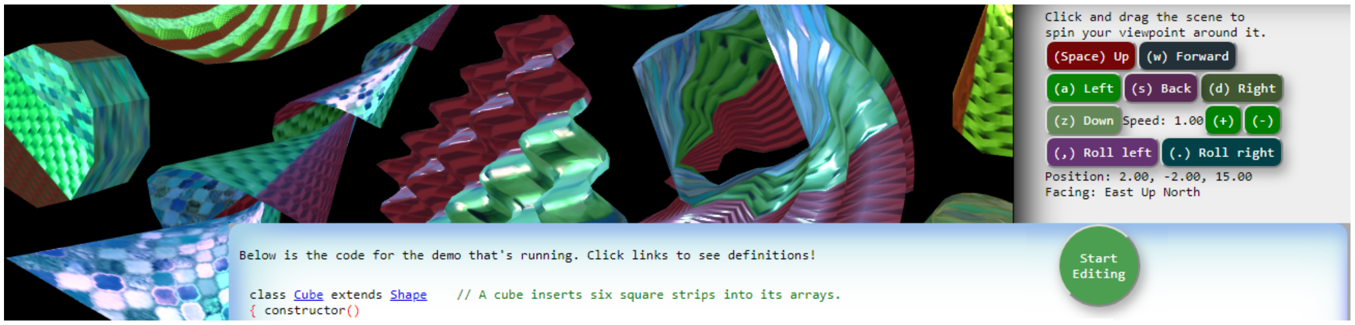
An Online Collaborative Ecosystem for Educational Computer Graphics

Garett D. Ridge
garett@cs.ucla.edu
Computer Science Department
University of California, Los Angeles

Demetri Terzopoulos
dt@cs.ucla.edu
Computer Science Department
University of California, Los Angeles


Click and drag the scene to spin your viewpoint around it.

Position: 2.00, -2.00, 15.00
Facing: East Up North

Below is the code for the demo that's running. Click links to see definitions!

```
class Cube extends Shape    // A cube inserts six square strips into its arrays.
{ constructor()
```

Figure 1: Some interactive panels from our collaborative online code editor, the "Encyclopedia of Code".

ABSTRACT

We introduce a coding framework that supplements introductory computer graphics courses, with the goal of teaching graphics fundamentals more effectively and lowering the excessive barrier of entry to 3D graphics programming. In particular, our framework provides tiny-graphics.js, a new WebGL-based software library for implementing projects, including an improved organization system for graphics code that has greatly benefited our students. To mitigate the difficulty of creating 3D graphics-enabled websites and online games, we furthermore introduce the "Encyclopedia of Code"—a World Wide Web framework that encourages visitors to learn 3D computer graphics, build educational graphical demos and articles, host them online, and organize them by topic. Our own contributed examples include various interactive tutorials and educational games. Some of our modules expose students to new graphics techniques, while others explore new modes of online learning, collaboration, and computing. In comparison to earlier online graphics coding platforms and mainstream graphics educational materials, the resources that we have developed offer a significantly unique set of features for both inside and outside our classrooms.

Web3D '19, July 26–28, 2019, Los Angeles, CA, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6798-1/19/07…$15.00
https://doi.org/10.1145/3329714.3338133

CCS CONCEPTS

• **Computing methodologies** → **Computer graphics**; • **Applied computing** → *Computer-assisted instruction*; • **Software and its engineering** → Software libraries and repositories.

KEYWORDS

Computer Graphics education; WebGL; tiny-graphics.js; JavaScript Library; Encyclopedia of Code.

ACM Reference Format:
Garett D. Ridge and Demetri Terzopoulos. 2019. An Online Collaborative Ecosystem for Educational Computer Graphics. In *Web3D '19: The 24th International Conference on 3D Web Technology (Web3D '19), July 26–28, 2019, Los Angeles, CA, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3329714.3338133

1 INTRODUCTION

Creating computer graphics visualizations while problem solving is a great way to make topics seem more intuitive. 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 are tasked with finding the answer.

Some students begin graphics courses without a programming or math background, and yet graphics can be a way for them to gain such a background. Even those who are outside the university system can benefit from learning graphics from a programmer's or mathematician's perspective, in order to increase their understanding of both, rather than trying to glean this skill from online graphics tutorials, which may not emphasize such fundamentals.

Unfortunately, due to numerous complex steps, most of today's approaches to creating a low-level, math-based graphics program come with a substantial learning curve. The initial obstacles do not particularly help the learner acquire the math and programming intuition that they ultimately seek, promised by gaining programmatic control over visualizations.

1.1 Contributions

The overarching contribution of our work is to lower the difficulty for graphics learners by eliminating steps, and to make the job of graphics instructors easier as well. We offer source code and 3D web-based content that can supplement a university or college-level Computer Graphics course. We also offer a novel internet-based framework for supporting and expanding that content.

First, we provide "tiny-graphics.js" (on GitHub, at URL https://github.com/encyclopedia-of-code/tiny-graphics-js), a new WebGL-based programming library for implementing projects in the classroom. It is a single-file JavaScript utility. Unlike popular 3D graphics frameworks like ThreeJS [Dirksen 2013], tiny-graphics.js is purpose-built for education. It is small enough to accompany an assignment, and has a strong record of such use with our own assignments. It is designed to organize the 3D graphics process into object-oriented modules for the programmer, sparing them from clutter.

Second, we preview a new online coding platform that uses tiny-graphics.js. The website, called the "Encyclopedia of Code" (URL http://encyclopediaofcode.glitch.me/), encourages web visitors to learn graphics. Without installing anything, anyone can use its online editor to build educational graphical demos and tutorials, host them online, and organize them by topic. Our goal is to build a crowd-sourced repository of remixable 3D demos and educational tutorials. This consists of one unified codebase, so our encyclopedia's examples can collectively serve as a programming engine in addition to being a source of documentation and tutorials.

Our platform's mission is to unify and democratize the creation of visual tutorials. We also deliver to the masses their easiest option for the creation and prototyping of low-level 3D programs. Together with our code library and its examples, we introduce new tools and paradigms for education, especially in topics relevant to Computer Graphics. Our unique contribution is a system-wide reorganization of the process of making graphics, thereby making the whole task friendlier for students, researchers, and programmers.

