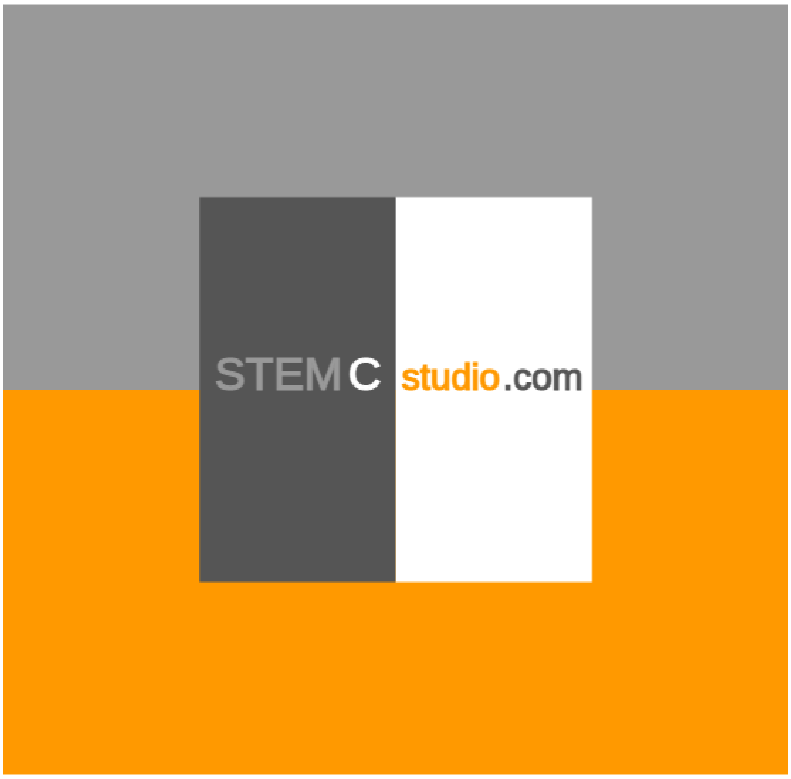
Expressing Ideas through Computational Modelling™ DRAFT EDITION

STEMCBook



How to Author Learning Experiences using STEMCstudio

David Geo Holmes

GeometryZen Press

STEMCbook

How to Author Learning Experiences using STEMCstudio

David Geo Holmes

Version 2.180.22, 2025-02-23: Draft

Table of Contents

Introduction

Foreword

STEMCstudio is a web-based educational tool for authoring STEM learning activities, such as demonstrations and assignments, and for student project-based learning of science and math through computational modeling. A STEM educational resource created using STEMCstudio may be run either standalone, embedded in another web site as an IFrame, or embedded in a Learning Management System using the LTI 1.3 protocol. Projects created with STEMCstudio are themselves web applications using modern standards-based web technologies such as HTML, JavaScript, and CSS. Using STEMCstudio, the complex software engineering tasks of building and deployment are avoided, allowing the educator and student to focus on the educational problem domain.

The *Learning Tools Interoperabilty* (LTI) version 1.3 standard defines the next generation of content creation for educational platforms by allowing external tools to be embedded securely as activities in courses. Using this technology allows applications called *tools* to be plugged into the LMS *platform* and safely interact with the LMS gradebook. STEMCstudio is one such *tool* and therefore facilitates the creation of educational resources for deployment in an LMS.

Both STEMCstudio and LTI 1.3 technologies are powerful, especially when used together, but they can be intimidating, and you may be left wondering how to get started, how they work, and how to use them effectively.

Demystifying these technologies and getting you productive is what this book is all about.

- David Holmes

Who Should Read This Book

Educational Content Authors

Modern ventures are often characterized by teams of specialists working together towards a common goal. There is every reason to believe that producing high quality educational content, just like producing a modern web site, will be enhanced by a team approach. That being said, new concepts are also pioneered by individuals who are motivated to go beyond their area of expertise in order to produce novel outcomes.

For the aforementioned reasons, this book is dually targetted at both the team producing educational content and the pioneer educator who is willing to learn the technical aspects needed to bring instruction online.

Computational Modeling Students

STEMCstudio was originally written with the student in mind. The goal of STEMCstudio was to enable Computational Modeling Experimentation for Students by lowering the Software Engineering bar. In other words, configuring an application, executing the application, and deploying it should be easy and the core task should be about the scientific model, translating it into code, and not much more. But STEMCstudio does not attempt to produce a zero-code experience which would reduce flexibility. instead, it achieves a low-code experience by being able to use standard third-party and custom libraries. Because of the attention to standards and industry best practices, it provides a high-quality developer programming experience that would be familiar to a modern front-end web developer. The programming language used by STEMCstudio is an industry standard, and so this book attempts to avoid being a programming tutorial. This book instead focuses on those unique aspects of STEMCstudio that do not have explainations elsewhere.

How To Report Bugs and Suggest Enhancements

Report issues about this book in GitHub: https://github.com/geometryzen/stemcbook/issues

About David Geo Holmes

You can contact David at david.geo.holmes@gmail.com.

Chapter 1. Getting Started

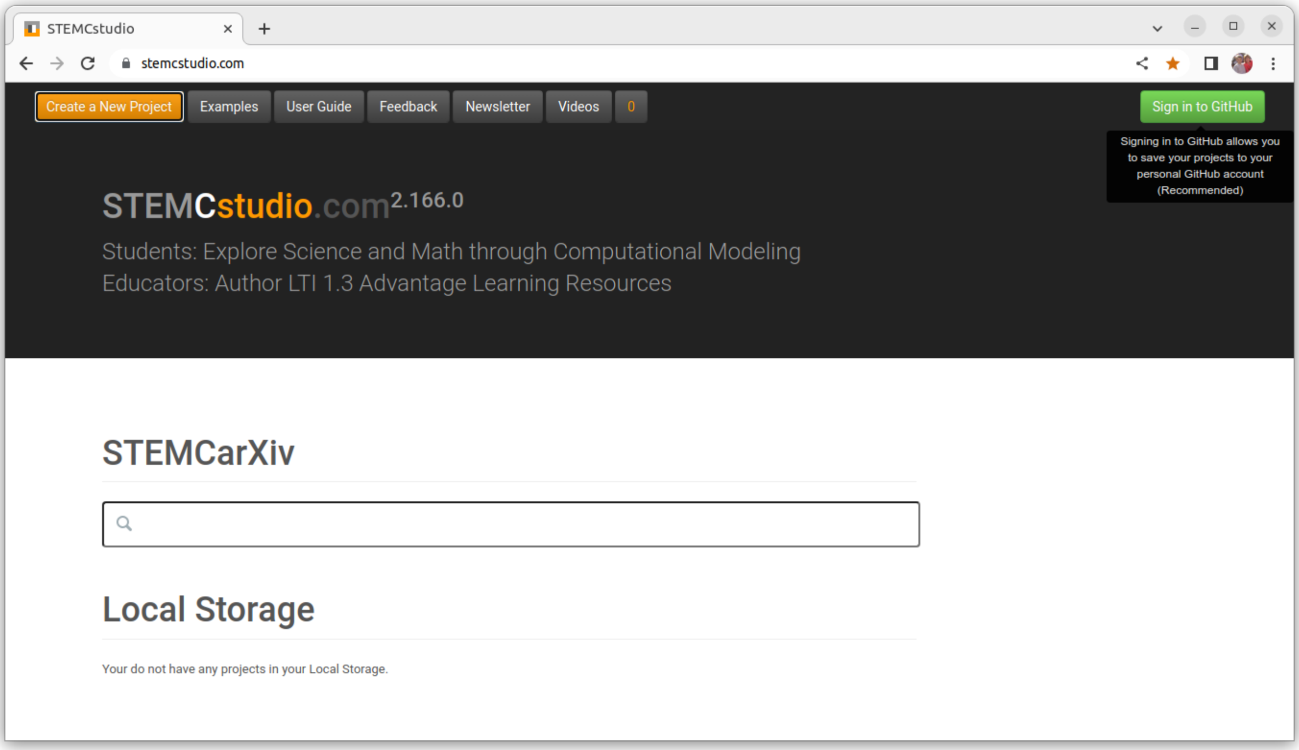
This chapter will be about getting started with STEMCstudio.

