming, and envision

their users learning through thousands of crowd-sourced educational tutorials. These are tiny user-submitted web applications divided up by topic that each teach visitors how to accomplish some web development functionality (sometimes directly providing the tool for it).

Glitch is made by the distinguished creators of stackoverflow.com, a crowd-sourced answer website that itself has changed the practice of programming worldwide. It is a reference source for most possible programming questions, and the top result on Google for most of them. While stackoverflow crowdsourced its conversations and programming answers, Glitch instead crowdsources deeper tutorials and demos.

First and foremost, Glitch is a free host of websites, of which our Encyclopedia of Code is one. To understand Glitch, first consider pastebin.com, a popular online host of text snippets, known for immediately giving visitors their own permanent URL for any text they have pasted in to its interface. Glitch is similar to that, although visitors to Glitch do not enter just any text. They enter the JavaScript code for a web server front and back end. Whatever they entered is saved at a permanent URL, and then the code executes forever on both Glitch's machine (the back end) and the website's visitors' machines (the front end). A working website results, and the back end allows the website to have memory, so that website content can reflect prior visits. Projects running on Glitch back end sleep and wake up when requested to keep the service moving fast enough.

Any website made with Glitch can alternatively be viewed with an advanced source code editor that is provided. Other Glitch users can see it and, with the touch of a single button, remix the page in minor or major ways, basing their own website on it.

Our project is more than a smaller Glitch hosted on Glitch; we distinguish it from its host in a few ways. Our tutorials are (currently) more focused in scope, on Computer Graphics education, and use extra technology (WebGL calls) to do so. This means we can show 3D animations and explore graphics, unlike most Glitch pages. Our description during the comparison to GitHub of how our encyclopedia organization enables code projects to be more tightly integrated applies here to Glitch too. Besides a code editing interface like Glitch

provides, we also provide a separate code *reading* interface complete with hyperlinks for navigating related classes in a large codebase.

5.4.3 p5.js

The p5.js website (McCarthy, 2013) consists of tiny self-contained programs with similar structure to the Encyclopedia of Code. Many sample demos that use the p5.js JavaScript library are shown. Each graphically illuminates a particular educational topic. To name a few, there are mathematics examples, L-Systems, flock simulation, interaction, and the practical use of sound, out of many dozens of others. These are interactive animations like ours with live code editors beneath them. Anyone using the library can embed editable code widgets into their web pages using the animation capabilities of p5.js. Visitors get live visual results right above where they edit in these small code panels.

Multiple researchers have mentioned the potential that exists in p5.js's interface for Literate Programming, using the small code editors and visual results pane it adds to websites; Zubrycki and Granosik (2017), in similar spirit to this project, use p5.js for Literate Programming and call it as such. They embed p5.js's runnable, editable animation code widgets on their website in between their written materials for robotics students. One such demo is for numerical inverse kinematics. Code and an animation of the mechanical linkages is displayed right in between the involved steps of math formulas.

In each p5.js-based post written by Zubrycki and Granosik (2017), they embed multiple versions of their program, mirroring our Active Textbooks concept, where we suggest illustrating programming concepts that build upon each other by showing many 3D canvases containing incremental intermediate programs.

Zubrycki and Granosik (2017) experimentally found that cloud tools such as p5.js create a better experience for students compared to similar robotics lessons they encounter with MATLAB, another common tool in engineering classes. As a proprietary system, students only have temporary access to MATLAB while in school.

Compared to our website, the p5.js page lacks a way for visitors to host their remixed

code or show others, or any automatic interface to submit new educational articles to their example list. Above all, most demos offered for p5.js are only in two dimensions; the 3D capabilities of their library are not emphasized over the 2D graphics "contexts" that HTML canvases can display.

5.4.4 Dwitter

At `https://dwitter.com` (Selvik, 2016), visitors type into a code editor and show off their best graphics demo in 140 characters of plain JavaScript. This is yet another website that allows quickly contributing crowd-sourced tiny JavaScript applications that are graphics related. Here again the emphasis is on 2D HTML graphics contexts rather than 3D, and upon code golf (the art of designing short code) rather than on education.

This process of remixing toy programs on Dwitter can itself be educational, however. Performing a small edit and then re-running a demo can be useful even for gaining a first impression of how the submission's program works. Users unfamiliar with a piece of code, or even coding at all, can still try performing arbitrary changes to the math inside the code. They can then learn by observation, noting changes to the graphical output. This especially works on Dwitter due to the enforced 140-character code length of all demos. Even if this educational purpose is by accident, Dwitter shares it and other structural similarities with our website.

5.4.5 ShaderToy

At a 2016 ACM SIGGRAPH conference talk delivered by Edward Angel, we heard of `https://shadertoy.com` being described as the current greatest website on the internet. This should be understandable to anyone who visits and observes the thousands of impressive crowd-sourced visual effects found under their "browse" page. ShaderToy is one of the only other websites out there to actually host WebGL for visitors and display it for others, like we do in this project. The creators of ShaderToy have organized courses about it for the SIGGRAPH conference (Jeremias and Quilez, 2014).