2 RELATED WORK

Our tiny-graphics.js software library serves as a direct replacement of software-based course materials designed by Edward Angel. Angel created new ways to teach graphics, and was additionally part of an effort to unify graphics education at *ACM SIGGRAPH 2017*, calling for submissions of unique and interesting assignments for graphics courses [Duchowski et al. 2017]. Our work aspires to similar ends.

The tiny-graphics.js library has accompanied our university's *Introduction to Computer Graphics* course assignments since 2016, replacing our use of supplemental code from the textbook by Angel and Shreiner [2014] entitled *Interactive Computer Graphics: A Top-Down Approach with WebGL*. As this widely cited book has been used as the basis for SIGGRAPH's introductory graphics courses in recent years [Angel and Haines 2017; Angel and Shreiner 2016], it sets the standard for graphics education. Compared to Angel's library, our tiny-graphics.js library has surpassed its original scope in many ways, offering improved educational utility, organization, functionality, and performance.

The most famous of other efforts to pre-organize the process of 3D graphics programming have resulted in large engines such as ThreeJS for WebGL [Dirksen 2013], or for offline use, large proprietary commercial graphics modeling suites like Maya [Govil-Pai 2006]. Angel [2017] points out that such large frameworks are unsuitable for the engineering classroom, since they purposefully hide low-level architectural details that engineering students need to see. Our own tiny-graphics.js is purpose-built for education and keeps students close to these details. Its current scope is focused merely on easing students into the organization of the graphics process, such as managing shaders and geometry. In its current phase, it should not be compared to industrial tools like ThreeJS outside this educational niche because of an important difference: Our source code is small enough for a student to read and understand completely, including all the WebGL calls inside. The same cannot remotely be said of ThreeJS's source code. Besides being an API, our library is meant via its size and readability to suggest an organizational scheme for all graphics programs, not just ours.

There have been prior attempts to integrate graphics courses with the internet. Angel [2017] describes using his textbook and the Coursera platform to host a Massive Open Online Course (MOOC), which 5,500 students began. Bourdin [2016] evaluated the few Computer Graphics MOOCs in existence by supplying their own students. They identified several pitfalls, including a low completion rate attributed to the demotivating impersonal aspect of online courses. Their students lost the Socratic eagerness to impress the (distant) teacher. Our online platform compensates for this by allowing submission of finished demos, sharing, and showing off.

Project Jupyter [Perez and Granger 2015] includes interactive web editors and tutorials not unlike our educational platform. Jupyter Notebooks are a means of publishing a computational method that can be readily read and replicated using a web browser. They embedded panels of code, prose, and results within the HTML document. Lines of code in their editors appear in an order that suits the documentation presented around them, an integral characteristic of Knuth's Literate Programming [Knuth 1984]. Our system goes further than Project Jupyter in several ways. The full interactivity of Notebooks is hidden from casual, untrained web visitors; to see the Notebooks as more than a static page requires either third party web tools or the installation of Python packages. Our own "Active Textbooks" described in Section 4.2 can instantly benefit all web visitors, not just programmers or power users. Secondly, Notebooks are hosted by individuals (such as on GitHub). There is no central repository like ours that easily allows users to generate and host new Notebooks themselves for purposes of remixing or rapid experimentation. Lastly, Notebooks are often graphical in nature, but usually only by plotting 2D charts and graphs. They do not generally use 3D or involve the graphics pipeline or GPU in any way, nor do they create 3D HTML canvas contexts (WebGL) as we do. Notebooks are thus less suitable for teaching 3D graphics or helping others develop low-level graphics programs, engines, or games.

Code.org is a nonprofit dedicated to expanding access to Computer Science, and they investigated the effects of their own programming instruction website [Kalelioğlu 2015]. 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. The online games listing for the Hour of Code collects the same sort of interactive educational programming demos that our Encyclopedia of Code seeks to crowd-source, and some of them even use WebGL. However, the workings of their games is not shown, which is a lost opportunity to educate. Many of their sponsors' contributed games overlap in topics of basic programming, quickly bottoming out in educational value. With our platform's encyclopedia organization, there are as many opportunities to engage students with such demos as there are topics in the Computer Science curriculum. Our platform, unlike theirs, includes free hosting for individual visitors to use for sharing new educational games or for remixing programs during experimentation.

The arts-supporting "Processing" project [McCarthy et al. 2015] includes the p5.js JavaScript library for making interactive graphical code editors and tutorials, as well as a website containing examples on diverse topics. Academics have used its embeddable web panels to create cloud tools, enabling their students to experiment with graphics and robotics programming [Zubrycki and Granosik 2017]. The p5.js editors have no "save" button nor free re-hosting.

Hartmann et al. [2007] made an academic attempt to democratize application development via collaborative coding. Like our project, their "d.mix" tool 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. Only certain major website APIs were supported, and unlike our web-based system, d.mix must be installed to be used.

Selwyn and Gorard [2016] explored modern students' use of Wikipedia as an academic resource. Although the Wikipedia model of collaboration has produced comprehensive educational materials, our platform can offer additional dimensions of 3D visualization, interactivity, and automation. Wikipedia articles are purely reading resources generally without an associated program for the computer to work on while the user reads. On our our platform, the computer's workload when rendering an article can either draw a visualization for the current visitor or perform topic-relevant calculations that are cached server-side to benefit concurrent or future visitors.

Similar websites such as BabylonJS's tutorials, WebGL Playground, Glitch, D3JS, and Shadertoy exist, but each merely includes a specific subset of the features our platform offers, while other important features are missing, such as interactivity in documentation, free hosting and remixing for individuals, transparency, or control over the entire source code as opposed to a single sub-component or shader program.

3 MOTIVATION

In terms of preparation and learning, it is costly to build prototypes using low-level 3D graphics programs today. A common alternative is to forgo low-level control, and to employ overpowered industrial tools to wrap basic graphics functionality—the simple mathematics of projecting 3D triangles onto a 2D plane of pixels.