1.1. Your First Project

In this section we'll create a simple application in order to explore the essential aspects of the user interface. Our main concern will be to launch the application, save it to GitHub, and reload it in various ways.

Creating a New Project

Navigate to the STEMCstudio Home Page at *https://stemcstudio.com*



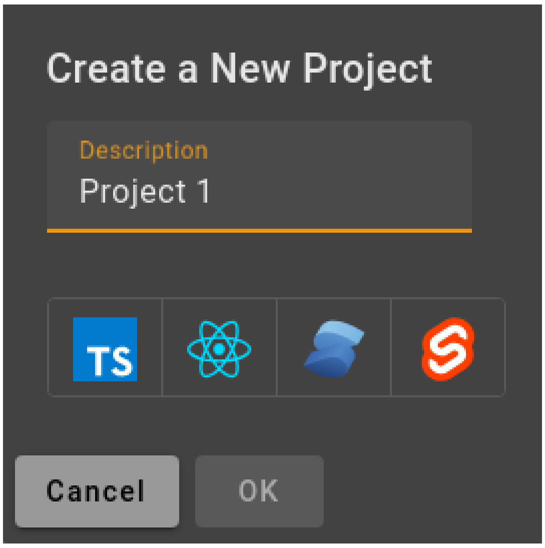
Signing in to GitHub allows you to save your projects to your personal GitHub account (Recommended)

Students: Explore Science and Math through Computational Modeling
Educators: Author LTI 1.3 Advantage Learning Resources

Your do not have any projects in your Local Storage.

Figure 1. STEMCstudio Home Page

Press the *Create a New Project* button to begin creating a new STEMCstudio project.



Create a New Project

Description: Project 1

Figure 2. STEMCstudio New Project Dialog

Enter a description for your project, e.g. `My First Project`. The description is not critical but it may make your project more searchable. It can also be changed later.

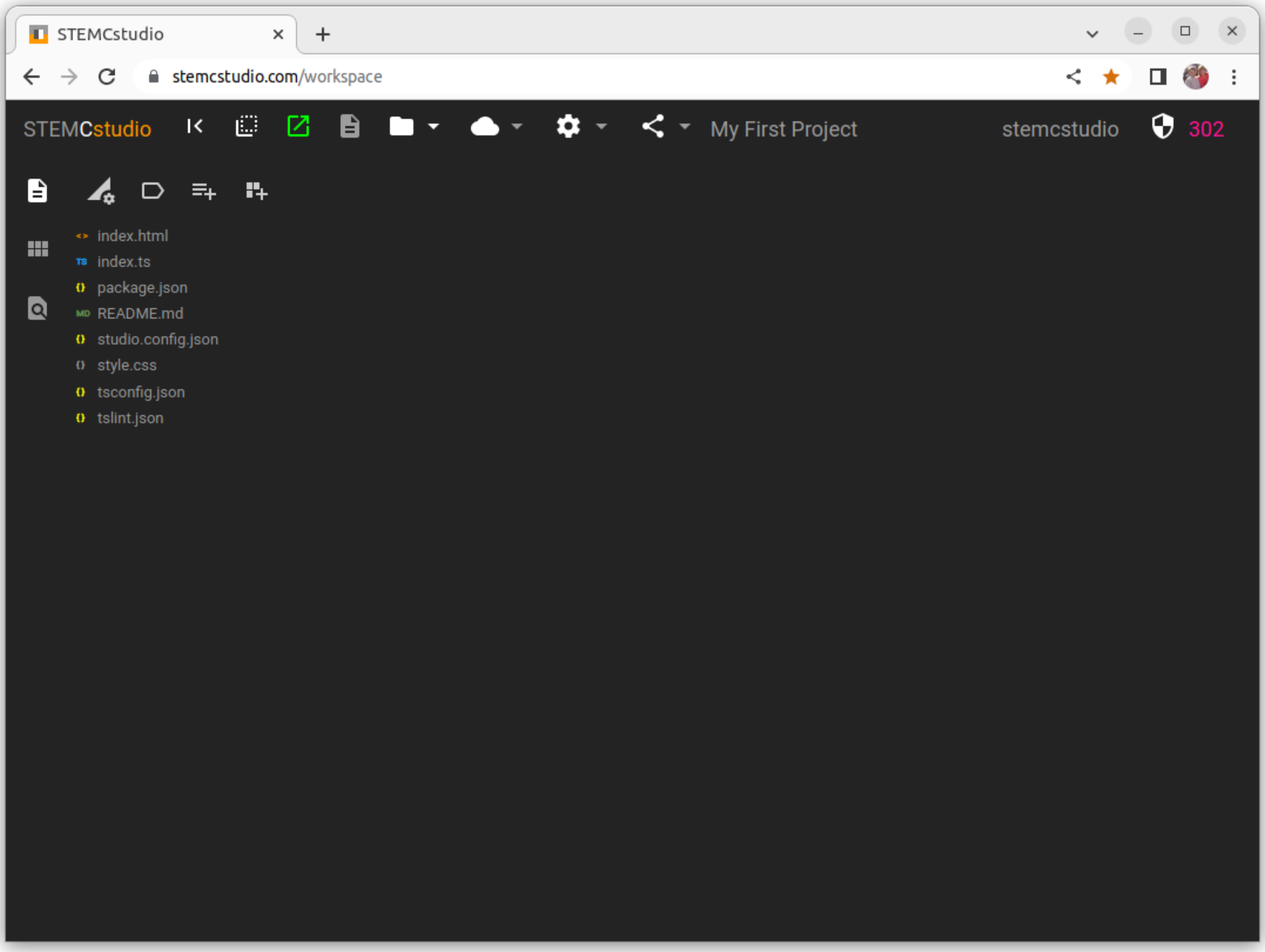
Choose a *template* for your application, that is to say, a working project that will be cloned to make your new project.