ShaderToy is an online code sharing and hosting platform for WebGL. There is emphasis on shaders, specifically the fragment shader. Their users' submissions explore the wide aesthetic range of possible fragment effects that graphics cards are capable of. The vertex geometry submitted to the card is always a single quadrilateral, and then ray tracing is used by the fragment shader to draw the real scene. Audio output of graphics cards is also explored.

ShaderToy's focus on custom shaders allows the demos to produce breathtaking original visual effects, and there are thousands of them available to browse and instantly activate. ShaderToy provides a single static JavaScript program for all projects, and the users write a fragment shader for it; recall that this is its own independent program separate from the JavaScript. This fact affords opportunities to extract shaders from shadertoy.com and insert them into piecemeal into any other WebGL program. One of our own students successfully did so. They created a script that embeds shaders hosted on `https://shadertoy.com` into their game made using our tiny-graphics.js framework. They used them to color in or texture their polygons with unique effects such as fire and water. In this way, our web demos and animations can co-exist with the ShaderToy website. They are more specialized than us in the niche domain of fragment shaders, and they maintain a good repository of shaders that we can import from.

ShaderToy has a unique website navigator that takes advantage of WebGL. They embed several WebGL canvases together on the page to browse submitted programs. In their navigator the programs run one at a time when you hover the mouse over them, but otherwise are all pre-loaded onto the same page and can thus run instantly upon user interaction. Clicking allows the submission's code to be inspected. The user is then greeted by a built-in code editor with syntax highlighting, that is much like the interface on our website. Their editor pages are capable of both running visitor's programs and saving them for permanent hosting. ShaderToy already has an existing community by providing forums, comment sections, and likes.

ShaderToy provides a single static JavaScript program that does not give users control over their own web page layout (including interactivity) or their own CPU-side program like

our tiny-graphics.js does. The ShaderToy website does not put forward official educational articles on graphics topics, much less collect tutorials of a general encyclopedic nature. Unlike in our Encyclopedia of Code, ShaderToy's visitors can generally only interact with one submission's capabilities at a time, and must combine them by using the remix feature on them individually.

5.5 Frameworks Building on WebGL

Our project offers a solution for using WebGL, especially for graphics education. In Section 2.2 we compared our solution in detail to another WebGL framework that is purpose-built for graphics education, that of Angel provided with his Computer Graphics textbook (Angel and Shreiner, 2014a) and SIGGRAPH course (Angel and Shreiner, 2016; Angel and Haines, 2017; Angel, 2017). Ours is far from the only WebGL utility library out there. A popular game engine, PlayCanvas, is based on WebGL and is discussed at the end of this chapter. But foremost among WebGL utilities is the "three.js" library.

5.5.1 three.js

Three.js (Cabello, 2018) is another frequently used option for teaching today's graphics courses. One example of an online course that uses Three.js on Udacity is (Haines, 2014). The three.js framework has more "favorites" on GitHub than any other WebGL project by nearly two orders of magnitude (Cabello, 2018); it is an extremely popular base to build WebGL programs on top of.

One of Angel's design goals for his course was to avoid layers of separation between students and any details about implementation and architecture. Hence, he ruled out three.js, which has hundreds of source files and an interface adding its own learning curve to basic graphics concepts. As Angel described in (Angel, 2017), three.js fills a different niche than what is desired for an engineering course where students are mainly concerned with architecture and low level operations.

Whether it belongs in the classroom or not, three.js is impressive to behold. Its makers' website features a long list of links to beautiful web demos to showcase their useful WebGL utilities. While this is more flashy than the bare bones framework and demos provided by Angel, it does not surpass what the Encyclopedia of Code could eventually do with the sum of hundreds of articles. Unlike three.js, each of our demos depends on only a few files and is packaged as a minimal executable. Ours are thus easier to fully demystify and to master.

5.5.2 BabylonJS

The other big name in WebGL frameworks is BabylonJS (BabylonJS, 2013), an adaptation of Microsoft's Silverlight 3D system. The babylon.js library, like three.js, wraps WebGL. Also like three.js, the website for it provides a number of working examples organized by topic. In Babylon's case the examples (BabylonJS, 2014) each have supporting documentation that is far more learner friendly than three.js; BabylonJS intersperses demos with educational tutorials that include photos of the demo for each step. They include code editor widgets with syntax highlighting like ours. Out of their many tutorial lessons there is a heavy emphasis on ones about game and physics topics like collisions rather than just basic animation and general graphics card capabilities.

Many of the same things already said of the p5.js examples page (McCarthy, 2013) apply to BabylonJS except this time the emphasis is on 3D and the same WebGL technologies we use are present. Many parallels to our own project can be drawn, so it is worth pointing out the clear differences that put our project into a different niche. BabylonJS already dominates its own niche, with a large community of developers maintaining it, and having powerful backing from Microsoft from day one.

One major difference is that BabylonJS is not a collaborative platform; it does not host user-submitted demos or provide an interface for them to be sampled or remixed. BabylonJS creations are typically self-hosted and posted on the html5gamedevs.com forum (html5gamedevs, 2013). Our platform provides its own instant hosting. Code submissions go in our repository where they can work together in various ways. We envision our encyclopedia

codebase as one giant project instead of many independent projects. Progressively more phenomena could be modelled by our system in this way to create one comprehensive virtual world, distinguishing it from a community like Babylon's where each member makes their own game-engine powered app.