Graphics beginners are faced with long lists of setup steps, especially in the case of newer "shader-based" approaches.[1] These approaches are both more complex and harder to learn. The graphics learning curve is extreme. Drawing just a single triangle requires secondary "shader" programs, multiple types of GPU memory management, and at least three computer languages (in the case of WebGL). Mandatory setup steps to initialize the graphics card (GPU) give it the shader program code and any raw data buffers pertinent to the 3D scene, and obtain pointers to these in GPU memory. Even after this setup, all the subsequent shape-drawing actions of the programmer are still cluttered with boilerplate code for loading and switching between pointers to the GPU. Nothing is drawn without these steps [Angel and Shreiner 2014]. There is no built-in way to organize them.

Common graphics card interfaces exposed for doing the above steps are called DirectX and OpenGL, the latter being more widely available and more prevalent in education [Angel and Shreiner 2014]. Regardless, graphics methods all follow a similar pattern. The phrase "OpenGL program" is widely taken to imply C++ due to the ubiquity of the language in early graphics education, but it need not be—Python, Java, and JavaScript can make the same OpenGL calls. 3D Graphics programming in one language feels familiar in all the others.

JavaScript is currently the only means of running code on browsers inside of web pages. When JavaScript is used, OpenGL (version ES) commands can be used which are then called WebGL, but they are still the same API function calls as would appear in C++.

Angel [2017] described his rationale for eventually moving his helpful C++ based libraries over to WebGL. He lamented a worsening learning curve of C++ graphics setup, citing difficulty in setting up uniform C++ compiling environments for all students, who also have inconsistent hardware. Angel found that WebGL has comparable performance to C++, plus the advantages of a standardized environment on all platforms (including phones), an interpreted code engine that aids development, and advanced coding tools built right into modern web browsers.

We too have found WebGL to be the current best platform for graphics instruction and training. Code examples that run inside of websites are more easily analyzed, packaged with inline tutorials, compiled to any machine, debugged, hosted, shared, and remixed. JavaScript's presence of functional programming styles have specific graphics applications, and tend toward smaller total source code. We make full use of the 2015 "es6" version of JavaScript, which adds further brevity and power to the language. Performance workarounds such as WebAssembly and Emscripten can run at near native speeds. JavaScript's benefits for prototyping and sharing code on the web are clear.

4 THE TINY-GRAPHICS.JS LIBRARY

The tiny-graphics.js file factors away the repetitive logic of GPU communication into reusable objects. It gives a JavaScript program access to linear algebra routines, useful user-interface controls and

[1]Graphics tutorials online abound that still use outdated "pre-shader" coding paradigms, which are alluring due to their simplicity. However, support for their older commands has been removed from new graphics cards, and never existed in web browser implementations. Eventually it dawns on newcomers that they must commit to learning the new way [Davidovi'c 2014].

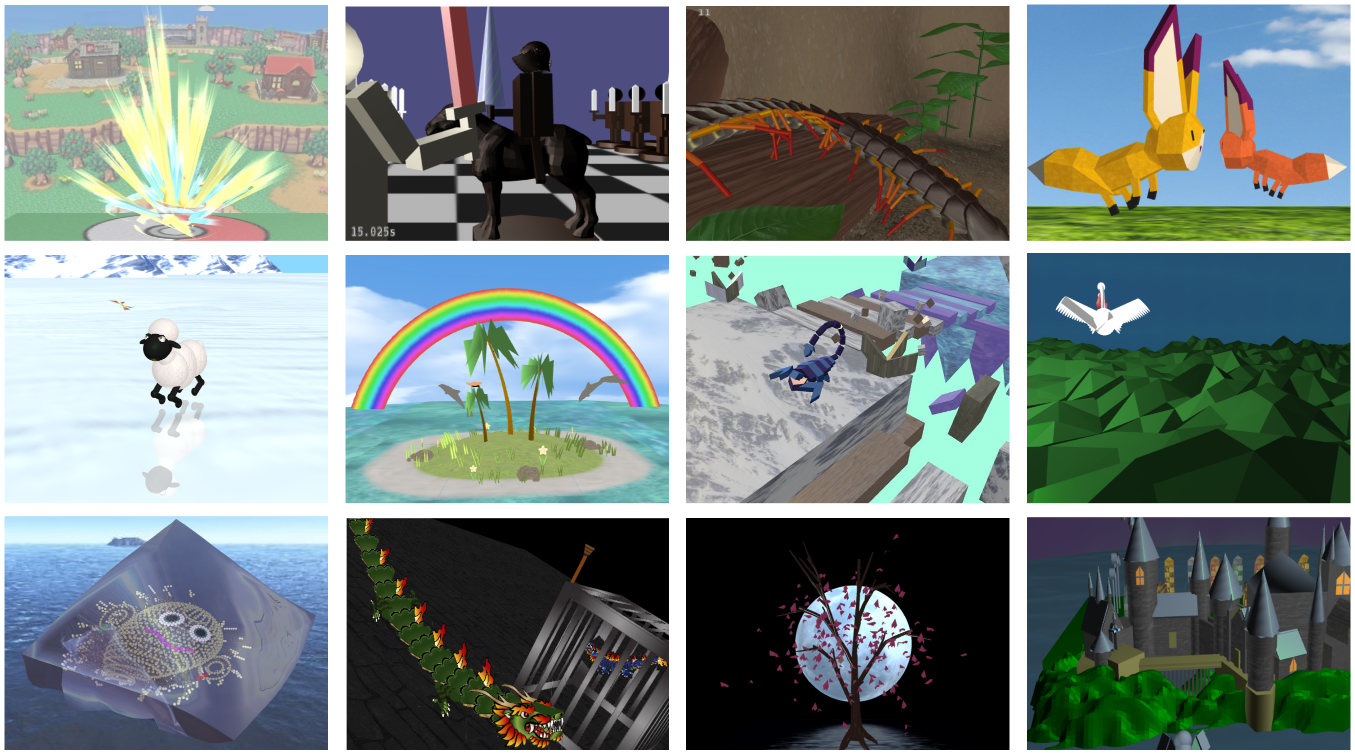


Figure 2: These animation projects were created by students using our tiny-graphics.js library during offerings of our university course *Introduction to Computer Graphics*. The library has been an effective educational tool, both for easing the students into learning graphics programming, and for enabling their programs to achieve visual results like these using little more than mathematics. This was a motivating force in conceptualizing and implementing course projects. Section 6 presents our observations about the project submissions.

readouts, and drawing utilities needed by modern shader-based graphics.