Several *templates* are available for popular application frameworks as well as a minimal project using just HTML and TypeScript:

Icon	Name	Home Page
TS	TypeScript	https://typescriptlang.org
	React	https://react.dev
	SolidJS	https://solidjs.com
	Svelte	https://svelte.dev

To keep things simple, for this example, we'll choose TypeScript to create an application that does not use a framework, then press `OK`.

You will land on the STEMCstudio Workspace page with your new project files listed in the *Explorer View*.



index.html
index.ts
package.json
README.md
studio.config.json
style.css
tsconfig.json
tslint.json

Figure 3. STEMCstudio Workspace

The following guide is provided as a reference to the controls in the *Workspace Window*.

Horizontally, across the top or the *Workspace Window* is the *Main Toolbar*:

|< (*Hide Explorer*) or >| (*Show Explorer*) - toggles the visibility of the explorer view.

⬚ (*Hide Editors*) or ⧉ *(Show Editors)* - toggles the visibility of the editors and the explorer view.

↗ *(Launch Program)*, ■ (*Stop Program*) - toggles the execution of the program.

📄 (Hide Documentation) or (Show Documentation) - toggles the visibility of the rendering of the *README.md* file.

📁 (Project Menu) - container for the project dropdown menu.

☁ (Cloud Menu) - container for the cloud dropdown menu.

⚙ (Workspace Settings) - container for the workspace dropdown menu.

(Share Menu) - container for the share dropdown menu.

Vertically, down the side of the *Workspace Window*:

(Files) - Explorer mode showing files.

(Dependencies) - Explorer mode showing dependencies.

(Usages) - Explorer mode showing usages of symbols in the code.

Horizontally, across the top of the *Explorer View*:

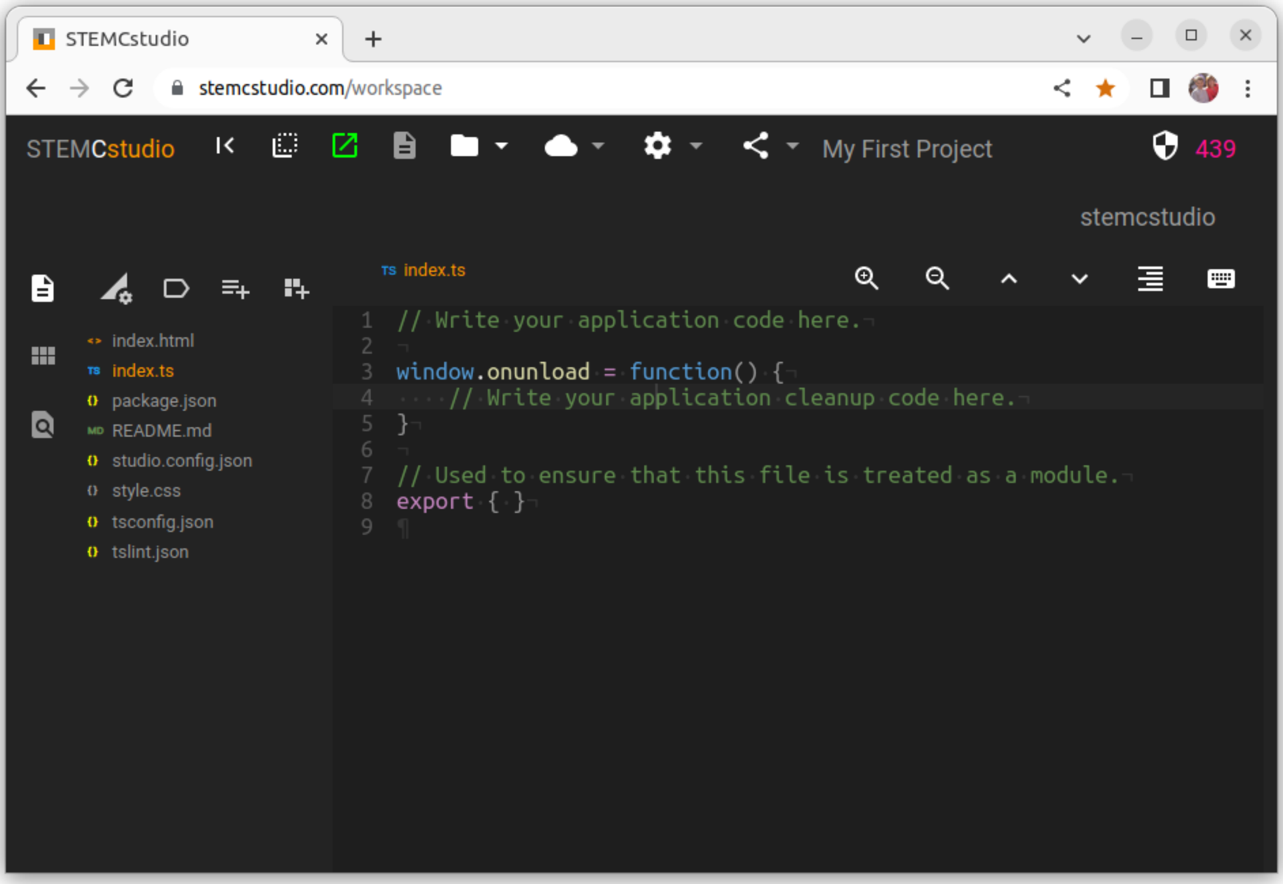
(Project Settings) - Used to configure the project build and linting

(Labels and Tags) - Used to configure meta-data used for searching.

(Add File) - Used to add a new file to the project.

(Add Dependency) - Used to add an external package dependency to the project.

Click on the *index.ts* file in the *Explorer View* to open the corresponding editor.



```
// Write your application code here.

window.onunload = function() {
    // Write your application cleanup code here.
}

// Used to ensure that this file is treated as a module.
export { }

```

Figure 4. STEMCstudio Editor

The editor window contains some convenient functionality using the icons in the editor toolbar:

🔍 (Increase Font Size)

🔍 (Decrease Font Size)

^ (Fold)

⌄ (Unfold)

☰ (Format Document)