One individual contributor made a billiards game project showcased on BabylonJS's main examples page that plays the same way as our `Visual_Billiards` demo, right down to similar controls and colorful ray visualizations of which balls are in the cue ball's bounce trajectory. Ours has the advantage of being open source and available to learners. It is also unclear how large the source code is—their obfuscated code is around 100,000 lines. Something of that scale could never neatly fit in with a couple of text pages in an educational tutorial. Their purpose in making this billiards game was different from ours, which was to educate and prove the grid computing and delegation capabilities of the Smart Article. Rather than only one running program we have multiple on several browsers helping each other out.

BabylonJS generally cannot implement our concept of Smart Articles. In these, educational demos hosted on the same place delegate work to one another. They serve to teach each other, not just the students who are reading the demo page, how to perform an algorithm or do some simulation task. Our project's hosting also affords it the opportunity for grid computing, using the visitor's computing resources to make the demos go faster; this falls outside of Babylon's scope.

Babylon's excellent tutorials do not fall under our definition of Active Textbooks, since they contain static photos instead of embedded demos / code editor widgets. We have that advantage over Babylon, although they easily could have achieved Active Textbook articles; it is possible to embed Babylon's demos and editors in any website using the CodePen (CodePen, 2012) JavaScript sharing website, which comes with a package manager compatible with BabylonJS. In a similar vein, three.js is on there, as well as explicitly built in to a different popular JavaScript sharing website, JSFiddle (JSFiddle, 2010).

Our project's organization is different from BabylonJS. This is influenced by the way

the Encyclopedia of Code began and its intended audience of ambitious students trying to put their project over the top with advanced topics for extra credit. As a general-audience encyclopedia, our list of allowed topics for articles potentially has a much wider range than Babylon's list of examples. Lastly, our library does all this in a single file. Even for libraries of the same small size as ours, ours saves students the step of file management. The file is human readable and very compact at the same time, to keep the focus on higher level code organization, and make it possible to visualize whole code structures at once.

5.5.2.1 Academic Ties

Our project's academic ties provide the main distinction from Babylon, considering Babylon's otherwise very similar web demo arrangement and their extensive graphics tutorials. We, however, have direct comparability to an existing textbook, due to our association with Angel's library. Our project is an important bridge between community projects like BabylonJS and University projects like the Angel book.

We may have demonstrated an improvement compared to an academic library, but that is actually not hard to do; engines like BabylonJS exist that already easily outdo Angel's WebGL framework. Code used for teaching is never necessarily at the cutting edge. Due to institutional momentum SIGGRAPH courses like Angel's on WebGL (Angel and Haines, 2017) should only be expected to contain techniques attributed to renowned academics, not tools from just anyone who has made a breakthrough. Our project likewise may never be famous enough to appear in textbooks, but at least it mirrors the structure of current textbooks in a way that can better serve academic courses in graphics.

Libraries like Angel's and ours are also structurally *easier* to integrate with a university course. Specifically, they are small enough to fit in the pages of a textbook in full to show how to organize WebGL code. A BabylonJS or three.js based book that explains graphics topics in traditional textbook chapters would only have enough space left to superficially cover its external API. There would not be time to cover the underlying WebGL implementation of the framework, because these frameworks are huge. Comparatively, tiny projects like ours

and Angel's can readily have their workings fully laid bare in the small space of a tutorial, so the student feels like they are learning about WebGL and graphics in general, not Babylon specifically, and therefore are not wasting their time by learning a single framework out of many.

5.5.3 More Modular than Other Frameworks

The modularity of our website allows it to teach more with less clutter compared to most alternatives. Our core module, the tiny-graphics.js file, is orders of magnitude smaller than three.js or BabylonJS (even fewer lines than Angel's library), addressing Angel's concerns about how three.js's massive interface might abstract away architecture details from students—in our library, implementation and WebGL calls are always just one thin layer away.

Without relying on external code, our small single-file implementation nonetheless presents several of the same features present in three.js: Data loaders, math utilities, cameras, input management, texture images, and a framework for organizing geometry, shader, and material code.

In a small, thin layer our library manages to include all these features. It leaves out other things that three.js or BabylonJS can do, but we do not have to leave them out entirely. This is where our web resource's modular organization as an encyclopedia can shine. Any pieces we leave out of the core tiny-graphics.js can still be added back on later as self-contained demo articles covering one topic and nothing else. These can go well beyond that of setting up WebGL. Our code's organization into independent tutorials can gain back the major features we are missing from three.js such as sorting and frustum culling.

Any other common graphics capability found in three.js, such as its scene graph tools, can be hosted on our website isolated away from other code at a particular URL. Scene graphs can be included optionally from our web server by simply visiting the article called `Scene_Graph_Tool` (from Section 3.2.3.18), or by a user mentioning the class `Scene_Graph_Tool` in source code. When that topic's particular source code is not needed,

it is not imported and is not cluttering up the source code the visitor sees, or the main codebase, because all articles contain minimal programs for showing the given demo without anything else distracting from the education. The same goes for code about collision detection, rendering text, special shapes, and ray tracing. None of the source code for these concepts is included unless the user is visiting an article about them; it is invisible whenever it is not needed.