The tiny-graphics.js framework has been used during eight offerings of our university's course *Introduction to Computer Graphics*. It was rewritten for each new generation of students using prior students' feedback about points of confusion. The instructors of our course attest that tiny-graphics.js is enabling students to progress further by the end of the course than before, as evidenced by what topics appear in their submitted term projects. Our term projects consist of a short video or game that students present in a competition (Figure 2).

Prior to tiny-graphics.js we used the small linear algebra software library provided with the textbook [Angel and Shreiner 2014] enabling students to get started with hands-on graphics programming experience. Angel included supplemental code and links to demos that run inside websites. Since these demos are presented as fully working units, we found that our students liked adapting them when beginning projects. However, we observed students would later have trouble and produce spaghetti code when extrapolating these demos to the desired complexity of a full project, having been locked in to the textbook's loose organizational scheme.

The second chapter of the textbook is spent building a "hello world" like program that displays one triangle, showing students the minimal code needed to draw graphics. This leaves unanswered the question of how to organize a larger graphics program, the sort with multiple scenes, which our students' projects attempt. There is no consensus on an answer across WebGL tutorials that can keep all the unrevealing and cluttersome "boilerplate" code from being a distraction to the educational experience. Typically, as in Angel's demos, several lines of GPU management code are mixed in between every action.

Our chosen organization factors away this repetitive logic, hiding the boilerplate code. The source code region inside which students work exposes only the concepts that matter to the student. In some examples especially (Section 5.2.6), students only see lines of code about concepts with which they expect to deal, such as the matrix transforms and vector math that their instructor describes in lecture.

4.1 Improvements

Our tiny-graphics.js library transcends the approach in [Angel and Shreiner 2014] and other academic tutorials by organizing the user's graphics code, while remaining flexible enough to allow the programmer to selectively choose dynamic pieces of their program. Those pieces are typically shape vertex arrays, shader programs, texture images, or entire scenes.[2] Our first improvement was to recognize all of those as important logical units into which a graphics program naturally subdivides; our library therefore encapsulates each of these pieces in an object-oriented structure.

We designed tiny-graphics.js to flexibly switch between these encapsulated structures, while fitting in a single easily digestible file small enough to supplement a textbook or an assignment. It provides Document Object Model (DOM) elements and keyboard interfaces so user input can trigger the activation of different encapsulated structures in the program. It sets up the elements of graphics programs, such as shapes, shaders, and scenes, each encapsulated in a single JavaScript class made by extending base classes such as Shape and Shader. The base classes commit their data to GPU memory when necessary, and record to which HTML canvas region they are registered in case multiple display areas should appear on the website. By encapsulating scenes as objects, the library simultaneously draws potentially multiple scenes per canvas in an event-driven fashion. Additional classes manage the overall WebGL process.

4.2 Active Textbooks

Traditionally, demos provided to supplement a programming textbook suffer from the inherent limitation of existing on a separate medium from the textbook itself. Students must interrupt their reading and switch to a computer to view them. To get around this structural restriction, we now describe how we combine educational materials with our demos.

Our library's 3D animations can benefit websites across the whole internet due to our embeddable content. All the different types of UI content tiny-graphics.js can display on a web page are each procedurally generated as embeddable, highly portable panels. We insert UI panels into the DOM tree wherever the user has left a placeholder HTML tag in their web document, accommodating any existing content and layout. In JavaScript the user simply instantiates certain UI "Widget" classes defined in our object-oriented system, which we considered as more logical units to break off from the process of presenting graphics. This short JavaScript call also can appear alongside the placeholder tag, leaving the rest of any existing document intact with minimal footprint.

The resulting system is quite flexible. When the user puts a "Canvas Widget" from our library on their page, they will see a working WebGL region with an interactive 3D program. The user may pass into it any defined JavaScript "Scene" subclass that draws something.

To display the code of that "Scene" subclass instead, the user would pass that same Scene object into a "Code Widget" and embed that on their page. This generates a color-coded navigator that highlights how some or all of the source code works, including inline links to definitions wherever a class name appears.

[2]Vertex arrays are a necessary part of today's graphics programs. They are a chunk of arbitrary data that is divided up per point, supposing a collection of points. The data is copied to the GPU and re-used to draw similar shapes efficiently. Shaders are secondary programs, separate from the JavaScript and in their own language, written for the GPU to execute. A shader program's job is to define two things: Where a drawn shape will land onscreen, and how to color in the affected pixels. A sub-program called the "vertex shader" performs the former, and the "fragment shader" performs the latter. Potentially, a different shader program could be used for each shape drawn onscreen.

To see a textual explanation or tutorial article about the demo in question, the user would pass that same Scene object into a "Text Widget". The HTML documentation will be extracted out of the same JavaScript class, since we package documentation generators inside Scene code. The Text Widget panels can recursively contain the other panels. This means that the final document can alternate between textual documentation, a panel that highlights a piece of code, or an interactive 3D demo of what has been made so far that progressively grows throughout the article, in as many combinations as needed to walk the student through a program's workings.

This capability, which we dub "Active Textbooks", allows our users' creations to resemble Jupyter Notebooks [Perez and Granger 2015], but with the power of WebGL, including interactive 3D animated areas. Our tutorials can mix interactive areas, 3D scenes, code navigators, and most importantly for programmers, code editors. By mixing documentation with code and presenting the code in any desired order, we capture the essence of Knuth's Literate Programming [Knuth 1984] for a new, graphics-focused audience.

This year, all the documentation of tiny-graphics.js (describing how various shapes, shaders, and tiny-graphics.js itself are designed) is being converted into this Active Textbook form. The goal is for the student to learn how to build their own low-level graphics engine by showing them how ours is built, piece by piece, stopping along the way to show intermediate 3D results with which the student may interact, or code snippets that grow in complexity throughout each tutorial. By showing how to make our engine, we pursue the ideal of everyone being able to make a game or graphics engine themselves rather than merely adapting one whenever they want to visualize any mathematical or graphical phenomenon on a computer.