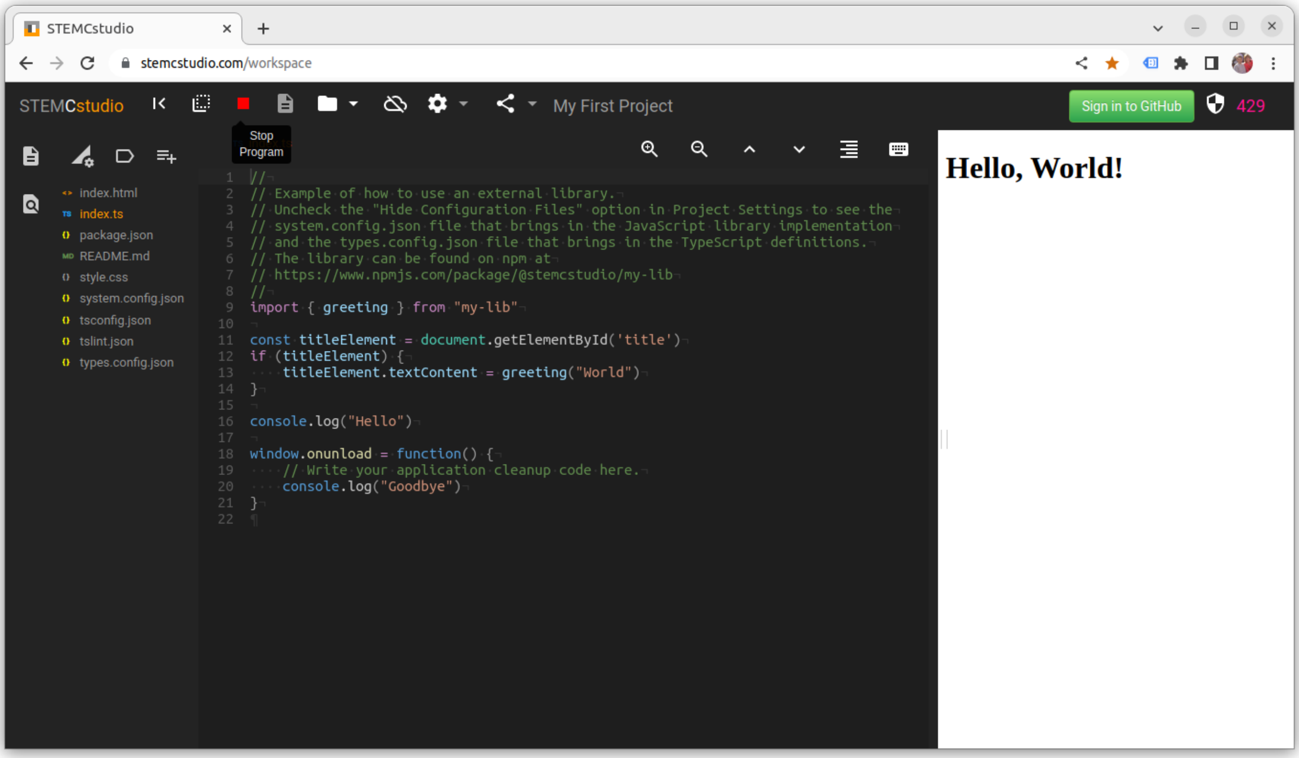
⌨ (Keyboard Shortcuts)

Feel free to explore the workspace, the buttons, and menus. I won't cover every option here, most of them will be familiar if you have used a modern IDE.

Launching the Project

Execute the program by pressing the `Launch Program` button, ↗, on the main toolbar.

The program web page will run side-by-side with the code.



```
//
// Example of how to use an external library.
// Uncheck the "Hide Configuration Files" option in Project Settings to see the
// system.config.json file that brings in the JavaScript library implementation
// and the types.config.json file that brings in the TypeScript definitions.
// The library can be found on npm at
// https://www.npmjs.com/package/@stemcstudio/my-lib
//
import { greeting } from "my-lib"

const titleElement = document.getElementById('title')
if (titleElement) {
    titleElement.textContent = greeting("World")
}

console.log("Hello")

window.onunload = function() {
    // Write your application cleanup code here.
    console.log("Goodbye")
}
```

Hello, World!

Figure 5. STEMCstudio Live Coding

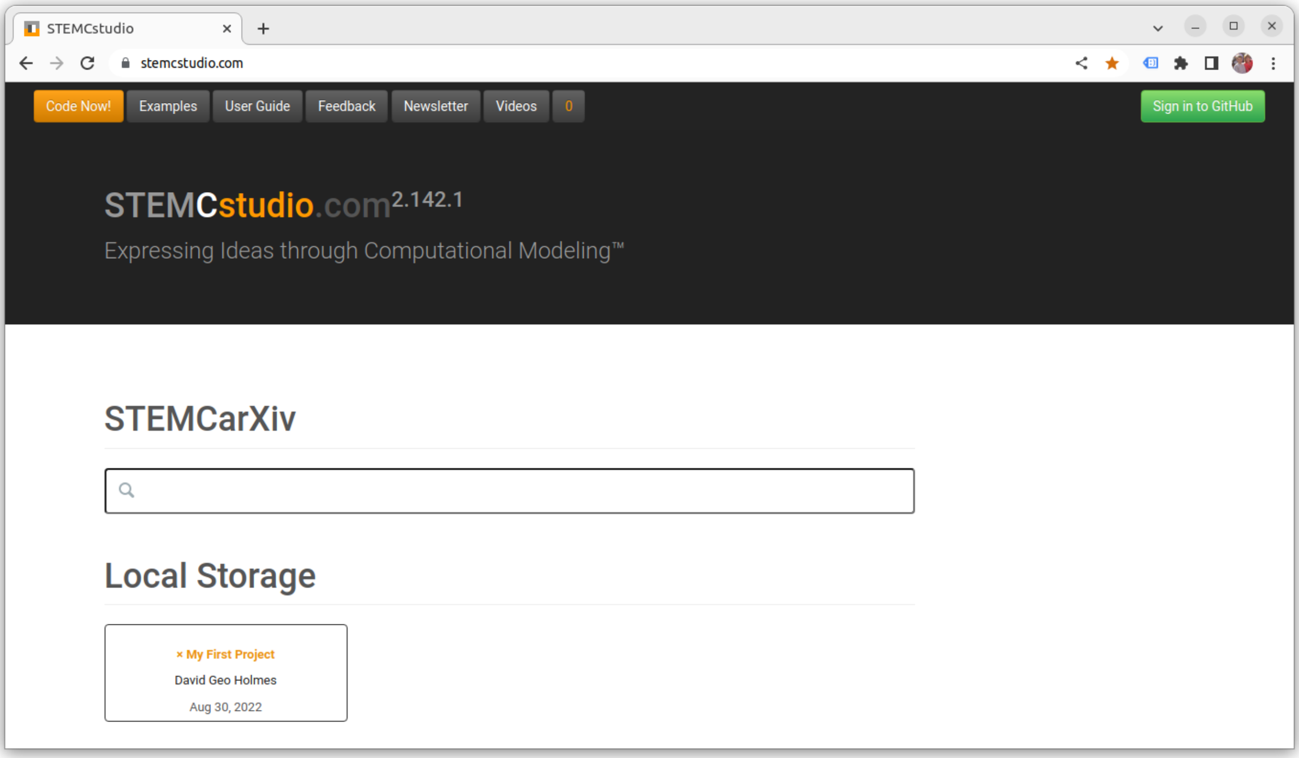
Change the greeting name parameter to "Your Name". Notice that any changes you make to the editor code are reflected almost immediately in the Live Code View.

End the program by pressing the `Stop Program` button, ■ .

Now return to the Home Page by clicking STEMCstudio brand icon, STEMCstudio.

Loading the Project from Local Storage

If you have been following along, you will now be on the STEMCstudio Home Page with your project details displayed under `Local Storage`.