5.5.4 Other Tiny Graphics Libraries Around the Web

Some JavaScript libraries exist for making 3D graphics on the web easier to work with, and try to do nothing more than wrap WebGL commands in helper functions or classes, just as our tiny-graphics.js utility does. Even some of these alternative wrappers also have "tiny" in the name, such as TWGL (Tiny WebGL) (greggman, 2015), adapted from the same author's TDL library. According to its author, the TWGL library's "sole purpose" is to make using the WebGL API less verbose: "TWGL is NOT trying to help with the complexity of managing shaders and writing GLSL. Nor is it a 3D library like three.js."

TWGL successfully organizes WebGL commands and simplifies the WebGL API, but stops there. Our library of course goes further, providing integration with all the complex examples on exhibit at the Encyclopedia of Code website. TWGL has a more limited scope than our utility, despite the same or bigger size in code; this is due to their more thorough coverage of WebGL's API and specification to ensure a complete wrapper.

In summary, the main issue with Tiny WebGL (TWGL) for the purposes of a graphics course is that it is not tiny enough. TWGL is for users who do not want to touch WebGL directly or make any of its API calls themselves. But the comprehensiveness of TWGL in wrapping every single possible usage of the WebGL API does not make for easy reading. Nor does its source code serve an educational purpose like our tiny-graphics.js, detailing what a student minimally needs to do to get WebGL running in a well-organized way. In our framework, only two JavaScript classes (`Vertex_Buffer` and `Shader`) do nearly all of the heavy lifting of calling WebGL, and are each about a page long, which is much more

digestible for a beginner. Our library's usage of only the details from the WebGL spec that it needs is more optimal for accompanying a course or textbook; our single file organization means it can feasibly be printed out in full.

5.5.4.1 Relation to the Encyclopedia Of Code

TWGL is not without educational ambitions. It is associated with a website at `https://webgl2fundamentals.org` by the same author, hosting a series of tutorials and WebGL advice articles. The GitHub development page for their website (greggman, 2018) details plans that bear a striking resemblance to our project. Their article topics wishlist overlaps our own list of articles that are either complete or currently under construction for the Encyclopedia of Code. Topics include scene graphs, skinning, frustum culling, grid or octree culling, integration with `https://shadertoy.com` submissions as mentioned previously, text rendering, and spot lighting.

5.6 Industrial Tools for Graphical Effects

Computer Graphics research supports multi-billion dollar industries such as film, websites, gaming, engineering, scientific computing, and data visualization. For the film industry, best-selling software tools that have emerged for graphics include Maya, Houdini, and 3ds Max (Govil-Pai, 2006; Labschütz et al., 2011). As with three.js, working with pre-built tools is usually a trade-off of flexibility for convenience. The user gains a way to re-use the visually appealing effects the software's designers happened to include based on customer demand. But for this the user often sacrifices flexibility to pave new ground or take a radically different approach than the software makers envisioned. While this drawback applies to high-level 3D modelling tools in general, software suites like Maya are actually mature enough to avoid it somewhat. Maya can integrate desired new methods from the research literature directly via plugins whenever the built-in features are not enough.

Our project, while adjacent to this industry, does not fill the same niche since its feature set (the sum total of its current educational demo articles) is so small. But our

`Scene_Graph_Tool` subsection in particular is relevant, since it provides a visualization of how scene graphs work, which are at the heart of modeling software. Our website can therefore at least build upon industrial tools educationally, by serving as a valuable training tool for those wishing to learn how they work.

Our `Scene_Graph_Tool`, described in Section 3.2.3.18, performs and visualizes edits to a scene graph. It constructs elaborate 3D shapes showing a high degree of symmetry. With a single keystroke our tool can create self-similar designs, by creating child graph nodes that continue the transform pattern of their close ancestors. The same graph operation requires a sequence of several UI window interactions to do on Maya or on experimental tools that seem to support it such as in Jacobs et al. (2017). Giving the user the option to see their self-similar designs emerge at interactive speeds does not seem to have been done before.

5.7 Digital Game-Based Learning

The Encyclopedia of Code includes educational games. Our `Bases_Game`, detailed in Section 3.2.2.1, is an interactive page and 3D demo that teaches a general audience concepts from matrix algebra, especially order of multiplication.

Perhaps the biggest effort toward Digital Game Based Learning (DGBL) today is Code.org (Kalelioğlu, 2015), a nonprofit that is dedicated to expanding access to Computer Science. Their Hour of Code campaign (Wilson, 2014) uses Computer Science games in the classroom to engage tens of millions of the world's K-12 students, and has also successfully lobbied for changes to policies and curricula.

The Code.org and Hour of Code websites showcase a search engine where hundreds of their educational programming games are visible in one place. Each is displayed with their recommended grade level and topic coverage. They include contributions from partners such as Disney, Microsoft, and MIT, and feature popular kids games like Crossy Road, Lego, and Minecraft. Their website is important to our project because it is so similar as a collection of appealing interactive educational programming demos, as the Encyclopedia of Code aspires to be.