In the following section we will present our web platform for spreading this information to the masses and democratizing the creation of 3D graphics prototypes online.

5 THE ENCYCLOPEDIA OF CODE

Introductory graphics courses typically end by covering some of the diverse special effects topics and applications explored by the industry. These industry techniques normally appear in the computer graphics research literature or even make their way into textbooks. We use them as "extra credit" topics students may add to their term projects. Concluding the course this way encourages students to branch off into different areas of the graphics research field.

A graphics course therefore benefits from having a supplemental repository of graphics effect examples. By collecting these we can better field questions about the specialized branches of graphics each student wants to explore. Thanks to tiny-graphics.js, it has been fairly easy for us to create new educational examples on any given topic. We have developed a set of working demos as supplemental class materials. In addition to showing how to use tiny-graphics.js in a variety of ways, they cover various graphics techniques and applications. Some stand alone as educational pages about the mathematical underpinnings of graphics, thereby extending educational benefits to a wider population than just the users of tiny-graphics.js.

Our "Encyclopedia of Code" thereby emerged from our university's *Introduction to Computer Graphics* course. It consists of a web server (implemented with Node, Express, and MongoDB) with hosting from glitch.com.

5.1 Innovation

Our crucial realization was that we can crowd-source more of these special topics demos from other graphics experts, enthusiastic students, or hobbyists using the Internet.

Using standard web forms, our online coding platform allows ambitious users of our library to add their own demos of computer graphics effects, ideally packaged with documentation and tutorials written using our Active Textbook model from Section 4.2. Each new demo gets placed on its own sub-address URL. Most contributions so far are subtle modifications of existing demos, generated as visitors play around with code and observe the graphical consequences of each change.

Our back-end database is structured such that all the demos stored there are class definitions in JavaScript source code. Each of them can access others within the same codebase. Compared to services like GitHub that isolate individual projects [Dabbish et al. 2012], anyone writing code for our web platform can instantiate any JavaScript class definition ever submitted. It is acceptable to maintain such a full namespace because we encourage the cumulative submissions to take the form of an encyclopedia about everything. Within encyclopedias, one expects a name to map uniquely to a definition. Our design promotes the eventual expansion of our articles into an openly editable online encyclopedia with broad coverage, much like Wikipedia [Selwyn and Gorard 2016], but specialized for any topic better served by 3D visuals and interactivity.

A crowd-sourced set of JavaScript classes will be large, so rather than sending the entire stored codebase to every web visitor, we send source code as small, digestible educational pieces. We use dependency injection to deliver minimal programs to visitors instead of the whole codebase. In some cases only three classes (implementing each of our base classes Scene, Shader, and Shape) need injection into the website's source code—this happens to be the minimum amount of JavaScript necessary to use tiny-graphics.js. Supplying small subsets of our stored source code ensures good performance on our client pages. In addition, only the few injected classes are displayed in the user's code navigator, so the reader does not see anything extraneous when trying to understand any example programs the Encyclopedia of Code shares.

In summary, our goal is to allow anyone to make a 3D graphics prototype without any large source code dependencies just by visiting a website, an option that did not appear to exist previously. Most existing web-based coding platforms for fast graphics prototyping focus on 2D graphics, or lack hosting. With our platform, coders can post their result for others to remix and adapt.

5.2 Example Pages in the Encyclopedia of Code

Next, we describe how the Encyclopedia of Code has been applied in the classroom, via specific example 3D applications we have hosted that offer coding help or instruction. These examples help the instructors of our course explain graphics effects and field topic-specific questions from students. Furthermore, our demos offer tiny-graphics.js as a better starting point for students who wish to adapt it into projects, and our instant online code editor makes this process easy. The following sections describe our example pages.

5.2.1 Minimal Executable Demo. Navigation to this demo reveals roughly the smallest possible program that can be implemented with tiny-graphics.js. This colors in a single triangle. Despite its simplicity, it shows students the organization scheme of tiny-graphics.js. This demo comes with a "Code Widget" UI panel (Figure 1) for source code navigation, which shows the visitor the rather short JavaScript class that generated the graphical scene. Two inline links that are visible in the code are one trivial shape definition (the triangle) and one trivial shader program. Clicking nearby links reveals all the other source code that came with it on the web page, including tiny-graphics.js. As described in the previous section, our web server will not send any extra code classes not needed by the demo.

5.2.2 Dynamics Demo. This page shows an animation of falling shapes (Figure 3). It teaches students how to animate shapes smoothly, in incremental motions that are not restricted to known paths. To do so, it simulates physical dynamics to compute linear and angular velocity. The demo and its documentation show students how to organize timestepping for maximum smoothness and reliability according to the famous blog post "Fix Your Timestep!" by Fiedler [2004], which decouples the simulation step rate from the frame rate by interpolating steps.

Compared to the previous demo, this one requires more JavaScript classes from our server, particularly for drawing the various shapes that appear onscreen. Our server injects code for these into the page, which in turn causes our displayed code navigator to include many unique shape definitions. Scrolling down past the animation reveals these definitions injected into the dependencies.js file, and the visitor can click through them to explore the program. Our demo has been helpful to students who have attempted the realistic animation of 3D shapes, and they have incorporated its code into many of their term projects.

5.2.3 Collision Demo. This demonstrates how to approximately detect when 3D geometry collides using a simple mathematical heuristic that involves change of bases and discretization. It animates a variety of flying shapes to contrast them with the hidden collision shape. The geometric volumes stop moving and stick together in a structure whenever their rotations or translations cause them to touch. This demo manages its time-stepping, animation, and shape generation by re-using all of the same JavaScript classes as the previous demo, but its definition and accompanying explanation teach something different.

5.2.4 Matrix Game. Matrices are the main mathematical tool for moving shapes into place in graphics. This educational game teaches the concept that we observed our students to miss most frequently on exams—the order of matrix multiplications. Our students have difficulty performing the correct change of basis that reflects what they are intuitively trying to do to place geometry in the virtual world. Mistakes are not necessarily made when selecting an individual matrix to change the basis, but when choosing the correct left to right ordering of several matrices in a product.