× My First Project
David Geo Holmes
Aug 30, 2022

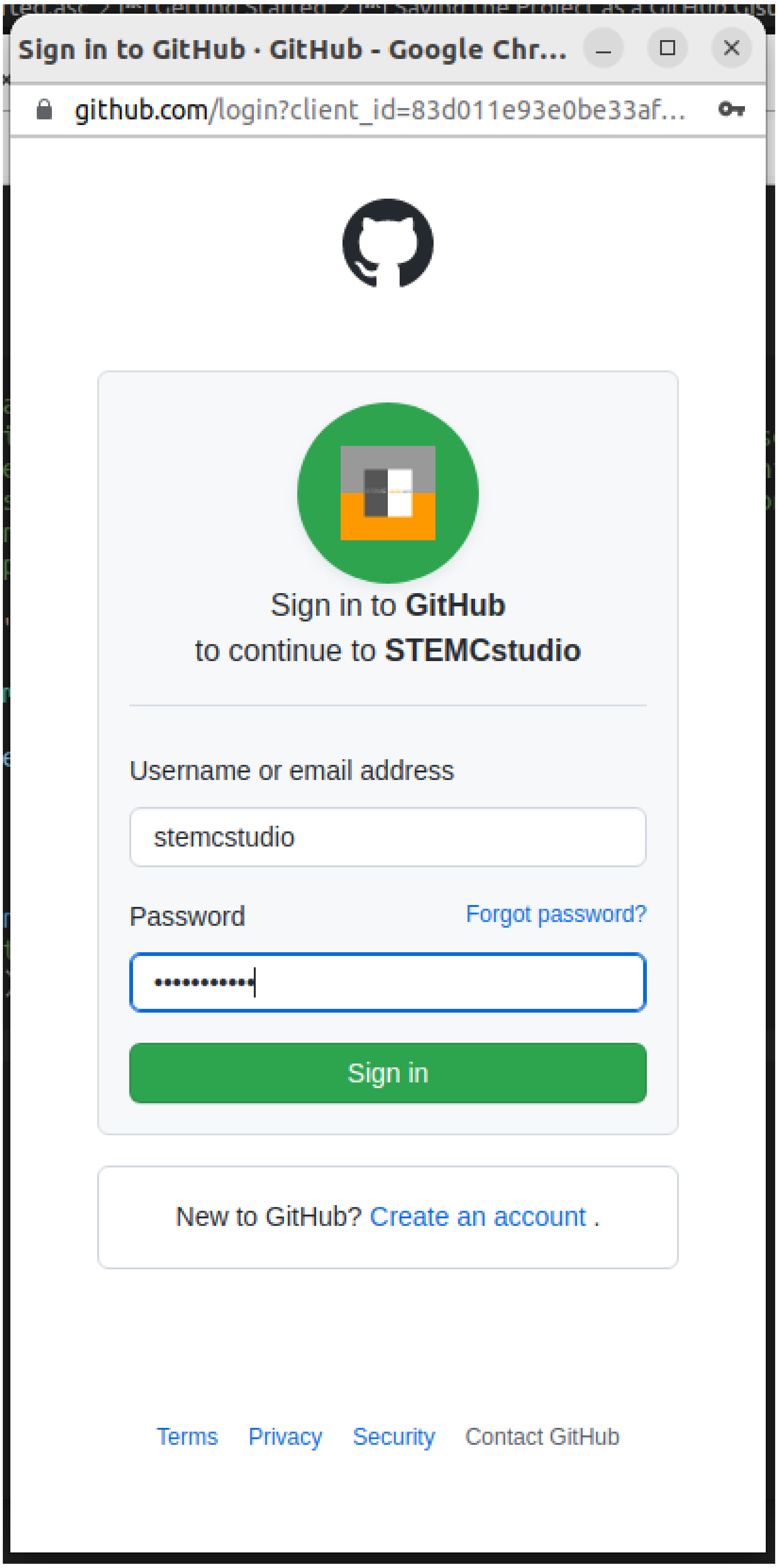
Figure 6. STEMCstudio Project in Local Storage

Your project is now stored in your current browser on your current machine and nowhere else. That's fine for making changes locally, but you will want to save your work permanently. STEMCstudio allows you to save your work to a GitHub Gist in your GitHub account. If you don't have a GitHub account, now is a good time to sign up at *https://github.com*.

Saving the Project as a GitHub Gist

Click the description of your project in Local Storage to re-open your project in the workspace. Notice that the location URL in your browser is simply `stemcstudio/workspace`. Press the `Sign in to GitHub` button, Sign in to GitHub.

You will be presented with a popup page from GitHub that allows you to sign in.



Sign in to GitHub · GitHub - Google Chr...

github.com/login?client_id=83d011e93e0be33af...

Sign in to **GitHub**
to continue to **STEMCstudio**

Username or email address

stemcstudio

Password

Forgot password?

••••••••••••

Sign in

New to GitHub? Create an account .

Terms Privacy Security Contact GitHub

Figure 7. STEMCstudio GitHub Sign In Page

Enter your GitHub Username and Password and press the `Sign in` button.

You will be redirected back to STEMCstudio. To save your project as a Gist, click the `Upload Project to GitHub` menu item under the cloud menu, ☁. Your project will be saved, a new Gist identifier will be assigned, and STEMCstudio will reload the project. Notice that the browser URL is now *stemcstudio.com/gists/your-gist-identifier*

Anytime that you wish to save your project to GitHub after making changes, simply ensure that you are signed in and upload it.

Loading the Project from GitHub

Return from the workspace back to the STEMCstudio Home Page. Your project is in Local Storage but it is also saved in GitHub.

> ⚠ Whenever your project is in Local Storage, STEMCstudio will load it from there and not from GitHub. In other words, the copy in Local Storage is your working copy. Be careful not to lose your changes if you are working on more than one computer.

Delete the copy in Local Storage by clicking the `x` symbol next to the project description.

To retrieve your project from GitHub, click the `Download` button on the Home Page.

> ℹ The `Download` button on the Home Page will only be be visible if you are signed in to GitHub.

Click on the description of the project that you wish to download. You will be returned to the workspace with your project loaded.

Publishing and Finding the Project using the STEMCarXiv

STEMCstudio allows you to index your project for searching in a public archive. The archive name is STEMCarXiv. Adding your project to the STEMCarXiv index is easy. Open your project in the workspace. Ensure that you are signed in to GitHub to enable the cloud menu. Select the `Publish Project to STEMCarXiv` option under the cloud menu. Your project will be indexed based upon meta data such as its description and keywords.

To search for your program in the STEMCarXiv, use the search box on the Home Page.

Embedding the Project in a Web Page

Your project may be embedded in a web page. Use the `Embedding Builder` menu item under the `Share Menu` in the workspace view to build the HTML Embed String.

Running the Project in STEMCviewer

STEMCstudio has an execute-only companion application called STEMCviewer. This can run your application without loading the design-time parts of STEMCstudio. Simply change the URL in your browser to *stemcviewer.com/gists/your-gist-identifier*.

> ℹ For this to work you must have executed your program and uploaded it to GitHub. The explaination can be found in the *How it Works* section of this book.