Their collection includes a handful of games that are even built from the same technology we use, WebGL. Disney's *Moana*-themed game (Disney, 2015) uses WebGL contexts to draw its animations, to teach basic coding with the goal of character navigation in a 3D world.

The Encyclopedia of Code aspires to become another recognized host and list of educational games. But how worthwhile is this? Some studies have questioned the efficacy of game based learning. All et al. (2014) observed that engaging in these educational games is mostly the result of external coercion. They found "a lack of sound empirical evidence on the effectiveness" of DBGL, and noted papers with incomplete information on their study design, impeding replication of results or any opportunity to disprove them.

Still, individual positive results can be found across the literature. Many other projects in the literature describe themselves under the category of Digital Game-Based Learning (DGBL). Yannier (2016) created EarthShake, a mixed reality game for helping children learn physics. Using a Kinect, children can observe stacks of objects colliding in physics simulations. This is paired with real-world demonstrations of the same. Their experiments revealed a five-fold improvement in learning and enjoyment due to the real-world component. Hearns (2014) created a literacy game for college students in the virtual world of Second Life. Their measurements of the platform showed it even successfully got students to enjoy the dull selected academic topic, sentence construction.

In the author's experience, programming-related games appeared in their early education with HyperStudio (O'Keefe, 1989), a software suite that could design special interactive animations or games. These multimedia creations went through state changes triggered by user interactions, acting as though the viewer was following a network of hyperlinks. The young programmer was technically building their first Finite State Machine. Experiences like this promote an interest in programming concepts by creating art with computers at an early age. Such digital games and modelling tools are a valuable way to motivate children of any generation towards the Computer Science field.

5.7.1 WebGL and Game Engines

Disney's *Moana* game above is also noteworthy for using PlayCanvas, a WebGL-based game engine also used by Leapfrog and other major players in digital education (PlayCanvas, 2014). PlayCanvas shows what our WebGL-based Encyclopedia of Code could evolve into if we successfully integrate more support for games in our codebase and host more crowd-sourced games. In keeping with our website's purpose we would additionally organize all submissions that are games by educational topic, if any. PlayCanvas also shows us the current extremes of what WebGL can do to support games. A collaboration between PlayCanvas and Mozilla resulted in a particularly stunning tech demo called "After the Flood" (PlayCanvas and Mozilla, 2017) that presents the cutting edge features of WebGL 2.0 (Gilbert and Albeza, 2017).

Other game and modeling engines may not involve programming in WebGL, but nonetheless can export the finished code to WebGL and still be run inside of web pages. Unreal and Unity both can export to WebGL (Hu et al., 2017) using Emscripten, a cross compiler that translates C++ programs into a subset of JavaScript that is faster than normal and comparable in speed to the original native code. Emscripten produces code that leaves out JavaScript language features that would trigger expensive automatic memory management. This usage of it by Unity and Unreal is another way that the games industry is currently reaching out towards WebGL.

In summary, WebGL offers substantial potential for educational games by bringing 3D graphics to the web, and there are as many opportunities to engage students with these as there are topics in the Computer Science curriculum.

By hosting a collaborative, crowdsourced coding environment, the Encyclopedia of Code can hopefully encourage the creation of more digital games, as well as encourage students to make games of their own. Our project provides an easy way for them to remix existing games we host, even for minor exploratory edits the student might want to try.

CHAPTER 6

Conclusion

6.1 Summary

Chapters 2 and 3 of this dissertation presented the two major components into which our work is naturally organized—an education-friendly WebGL code library called tiny-graphics.js, and a collaborative online repository of 3D graphics demos and articles called the Encyclopedia of Code. In the course of exploring these two components, we developed several other contributions that were listed in Chapter 1.

In Chapter 4, we presented results using our tiny-graphics.js library, by using it to rethink and reinvent Angel's supplementary textbook demos, comparing our most similar counterparts side by side. We also described a real-world case study and our observations here at UCLA, where our library has been employed in the graphics courses multiple times, with very favorable outcomes compared to prior student projects created in the absence of our library.

In Chapter 5, we speculated about the future of this project and compared its trajectory to Wikipedia's. We also provided a table comparing our website's intersection of features to other projects around the web, and a more detailed exploration of each.

The Encyclopedia of Code remains an ongoing effort. We aim to improve its features, to foster a community, add tutorials and documentation, encourage others to program in WebGL, and crowd-source JavaScript code from other programmers in order to generate new educational articles and games.

6.2 Future Work

Our current priorities for future work are as follows:

- The expansion of the Active Textbook articles, to drive home the academic potential of replacing illustrations with interactive WebGL demos.
- The addition of new Smart Articles that show their power when it comes to distributed applications (grid computing) such as solving NP-complete tasks, or ray tracing a scene faster as more users log on, thereby contributing more processing power.
- For better graphics and physics capabilities, the development of a complex mesh data structure library using our array-based half-edge meshes from Section 3.2.3.19. In this effort, we plan to support the following features by pursuing as yet only partially explored avenues:
  - Automatic generation of our meshes from points using classic Delaunay triangulation.
  - Moving to 3D with the Half-Face Data Structure (Alumbaugh and Jiao, 2005) for storing volumetric meshes as tetrahedrons.
  - The Deformable Simplicial Complex (DSC) technique (Misztal, 2010) for tracking surfaces as boundaries embedded within a larger tetrahedralized domain.
  - Ray tracers and collision detection algorithms that work inside of DSC meshes to avoid all costly search queries through the scene.
  - Simulations of rigid bodies, elasticity, fluid flow, and other physical phenomena that work within a DSC domain (and re-mesh parts of it adaptively) to manage virtual worlds.