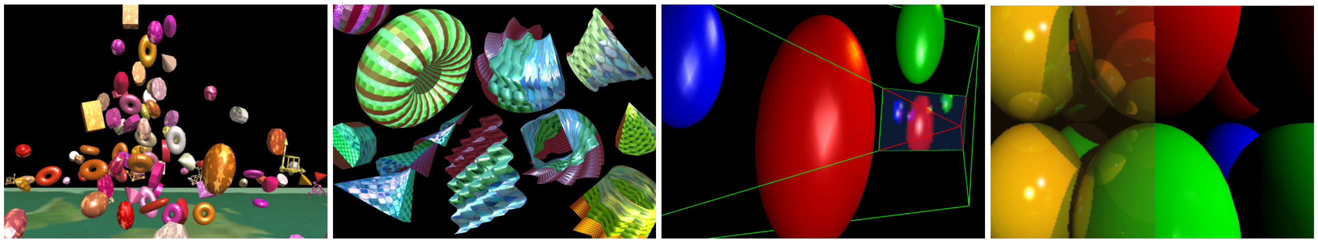
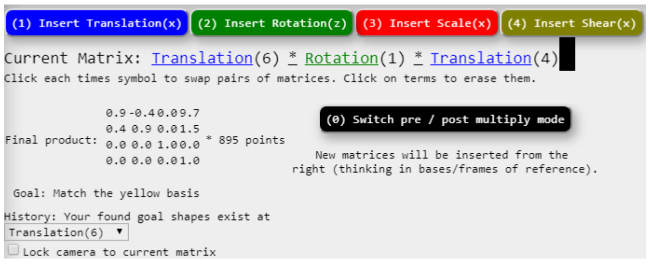


Figure 3: From left to right, our demos on dynamics, surfaces, frustums, and ray tracing.



(1) Insert Translation(x)
(2) Insert Rotation(z)
(3) Insert Scale(x)
(4) Insert Shear(x)
(0) Switch pre / post multiply mode

```
Current Matrix: Translation(6) * Rotation(1) * Translation(4)
Click each times symbol to swap pairs of matrices. Click on terms to erase them.

                0.9-0.40.09.7
                0.4 0.9 0.01.5
Final product:  0.0 0.0 1.00.0  * 895 points
                0.0 0.0 0.01.0
                                        New matrices will be inserted from the
                                   right (thinking in bases/frames of reference).

 Goal: Match the yellow basis

History: Your found goal shapes exist at
Translation(6)
Lock camera to current matrix
```

Figure 4: Colorful buttons control the Matrix Game. The game gives 3D feedback and also shows the matrix product mathematically, in two different ways.

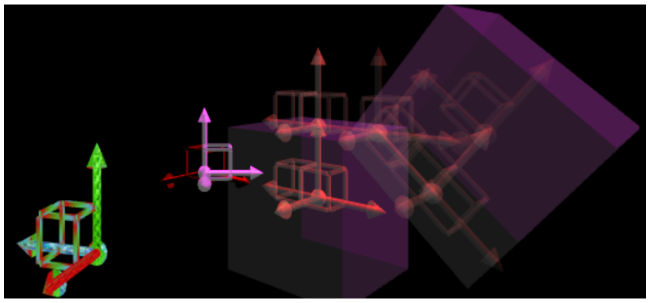


Figure 5: The second level of the Matrix Game, with targets (axis arrows) whose transforms oscillate as functions of time. Their placements correspond exactly to the matrix operations that a programmer would iteratively make in order to draw the two faded, hinged cubes.

Our hardest exam questions ask students to choose between several alternate orderings of the same matrices to accomplish a certain geometric movement. When programming, they will similarly need to sequence matrices together in the correct order to draw shapes. Since matrix products are incrementally built within an intermediate variable, it is difficult to build good intuition about the whole product.

Our "Matrix Game" (Figure 4) depicts a coordinate system as axis arrows and includes a few buttons, one for each of the basic affine transformation matrix types. Students can first learn what translation, rotation, scale, and shear matrices are by observing how they modify the drawn local reference frame. For a clearer understanding, below the animation the final matrix product is written two ways—as abbreviated basic matrix terms multiplied together, and numerically as a final product affine matrix. Both update live as the student presses buttons. A cursor shows where new matrices will be inserted; students can switch between post-multiplying and pre-multiplying new terms to generate either matrix order, while observing how that changes the drawing, thus improving their intuition.

Failure to mentally separate the code from the mathematics that students are actually doing has been a clear source of confusion. It is therefore important for students to see the written forms of matrix products independently of the lines of code they would normally type to multiply matrices in an incremental fashion. This helps disabuse them of the flawed common question of which matrix happens "first" or "last".

Students can optionally play our interactive matrix visualization as a game, in which they chase a yellow coordinate basis drawing as it flies to different locations. Each goal must be reached with a single edit to the product. The student must reason whether to use pre or post multiplication when appending additional basic affine transformation terms to their net transformation. Some gameplay moments are directly based upon several of our course's previous exam questions with which students had trouble, such as one goal that involves a rotation action counter-intuitively changing the lengths of the axis arrows due to the presence of a scale matrix.

Our game includes additional "levels" with further educational value. In one that teaches animation concepts, the student appends matrices that vary over time (such as by rotating at a constant rate or periodically back and forth). The student chases moving targets and ultimately traces out the transformation matrices needed to perfectly hinge two cubes at their corners (Figure 5), addressing a common student question. Another level of the game explores the rules of the inverse of matrix products, which students encounter when they deal with camera matrices.

Employing this tool in class live on a projector for pre-exam review saves the instructor time and energy compared to the traditional alternative of drawing several of these situations on a chalkboard. Since it is public, this web page can benefit graphics instructors the world over regardless of whether or not they are using tiny-graphics.js.

5.2.5 Shapes Tutorial. This tutorial walks students through how to model an increasingly complex progression of shapes, starting with gentle examples like a triangle and tetrahedron, and ending with a subdivision sphere. One pinwheel shape is generated using matrices and iteration. Another example combines several squares into a cube, showcasing the ability of our tiny-graphics.js library to generate performance-friendly compound shapes. This approach likewise built the axis arrow shapes used in our Matrix Game (Section 5.2.4) into a single performance-friendly buffer.