1.2. How It Works

Understanding how STEMCstudio works will enable you to consume external libraries, author external libraries for consumption by STEMCstudio, optimize performance, and resolve issues that inevitably arise.

Minimal Application Example

We’ll use, as an example, an application that results from using the minimal template to create a new project. If you have access to your computer, create such a project and follow along as we walk through it. Make changes to check your understanding of what is going on.

Every web application has, at its core, an HTML file that is loaded by the browser. The source code for the HTML file, *index.html*, in this example is rather unremarkable and doesn’t look like it will do anything dynamic.

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <base href="/">
    <title></title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <h1 id='title'>Hello, World!</h1>
</body>

</html>
```

About all we can surmise from this file is that it is going to bring in the *style.css* from the project.

Let’s take a look at the *index.ts* file.

```typescript
// Write your application code here.

window.onunload = function() {
    // Write your application cleanup code here.
}

// Used to ensure that this file is treated as a module.
export { }
```

This certainly looks like the code that produces the behavior of the application. But how does this file get loaded?

There's an underlying rule in STEMCstudio that a file called *foo.html* will try to load a file called *foo.js*. If you know anything about TypeScript it is that browsers execute JavaScript and there exists a transpiler that can convert a TypeScript *foo.ts* to JavaScript *foo.js*. Every time you make a change to your TypeScript files, STEMCstudio will be transpiling them to JavaScript in the background.

So now we need to know how all of this gets loaded into the browser. STEMCstudio is a little different from other live coding environments in that *all* the processing is taking place in the browser. While code may be loaded from elsewhere on the internet such as CDNs, there is no code execution elsewhere. This is by design. If all the processing takes place in the browser then we avoid the problems of scaling with the number of users which would otherwise require us to spin up costly virtual machines in the cloud.

You may also know that web development often requires you to run a local web server because the browser isn't allowed to load resources using the file protocol. So how does STEMCstudio do it? The solution is that STEMCstudio converts all JavaScript files to an industry standard module format called *system* and then bundles all the resources it needs, including few extra bootstrapping scripts, into a single string that the browser can load. It loads the bundled code into an *iframe* HTML element. This is all happening in memory so it's not obvious how to see directly what is going on. However, there are two things we can do to peek behind the curtain!

One approach is to open the developer console in your browser and inspect the elements. You should be able to find something like this:

```
<iframe id="...">
#document
    <html lang="en">...</html>
</iframe>
```

Another approach is to know that STEMCstudio creates a cache of this bundled content and stores it in a hidden file in your project called *generated.index.html*. So if you save your project to a GitHub Gist (after it has been launched) then you can inspect this file in GitHub. You can also view it in STEMCstudio by checking the *Show Generated Files* option under the *Project Settings* menu. Here's what it looks like in this example:

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <base href="/">
    <title></title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css">
    <style>
        body {
            background: #ffffff;
            font-family: sans-serif;
            margin: 8px;
        }
    </style>
    <script src="/assets/js/stemcstudio-systemjs@1.0.0/system.js"></script>
</head>

<body>
    <script>
        System.config({
          "warnings": false,
          "map": {}
        });
    </script>
    <h1 id="title">Hello, World!</h1>


    <script>
        System.register("./index.js", [], function (exports_1, context_1) {
            "use strict";
            var __moduleName = context_1 && context_1.id;
            return {
                setters: [],
                execute: function () {
                    window.onunload = function () {
                    };
                }
            };
        });
        //# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJmaWxlIjoiaW5kZXguanMiLCJzb3VyY2VSb290IjoiIiwic291cmNlcyI6WyJpbmRleC5qcyJdLCJuYW1lcyI6W10sIm1hcHBpbmdzIjoiOzs7Ozs7WUFBQSxNQUFNLENBQUMsUUFBUSxHQUFHO1lBQ2xCLENBQUMsQ0FBQztRQ0FBQztRQUVGLENBQUMifQ==
    </script>
    <script>
        System.defaultJSExtensions = true
        System.import('./index.js').catch(function(e) { console.error(e) })
    </script>
</body>

</html>
```

The first thing to notice is that STEMCstudio has added a *script* to the *head* element that loads a JavaScript file called *system.js*. The *system.js* file creates a global object called `System` which is known as the module loader.

The second thing to notice is that the *index.ts* file has been transpiled and wrapped in a `System.register` function call. This makes the transpiled contents of *index.ts* known to the `System` module loader.

Further down you will see a `System.import` function call. This invocation executes the module that is referred to by name.

And that's the essence of how it works in the most simple case. More complicated multi JavaScript file examples are handled automatically.

Cascading Style Sheets

Notice that the contents of the project *style.css* file have been inlined as a *style* tag in *generated.index.html* at exactly the point in *index.html* where there was a *link* tag.

1.3. Writing and Calling Internal Modules

In STEMCstudio, modules are files in your project that export JavaScript resources such as classes, functions and variables, and TypeScript-specific resources such as types and interfaces.

Modules provide a way to organize your code into separate concerns while controlling what you expose to other modules. This is important for ensuring the maintainability of your code as it grows in complexity.

STEMCstudio modules follow the ECMAScript ES6 modules specification exactly, so I won't repeat the specification or the many good tutorials on the web. The only thing that you do need to know is how STEMCstudio handles module names and a small thing to be aware of concerning the defaulting of JavaScript extensons.

Module Names

In STEMCstudio you will be writing your code as TypeScript and your files will have the *.ts* extension. But the browser runtime uses JavaScript files, so we need to known how a TypeScript file that contains the source code for a module is named as a JavaScript module.

In STEMCstudio, a file called *foo.ts* that is the source for a module becomes a JavaScript file with the module name `./foo.js`.

Incidentally, any file that uses the `export` keyword is treated as a module.

Suppose that you have a file called *foo.ts* that exports a function:

foo.ts

```
export function greeting(name: string): string {
    return `Hello, ${name}!`
}
```

To import the function from the file *foo.ts* you would use the *import specification* syntax:

```
import { greeting } from "./foo.js"
```

If you press the keys *Ctrl+Spacebar* while the cursor is within the curly braces of your import specification, STEMCstudio will provide a pick list of available remaining imports.

Defaulting of JavaScript Extensions

If you do not provide the JavaScript extension for the internal module name, it **MAY** work, but is not guaranteed to do so in future.

For example, suppose you import using the syntax:

```
import { greeting } from "./foo"
```

The legacy behavior of STEMCstudio is to default a JavaScript extension if one is not provided. You will not get an error from the STEMCstudio IDE at design time.

1.4. Using NPM Packages (a.k.a. External Modules or Libraries)

The ability to consume external modules in the form of NPM packages is one of the more powerful features of STEMCstudio. An external library is JavaScript code that exists outside of your STEMCstudio project and yet can be loaded and called from your STEMCstudio project. External libraries often provide coarse-grained functionality such as diagramming or plotting. External libraries are an effective way to reuse functionality, dramatically improving your productivity. Additionally, STEMCstudio can consume external libraries without the need to for server-side computing resources. This allows applications to be launched with minimum time to become usable, and ensures scaling to large numbers of users. As with any flexible system, there is an attendant increase in complexity of understanding and/or effort of implementation. The goal of this section is to enable you to consume external libraries reliably and efficiently. The process for consuming a package is systematic and therefore quite simple, though in some cases slightly laborious.