We believe these enhanced virtual worlds would vastly improve the animation capabilities of our framework. They may also provide more useful platforms for exploring the questions of what our Smart Articles concept is capable of—their multi-processing capabilities allow

them to delegate and offload specialized work to outside CPUs, which would be very helpful for hosting massive online virtual worlds. These avenues could lead to a variety of subjects for further study.

We maintain a long list of planned improvements to our library and online tool in an internal issue tracker. We also have nearly 70 articles on deck to be made (starting from descriptions), or already made but currently under further construction.

We call on all our readers to explore our website, familiarize themselves with the JavaScript code classes using the navigator, and experiment with using the editing interface to make small code changes and save them to generate permanent URLs to share with others. We encourage all users to contribute any code classes of their own that are interdependent with the existing classes in our library and database, and to submit them anonymously or under their name for proper attribution. You can help create a growing repository of useful code along with beautiful, educational graphics demos.

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? An empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 385–395. ACM. 12, 105

Abraham, B. V. (2012). Handling keyboard shortcuts in JavaScript. http://www.openjs.com/scripts/events/keyboard_shortcuts/. Accessed: 2018-07-23. 22, 78

Ahn, S. H. (2009). OpenGL projection matrix. http://www.songho.ca/opengl/gl_projectionmatrix.html. Accessed: 2018-09-04. 49

All, A., Castellar, E. P. N., and Van Looy, J. (2014). Measuring effectiveness in digital game-based learning: A methodological review. *International Journal of Serious Games*, 1(2). 126

Alumbaugh, T. J. and Jiao, X. (2005). Compact array-based mesh data structures. In *Proceedings of the 14th International Meshing Roundtable*, pages 485–503. Springer. 86, 129

Anderson, N. (2009). Google Knol six months later: Wikipedia need not worry. https://arstechnica.com/news.ars/post/20090119-google-knol-six-months-later-wikipedia-need-not-worry.html. Accessed: 2009-01-21. 105

Angel, E. (2017). The case for teaching computer graphics with WebGL: A 25-year perspective. *IEEE Computer Graphics and Applications*, 37(2):106–112. 2, 9, 11, 14, 16, 29, 105, 118

Angel, E. and Haines, E. (2017). An interactive introduction to WebGL and three.js. In *ACM SIGGRAPH 2017 Courses*, page 17. ACM. 2, 9, 118, 121

Angel, E. and Shreiner, D. (2014a). *Interactive Computer Graphics with WebGL*. Addison-Wesley Professional. 2, 6, 8, 9, 12, 50, 79, 87, 88, 89, 91, 92, 93, 94, 95, 96, 97, 100, 118

Angel, E. and Shreiner, D. (2014b). Web demo: figure. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/09/figure.html. 97

Angel, E. and Shreiner, D. (2014c). Web demo: gasket1. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/02/gasket1.html. 88

Angel, E. and Shreiner, D. (2014d). Web demo: hat. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/05/hat.html. 93

Angel, E. and Shreiner, D. (2014e). Web demo: particlesystem. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/10/particleSystem.html. 94

Angel, E. and Shreiner, D. (2014f). Web demo: perspective1. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/05/perspective1.html. 91

Angel, E. and Shreiner, D. (2014g). Web demo: reflectionmap2. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/07/reflectionMap2.html. 96

Angel, E. and Shreiner, D. (2014h). Web demo: robotarm. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/09/robotArm.html. 97

Angel, E. and Shreiner, D. (2014i). Web demo: shadedsphere1. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/06/shadedSphere1.html. 92

Angel, E. and Shreiner, D. (2014j). Web demo: shadedsphere2. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/06/shadedSphere2.html. 92

Angel, E. and Shreiner, D. (2014k). Web demo: shadedsphere3. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/06/shadedSphere3.html. 92

Angel, E. and Shreiner, D. (2014l). Web demo: teapot5. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/11/teapot5.html. 95

Angel, E. and Shreiner, D. (2014m). Web demo: trackball. http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SEVENTH_EDITION/CODE/04/trackball.html. 89

Angel, E. and Shreiner, D. (2016). An introduction to graphics programming using WebGL. In *ACM SIGGRAPH 2016 Courses*, page 5. ACM. 2, 9, 118

BabylonJS (2013). BabylonJS. https://www.babylonjs.com/. Accessed: 2018-08-27. 119

BabylonJS (2014). BabylonJS playground. http://www.babylonjs-playground.com/#J5E230#54. Accessed: 2018-08-27. 119

Bourdin, J. (2016). MOOCs in computer graphics. *Proc. Eurographics-Education Papers*. 30, 105

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM. 102, 105

Brochu, T. and Bridson, R. (2009). Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472–2493. 86

Brown, M. H. and Najork, M. A. (1996). Collaborative active textbooks: A web-based algorithm animation system for an electronic classroom. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 266–275. IEEE. 109