5.2.6 Transforms Sandbox. This page serves as ideal starter code for a first course assignment, since its animation contains only lines of code that deal with matrix transformation steps and single-line drawing commands. To do this, it shares code with our shapes tutorial, but as a JavaScript subclass that exposes only the function that renders each frame of the scene. This hides all initialization of shapes, Phong lighting, and materials in the superclass until students are comfortable enough to look there.

5.2.7 Surfaces Demo. This example includes code that generates arbitrary surface patches and surfaces of revolution. These range from simple shapes such as cylinders and cones to complex functions that sample and interpolate arrays of points (Figure 3). Most shape definitions require less than a dozen lines of code thanks to a helpful utility function that we provide for the generation of these shapes. This function allows the user to pass arbitrary operations as callbacks to the generator. The operations incrementally specify one point's location based on the previous point. We use these input callbacks to freely deform a triangulated sheet of rows and columns. Our Surfaces Demo is an introduction to drawing smooth shapes in the general case. It implements numerous shapes each in only a few lines of code, which provides students many possible examples to adapt into their own custom shapes as they learn basic 3D modeling.

5.2.8 Frustum Viewer. This page explains the workings of view volumes, universal tools in graphics for projecting 3D shapes onto 2D screens. The animation draws view volume shapes superimposed onto a 3D scene. This is a visual diagnostics tool that can augment any loaded scene, and does so to concurrently draw the view volume of that scene. Figure 3 shows how the frustum can project simple scenes (such as several balls) onto one of its 2D planes using ray tracing, a feature that updates live as the frustum is moved or switched between orthographic and perspective. Our Frustum Viewer promotes understanding of projection matrices, in source code small enough to explore and understand. It furthermore includes a useful utility function that, given any projection matrix, can recover the locations of the corners of the view volume.

5.2.9 Ray Tracer. In our introductory graphics course, we have traditionally issued an assignment that requires students to complete a partial implementation of a real-time CPU-based ray tracer using WebGL, which is shown in its complete form in Figure 3. Our ray tracer adapts the code from our Frustum Viewer to generate rays. This assignment allowed students to explore ray-volume intersections, recursion, and the formulas of Phong shading. As shown in Figure 3, a scanline progresses from left to right, replacing pixels of a non-raytraced rendering of the scene, to render reflections, refractions, and shadows.

5.2.10 Scene Graph Manipulator. Scene graphs are a major topic of computer graphics. These graphs help students not only to organize and re-use parts of their scene, but to reduce the calculations their program must do. A scene graph is a tree data structure that represents the spatial layout of a scene and the logical relationships between its parts. Since each graph node stores a local transformation relative to its parent, the location in space of a particular node is computed by traversing the tree down to the node, while accumulating a product of transformation matrices on the way down.

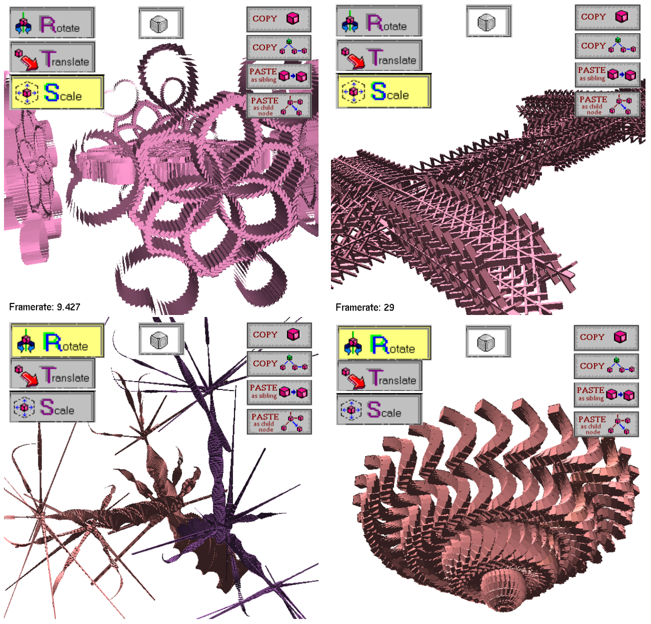


Figure 6: Our Scene Graph Manipulator Demo.

We demonstrate that scene graphs have powerful uses, even in modern shader-based graphics programs that no longer depend on them for managing some internal matrix "stack". 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 and stored matrices. The shape associated with the current graph node immediately redraws, along with the shapes of any child nodes further down the branch, since the relative change propagates down the tree. Our tool especially encourages the reuse of nodes and entire branches, allowing the automated repetition of nodes with the touch of a button. The visually compelling and surprising shapes that result are rewarding and informative to the student. Figure 3 shows some of the shapes that were built in seconds using our tool, with only a rough plan in mind for the final shape. Repeating nodes (and their pattern of transformations) using the UI sends them down a path that might curve and spiral as the repetition progresses. The tool may be used to create the highly symmetric shapes shown by procedurally re-attaching a scene graph's entire branches to other parts of a tree, or by duplicating them around the tree. These compelling examples serve as yet another tool we provide for building intuition about the effect of the order of matrix multiplications.

6 USER STUDY AND OUTCOMES

The students enrolled in multiple offerings of our *Introduction to Computer Graphics* course have reported high satisfaction with tiny-graphics.js and the resources contained in the Encyclopedia of

Table 1: Students answered the following questions with answers on a scale from zero (strongly disagree) to ten (strongly agree) by choosing integers.

Question	Median Score (0-10)	Response Rate (n)
Is the next graphical application you make likely to include tiny-graphics.js?	7	65
Did the **organization system in tiny-graphics.js** help to free your time, enough to make you reach topics that were farther in the graphics material (lecture slides, textbook) than you otherwise might have reached?	9	65
Did our **assignments based on the tiny-graphics framework** help make you reach topics that were farther in the graphics material (lecture slides, textbook) than you otherwise might have reached?	9	62
Did the **extra code examples posted by your TA (special topics demos)** help make you reach topics that were farther in the graphics material (lecture slides, textbook) than you otherwise might have reached?	8	62
Would you have rather built your assignments and project **without** tiny-graphics.js's system of pre-organizing your web application and WebGL calls for you?	1	74
Are you interested in contributing to an online tiny-graphics.js-based repository of example graphics effects?	5	61

Code. Table 1 shows glowing student responses to an anonymous questionnaire issued using Piazza.com after the completion of our Fall 2018 course. One question measured a high student interest in contributing to a potential crowd-sourced collection of educational 3D web demos. Overall, this survey shows our students had no regrets about using the resources we built for them. Otherwise our survey's power is limited; only limited conclusions can be drawn from it. As of yet we are unable to experimentally rule out other factors affecting student satisfaction, or to compare their reactions to other platforms or libraries.