Our approach for reaching the goal of consuming a library is as follows:

- Understand the general requirements for STEMCstudio interoperability.
- Know how to research the suitability of an external library.
- Be able to choose the correct implementation approach for a given library.
- Include the library as a dependency in a STEMCstudio project.

This approach uses standard software engineering techniques and the result is reliable and performant. In the following sections we discuss the parts of this approach in detail.

Library Requirements for Interoperability with STEMCstudio

STEMCstudio is open to use any external library provided that library meets some particular requirements. These requirements are mostly industry standards, but some additional requirements are specific to STEMCstudio. When you author your own library, it takes little extra effort to ensure that your library works seamlessly and efficiently with STEMCstudio. However, with third party libraries, and when these requirements are not met, it is always possible to create wrapper libraries that expose the required functionality suitable for consumption by STEMCstudio.

The requirements on a library support the design-time and runtime usage of the library. Let’s look at each of these.

Designtime Requirements

Designtime refers to the editing process where we would like STEMCstudio to assist us with the correct usage of the library.

STEMCstudio is able to offer editor support for an external library (package) when the TypeScript type definitions ('*.d.ts" files) for the package exist and are locatable. In the ideal case, the type definitions are maintained by the package authors in order to keep them synchronized with the implementation code, the definitions are co-located with the package on a CDN, and the 'package.json' file for the package provides the relative location of the type definitions. Increasingly, as package authors use TypeScript as their source language, the type definitions are built along with the source code, and the location is described in the 'package.json' file. However, there are examples of some popular libraries where type definitions are independently and manually maintained in a separate package, and some cases where they do not exist at all. There are solutions for all these cases that fill in the missing pieces and allow STEMCstudio to use almost any package.

Runtime Requirements

Runtime refers to the execution of a program and may take place while STEMCstudio is working in interactive mode or when STEMCviewer is presenting the working program.

STEMCstudio performs most efficiently when a library is in the 'system' module format. The 'system' module format is used by STEMCstudio because it allows module loading to be performed entirely in the browser. While the 'system' module format is an industry standard (and an output of the TypeScript compiler), popular libraries that directly implement this module format are rare. However, if a package does not implement the 'system' format you are not always required to implement a wrapper to perform the conversion. For eample, a well known module format that can be consumed directly, and almost as efficiently, by STEMCstudio is the Universal Module Definition (UMD). The operation with UMD is possible because the conversion to 'system' format, which happens at runtime, is relatively efficient. The UMD format, while common, is an unofficial legacy format. The industry is evolving towards use of the EcmaScript module format (ESM). The ESM format can also be transpiled at runtime to the 'system' format but at present this requires the use of the TypeScript transpiler and a hefty (approx 10MB) download. For this reason, it is preferable to convert ESM to system format by wrapping the original library. Whatever the case, there are solutions to make your intended third-party library available to STEMCstudio, and we will cover them in this document.

Chapter 7 provides concrete help and best practices for authoring a JavaScript library optimized for STEMCstudio.

Researching a Package for use in STEMCstudio

The first step to incorporate an external library is finding out where it is deployed and how the various library artifacts are stored in that deployment. We'll use the term Content Delivery Network (CDN) for a server that stores library artifacts and makes them available over the web using a URL.

The de-facto standard for making a JavaScript library available for widespread consumption is for the author to publish it to `npm`, the Node Package Manager. Don't be misled by the name, many packages are defined in `npm` even if there is no intention of using the library in Node.JS.

If the library has not been published to `npm`, you will need to contact the library author to get details about the library for consuming it. In this case your solution will likely include downloading the library and wrapping it in your own package that you publish to npm.

We'll assume that the library that you intend to use has been published to `npm`, and that you know the package name.

The information that we need about a library includes its name, available versions, available JavaScript module formats, the URL paths to the various JavaScript implementations and the URL for the TypeScript definitions.

Inspecting a Package using online tools

There are two online websites that may be used to inspect an `npm` package. One is the `npm` repository website itself, `https://www.npmjs.com`. Another is the popular 'jsdelivr' CDN, `https://www.jsdelivr.com` website. Both of these websites allow you to search for a package, inspect the available versions, and even browse the file structure and file contents of the available resources.

We'll use the `npm` repository website here because it is the point of contact for the library author, but you should also visit the jsdelivr website because you will need a CDN, and the URLs for retrieving code are CDN specific.

An alternative to the jsdelivr CDN is `https://unpkg.com`. This also contains packages that are published to 'npm'. However, I have found 'jsdelivr.com' to be more reliable than 'unpkg.com'. jsdelivr uses an extensive network of servers making it suitable for production use, whereas unpkg has far less hardware and is more suitable for prototyping.

Navigating to `https://www.npmjs.com` in a web browser will bring you to the `npm` search page. Enter the desired package name, or other known descriptive details, to find the package. The author may have provided the information you require in human-readable text on the package home page. Another way you can find the information is to inspect the contents of the `package.json` file that accompanies every library that is published to npm.

Modern and Legacy package.json properties.

With the advent of ESM, the specification for how the 'package.json' file should describe the location of file resources has become more sophisticated and slightly fragmented. One example of fragemtation is that there exists a "modern" specification based upon the 'exports' JSON property and a "legacy" specification that uses a number of JSON properties. Some packages will use both approaches and consider the "legacy" approach to be a fallback mechanism. Another fragmentation example is that runtime behavior is specified in the `Node.JS` specification, but this specification makes no mention of how designtime support is provided and we must resort to finding a standard that defines the location of TypeScript type definitions. Finally, module formats beyond 'ESM' are not covered for the 'exports' property and so there are no guidelines for the 'system' module format. The resolution of this fragmentation is that different interests have supplied their own specifications. Fortunately, the core `exports` specification is quite flexible and there is plenty of room for the different interests to coexist.

> This document will not attempt to reproduce the official specification, which is located here 'https://nodejs.org/api/packages.html#package-entry-points'. The TypeScript authors offer some guidelines at https://www.typescriptlang.org/docs/handbook/declaration-files/publishing.html, but this only appears to cover the "legacy" case.

Ideal package.json file for consumption by STEMCstudio

Before looking at the various ways to prepare a library for consumption by STEMCstudio it will be helpful to consider the ideal case of a package specifically engineered to work in STEMCstudio.

Consider the following fragment of a `package.json` file distributed with a package that was custom built to be used in STEMCstudio:

```
    "exports": {
        ".": {
            "types": "./dist/index.d.ts",
            "import": "./dist/esm/index.js",
            "require": "./dist/commonjs/index.js",
            "system": "./dist/system/index.min.js",
            "default": "./dist/esm/index.js"
        }
    },
```

Take a look at the `exports` property first and the descendant property `default`. The corresponding property value is "./dist/esm/index.js". The value is the location of the ESM module implementation relative to the `package.json` file. This is in accordance with the `Node.JS` specification.