Cabello, R. (2013). Three.js examples (camera). https://threejs.org/examples/#webgl_camera. Accessed: 2018-07-29. 52

Cabello, R. (2018). three.js on GitHub. https://github.com/mrdoob/three.js/. Accessed: 2018-07-22. 118

Celes, W. and Corson-Rikert, J. (1997). Act: an easy-to-use and dynamically extensible 3D graphics library. In *Computer Graphics and Image Processing, 1997. Proceedings., X Brazilian Symposium on*, pages 26–33. IEEE. 7

Chang, X., Yuksel, K., and Skarbek, W. (2017). WebGL and web audio software lightweight components for multimedia education. In *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017*, volume 10445, page 104452H. International Society for Optics and Photonics. 108

chrisdavidmills and other contributors, . (2018). JavaScript basics. https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics. Accessed: 2018-05-25. 16

CodePen (2012). Codepen. https://codepen.io/. Accessed: 2018-08-27. 120

Contributors, V. (2018). History of wikipedia. https://en.wikipedia.org/wiki/History_of_Wikipedia. Accessed: 2018-07-24. 105

Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM. 6

Davidovi'c, D. (2014). The end of fixed-function rendering pipelines (and how to move on). https://gamedevelopment.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on--cms-21469. Accessed: 2018-05-25. 7

Daxenberger, J. and Gurevych, I. (2013). Automatically classifying edit categories in wikipedia revisions. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 578–589. 106

Deepak (2015). differences between WebGL and OpenGL. https://stackoverflow.com/questions/8462421/differences-between-webgl-and-opengl. Accessed: 2018-05-25. 8

Denavit, Jacques; Hartenberg, R. S. (1955). A kinematic notation for lower-pair mechanisms based on matrices. In *Trans ASME J. Appl. Mech 23*, page 215221. ASME. 69

Dirksen, J. (2013). *Learning Three.js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd. 6

Disney (2015). Wayfinding with code - hour of code. http://partners.disney.com/hour-ofcode/wayfinding-with-code. Accessed: 2018-08-25. 126

Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., and Vigna, G. (2013). dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 1205–1216. ACM. 16

Duchowski, A. T., Angel, E., Gooch, B., and Luebke, D. (2017). CGEMS: Computer graphics educational material. In *ACM SIGGRAPH 2017 Panels*, page 3. ACM. 108

Fiedler, G. (2004). Fix your timestep! how to step your physics simulation forward. https://gafferongames.com/post/fix_your_timestep/. Accessed: 2018-08-30. 72

Finley, K. (2011). Github has surpassed Sourceforge and Google Code in popularity. https://readwrite.com/2011/06/02/github-has-passed-sourceforge/. Accessed: 2018-07-20. 112

frenchtoast747 (2015). webgl-obj-loader. https://github.com/frenchtoast747/webgl-obj-loader. Accessed: 2018-08-30. 77

Gilbert, J. and Albeza, B. (2017). WebGL 2 lands in Firefox. https://hacks.mozilla.org/2017/01/webgl-2-lands-in-firefox/. Accessed: 2018-08-25. 127

Gousios, G., Vasilescu, B., Serebrenik, A., and Zaidman, A. (2014). Lean ghtorrent: GitHub data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387. ACM. 112

Govil-Pai, S. (2006). *Principles of Computer Graphics: Theory and Practice Using OpenGL and Maya®*, volume 190. Springer Science & Business Media. 6, 124

greggman (2015). Twgl.js. https://github.com/greggman/twgl.js. Accessed: 2018-08-27. 123

greggman (2018). WebGL 2 lessons starting from the basics. https://github.com/greggman/webgl2-fundamentals. Accessed: 2018-08-30. 124

Ha, D. and Schmidhuber, J. (2018). World models. 107

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM. 9

Haines, E. (2014). Interactive 3d graphics (creating virtual worlds). https://www.udacity.com/courses/cs291. Accessed: 2018-07-22. 118

Hartmann, B., Wu, L., Collins, K., and Klemmer, S. R. (2007). Programming by a sample: rapidly creating web applications with d. mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, pages 241–250. ACM. 105, 113

Hearns, M. (2014). Literacy game in a virtual world. *Northern Hub Project Fund*. 126

html5gamedevs (2013). Babylon.js forum. http://www.html5gamedevs.com/forum/16-babylonjs/. Accessed: 2018-08-27. 119

Hu, W., Lei, Z., Zhou, H., Liu, G.-P., Deng, Q., Zhou, D., and Liu, Z.-W. (2017). Plug-in free web-based 3-d interactive laboratory for control engineering education. *IEEE Transactions on Industrial Electronics*, 64(5):3808–3818. 127

Hughes, J. F., Van Dam, A., Foley, J. D., McGuire, M., Feiner, S. K., Sklar, D. F., and Akeley, K. (2014). *Computer graphics: Principles and practice*. Pearson Education. 7

Jacobs, J., Gogia, S., Měch, R., and Brandt, J. R. (2017). Supporting expressive procedural art creation through direct manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 6330–6341. ACM. 125