A better metric comes from the instructors of our course. Citing their experience, they report that since introducing tiny-graphics.js, student term projects became more consistent; they no longer include submissions suggesting that a team is lost in the material. Figure 2 shows a sampling of student project submissions that used tiny-graphics.js. Prior to its adoption, projects always showed a high degree of effort and creativity, but were also quite limited in content. Most of our saved term projects from earlier offerings of the course showed a single scene from one or two angles. One custom shape was required, but in some cases it was as simple as a 4 sided pyramid. The most advanced topic students reached in the curriculum was texturing. The more recent projects in Figure 2 include more elaborate shapes like terrain and foxes (examples shown on the right side). The newer submissions typically included several scenes with lengthy camera movements, and more custom shapes per project. During the Fall 2018 quarter, all the team term projects that were submitted contained interactive controls, compared to just 2 out of 68 projects in a 2015 run of the course, just prior to our overhaul of the code. With tiny-graphics.js students were free to think about things like game logic and user interfaces because the low level details of graphics became easier for them, even though we did not hide these architectural workings from them.

Figure 2 also demonstrates that students reached further material within a graphics textbook. In projects students were able to show simulated physics of elastic volumes (bottom left example), reflections (center left), and custom shader effects (upper left). Custom shaders were previously unheard of in our course due to the deep modifications required in the code, but in our Fall 2018 run of the course nearly every team used custom shaders on some shapes. They used these to explore effects like bump mapping or reflection mapping, qualifying for extra credit. In projects that used tiny-graphics.js, over the years we have observed successful student implementations of advanced graphics and research techniques such as L-systems, machine learning, computer-vision-based user interfaces, marching cubes, the diamond-square algorithm, atmospheric effects such as volumetric lighting glare and shadowing, rippling water surfaces, fire, particle effects, elasticity, Newtonian physics, and spline curves. These positive observations about our term projects are our primary means of evaluating the successful outcome of our new way of organizing the graphics coding process.

7 CONCLUSIONS AND PROSPECTS

We have introduced the educational duo of tiny-graphics.js and the Encyclopedia of Code. The ease provided by tiny-graphics.js in creating organized code, simple or complex visuals, and interactivity has enabled us to create a variety of educational demonstrations for consumption by computer graphics students. The code of each example generates documentation that accompanies it online, thus resembling interactive, Literate Programs. We have paired our code library with the Encyclopedia of Code, an online platform that accepts crowd-sourced code contributions and enables anyone to generate and modify 3D graphics prototypes online with ease.

Our outcomes to date bode well for other universities and individuals who choose to adopt our library and online platform, as well as for our goal of making low level graphics coding available to the masses. In a world where anyone can read about how to use a game engine, we would like to usher in a new world where anyone can *make* a game engine themselves, in order to gain full control over visualizing anything that they can represent mathematically.

The Encyclopedia of Code remains an ongoing effort. We aim to improve its features, foster a community for 3D graphics on the web, add interactive tutorials and documentation, and crowd-source new educational articles presented as an open wiki.

For the benefit of engineering students and enthusiasts everywhere, we will advocate the expansion of our examples into an Encyclopedia of Visual Computing that covers a variety of concepts outside the traditional scope of computer graphics. Existing student-driven efforts to expand our tutorials will hopefully give way to outsider-led demos of creations on the Encyclopedia. Anyone can help create a growing repository of useful code along with beautiful, educational graphics demos.

REFERENCES

Edward Angel. 2017. The Case for Teaching Computer Graphics with WebGL: A 25-Year Perspective. *IEEE Computer Graphics and Applications* 37, 2 (2017), 106–112.

Edward Angel and Eric Haines. 2017. An interactive introduction to WebGL and three.js. In *ACM SIGGRAPH 2017 Courses*. ACM, 17.

Edward Angel and Dave Shreiner. 2014. *Interactive Computer Graphics with WebGL*. Addison-Wesley Professional.

Edward Angel and Dave Shreiner. 2016. An introduction to graphics programming using WebGL. In *ACM SIGGRAPH 2016 Courses*. ACM, 5.

Jean-Jacques Bourdin. 2016. MOOCs in computer graphics. *Proc. Eurographics:Education Papers* (2016), 49–52.

Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 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*. ACM, 1277–1286.

David Davidovi'c. 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.

Jos Dirksen. 2013. *Learning Three.js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd.

Andrew T Duchowski, Edward Angel, Bruce Gooch, and David Luebke. 2017. CGEMS: Computer graphics educational material. In *ACM SIGGRAPH 2017 Panels*. ACM, 3.

Glenn Fiedler. 2004. Fix Your Timestep! How to step your physics simulation forward. https://gafferongames.com/post/fix_your_timestep/. Accessed: 2018-08-30.

Shalini Govil-Pai. 2006. *Principles of Computer Graphics: Theory and Practice Using OpenGL and Maya®*. Vol. 190. Springer Science & Business Media.

Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R Klemmer. 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*. ACM, 241–250.

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

Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

Lauren McCarthy, Casey Reas, and Ben Fry. 2015. *Getting Started with P5.js: Making Interactive Graphics in JavaScript and Processing*. Maker Media, Inc.

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

Neil Selwyn and Stephen Gorard. 2016. Students' use of Wikipedia as an academic resource—Patterns of use and perceptions of usefulness. *The Internet and Higher Education* 28 (2016), 28–34.

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

Igor Zubrycki and Grzegorz Granosik. 2017. Teaching Robotics with Cloud Tools. In *International Conference on Robotics and Education RiE 2017*. Springer, 301–310.