Jeremias, P. and Quilez, I. (2014). Shadertoy: Learn to create everything in a fragment shader. In *SIGGRAPH Asia 2014 Courses*, page 18. ACM. 116

JSFiddle (2010). Jsfiddle: Create a new fiddle. https://jsfiddle.net/. Accessed: 2018-08-27. 120

Kalelioğlu, F. (2015). A new way of teaching programming skills to k-12 students: Code.org. *Computers in Human Behavior*, 52:200–210. 105, 125

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. 32, 105

Labschütz, M., Krösl, K., Aquino, M., Grashäftl, F., and Kohl, S. (2011). Content creation for a 3D game with Maya and Unity 3D. *Institute of Computer Graphics and Algorithms, Vienna University of Technology*. 6, 124

Larson, S. M., Snow, C. D., Shirts, M., and Pande, V. S. (2009). Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. *arXiv preprint arXiv:0901.0866*. 5

Lydell, S. (2016). js-tokens. https://github.com/lydell/js-tokens. Accessed: 2018-07-23. 22

Malvino, A. P. and Malvino, J. M. (1989). Textbook with animated illustrations. US Patent 4,854,878. 33

McCarthy, L. (2013). p5.js. https://p5js.org/. Accessed: 2018-08-24. 115, 119

Misztal, M. K. (2010). *Deformable simplicial complexes*. PhD thesis, Technical University of Denmark (DTU). 129

Mott, J. (2018). Object-oriented JavaScript: A deep dive into es6 classes. https://www.sitepoint.com/object-oriented-javascript-deep-dive-es6-classes/. Accessed: 2018-05-25. 17

O'Keefe, M. (1989). Hyperstudio [computer program]. el cajon. 126

Perez, F. and Granger, B. E. (2015). Project jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September*, 11:207.

PlayCanvas (2014). Playcanvas. https://github.com/playcanvas/engine. Accessed: 2018-08-25. 127

PlayCanvas and Mozilla (2017). After the flood. https://playcanv.as/e/p/44MRmJRU/. Accessed: 2018-08-25. 127

Rego, N. and Koes, D. (2014). 3Dmol.js: molecular visualization with WebGL. *Bioinformatics*, 31(8):1322–1324. 110

Rosenberg, M. N. (2015). Exploring the role of large-scale immersive computing environments in collaboration between engineering and design students. *ProQuest Dissertations Publishing (Iowa State University)*. 110

Ross, R. (1995). Education forum: Animated textbooks: A current example. *SIGACT News*, 26(1):40–43. 33

Selvik, A. L. (2016). dwitter. https://www.dwitter.net/. Accessed: 2018-08-24. 116

Selwyn, N. and Gorard, S. (2016). Students' use of wikipedia as an academic resourcepatterns of use and perceptions of usefulness. *The Internet and Higher Education*, 28:28–34. 6, 105

Sengstacke, P. (2016). Better JavaScript with ES6, Pt. II: A deep dive into classes. https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes. Accessed: 2018-05-25. 17

Sherif, T. (2015). The brainbrowser surface viewer. In Cozzi, P., editor, *WebGL insights*. AK Peters/CRC Press. 110

Simpson, K. (2015). *You Don't Know JS: ES6 & Beyond*. " O'Reilly Media, Inc.". 10

Söderberg, J. (2015). *Hacking capitalism: The free and open source software movement*. Routledge. 111

Software, F. C. (2000). Glitch. https://glitch.com/. Accessed: 2018-08-24. 113

Staub, T. and Hodel, T. (2016). Wikipedia vs. academia: An investigation into the role of the internet in education, with a special focus on wikipedia. *Universal Journal of Educational Research*, 4(2):349–354. 105

Terzopoulos, D. (2017). CS 174A animation examples. http://web.cs.ucla.edu/~dt/courses/CS174A/animations/. Accessed: 2018-08-28. 4, 99

Terzopoulos, D. and Rabie, T. F. (1995). Animat vision: Active vision in artificial animals. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, pages 801–808. IEEE. 54

Treude, C. and Aniche, M. (2018). Where does Google find API documentation? *2nd International Workshop on API Usage and Evolution*. 11

Warger, T. (2009). The open source movement. https://web.archive.org/web/20110717100415/http://net.educause.edu/ir/library/pdf/eqm0233.pdf. Accessed: 2011-07-17. 111

Wilson, C. (2014). Hour of code: we can solve the diversity problem in computer science. *ACM Inroads*, 5(4):22–22. 125

Wittern, E., Suter, P., and Rajagopalan, S. (2016). A look at the dynamics of the JavaScript package ecosystem. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 351–361. IEEE. 111

Yannier, N. (2016). *Bridging Physical and Virtual Learning: A Mixed-Reality System for Early Science*. PhD thesis, University of Pittsburgh. 126

Yu, D., Chander, A., Islam, N., and Serikov, I. (2007). JavaScript instrumentation for browser security. In *ACM SIGPLAN Notices*, volume 42, pages 237–249. ACM. 16

Zakai, A. (2011). Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 301–312. ACM. 10

Zubrycki, I. and Granosik, G. (2017). Teaching robotics with cloud tools. In *International Conference on Robotics and Education RiE 2017*, pages 301–310. Springer. 30, 105, 115