> I won’t fully explain the properties that can exist underneath the `exports` property, nor how they work. You should refer to the `Node.JS` documentation. For our purpose it is sufficient to know that the properties under `export` can describe paths that provide a means to filter the available resources and that the period in this

> case refers to the top-level module.

Now look at the types property that is a sibling to the default property. The value of this property is "./dist/index.d.ts" and defines the relative location of the TypeScript type definitions. The types property is important to STEMCstudio because it enables design time editing support.

The remaining system sibling property is a STEMCstudio standard. It describes the entry point for the runtime in system module format.

Finally, the module and types properties that are sibling to exports property are the "legacy" mechanism for locating resources. STEMCstudio will only use these if it cannot find the exports property.

Consuming an Ideal package in STEMCstudio

Now that you understand what constitutes an ideal package for STEMCstudio, we are ready to configure STEMCstudio to consume the package.

For this example, we are going to use the package @geometryzen/my-lib, which is the library example described in the appendix.

The appendix describes how to consume this library.

When the third-party package is less than Ideal for STEMCstudio

If a third-party package does not support the system format it may still be possible to efficiently consume the library if a UMD format exists. This is done by placing an override in your project *system.config.json* file. The following example shows the overrides for consuming the react and react-dom packages.

studio.config.json

```
    "overrides": [
        {
            "name": "csstype",
            "version": "3.1.2",
            "system": "https://cdn.jsdelivr.net/npm/csstype@3.1.2/package.json",
            "types": "https://cdn.jsdelivr.net/npm/csstype@3.1.2/package.json"
        },
        {
            "name": "prop-types",
            "version": "15.8.1",
            "system": "https://cdn.jsdelivr.net/npm/prop-types@15.8.1/umd/prop-types.js",
            "types": "https://cdn.jsdelivr.net/npm/@types/prop-types@15.7.5/package.json"
        },
        {
            "name": "react",
            "version": "18.2.0",
            "system": "https://cdn.jsdelivr.net/npm/react@18.2.0/umd/react.development.js",
            "types": "https://cdn.jsdelivr.net/npm/@types/react@18.0.28/package.json"
        },
        {
            "name": "react-dom",
            "version": "18.2.0",
            "system": "https://cdn.jsdelivr.net/npm/react-dom@18.2.0/umd/react-dom-development.js",
            "types": "https://cdn.jsdelivr.net/npm/@types/react-dom@18.0.10/package.json"
        }
    ],
    "references": {},
```

Ignore the `csstype` and `prop-types` overrides for now and focus on the `react` and/or `react-dom` override. We can see that there is an override for the runtime behavior by having the "system" property reference a UMD module. We're also getting the TypeScript type definitions through the `@types` organization.

How did we know that these overrides are needed? This can only be done by inspecting the distributed artifacts. Some Software Engineering detective work is needed to discover the alternatives. Once plausible alternatives have been found, the required override can be created and tested.

Consuming EcmaScript Modules in STEMCstudio

STEMCstudio simulates the browser ES module loader by transpiling ESM code into the `System` format. In the Live Coding Editor it does this on-the-fly for TypeScript code in your project. STEMCstudio can also transpile external libraries. However, there is a catch. For external libraries in ESM format, the `System` loader needs to transpile the code before it can be executed. This in turn requires loading the `typescript.js` file as an ordinary script in your HTML file:

```
<script src='https://stemcstudio.com/vendor/typescript@5.0.0/typescript.js'></script>
```

You can try the application at the following URL:

https://www.stemcviewer.com/gists/fee3eb03cf64db9fd3ee963875a656ca

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/fee3eb03cf64db9fd3ee963875a656ca

But `typescript.js` is a large (approx 10MB) file so this can be slow and can seriously affect the load time of your application. Using the `async` or `defer` attributes to control the load either does not help with performance or does not solve the problem. Additionally, transpiling the external module on-the-fly may not work if the the library is not bundled into a single file.

An alternative approach, described in the Appendix, is to convert ESM-only libraries into `System` format up-front. This is highly recommended for production applications. It can also solve the problem of bundling external module files.

For prototyping, it may be acceptable to convert the ESM module at execution time. To use an external library in the `esm` module format you will need to make TypeScript available to the `System` loader using the script tag above.

Best Practices for Maintainability

The ideal way to consume a library in STEMCstudio is for the library to meet all the requirements for avoiding overrides:

- The library has a bundled `system` module format.
- The `system` module format can be located using the "exports" property in the library *package.json* file.
- The library has TypeScript type definitions.
- The *index.d.ts* file can be located using the "exports" property in the library *package.json* file.

If these conditions are not met then it should be possible to create a package, owned and versioned by your own organization, that exposes the third-party library in the ideal form. See the appendix for more details.

1.5. Summary

In this chapter we have made a start by creating a project in STEMCstudio. We have seen how it works behind the scenes, how to scale up to larger applications by using internal modules and external libraries, and how to save and find your work.

Chapter 2. Learning Tools Interoperability

In the chapter we will understand the LTI 1.3 standard in relation to Learning Management Systems and how STEMCstudio can provide dynamic content for an LMS. Additionally, we will be looking at the programming API for interacting with the LMS as well as the process for registering STEMCstudio with the LMS and the process for authoring course activities.

2.1. What is Learning Tools Interoperability?

Learning Tools Interoperability (LTI) is a standard that defines how a web application such as a Learning Management System can host and interact with other web applications that provide learning activities and content for courses.

https://www.1edtech.org/standards/lti

Terminology

In the LTI model, the hosting web application is generically known as a *platform* and the hosted web application is known as a *tool*. The *platform* and *tool* communicate through secure web network interactions. The initial registration of a *tool* with a *platform* results in each party exchanging public keys for secure communication.

Background

The latest version of LTI is currently version 1.3 and is managed by the IMS Global consortium.

> ℹ The version 1.3 has caused some confusion because it is not backwardly compatible with the LTI 1.0 specification, and the 2.0 specification was abandoned being replaced by 1.3; a case of mixing marketing names with semantic version numbers! Other than the initial confusion, this should not pose any technical problems.

Capabilities

The 1.3 version of the standard mainly defines how a *platform* can link to the content provided by a *tool*, and how a *tool* can interact with the *Gradebook* of the platform to submit scores, retrieve results, and add new *Gradebook* items. A separate specification called *LTI Advantage* defines how the registration process can occur in a way that hides all the complexity of exhanging security keys and othe meta information that must be exchanged between the *platform and _tool*.

Benefits

LTI is a dramatic improvement over the traditional *plugin* model that requires a plugin author to provide code that is installed on the host LMS. The following table offers an at-a-glance feature and benefit comparision of the two different architectures.

Feature	STEMCstudio+LTI	traditional plugin	Benefit
Modules	ES6	No	Scalability
Libraries	Open	Closed	Extensibility
Importing	Modular	Global Variables	Safety
Versioning	Yes	No	Reliability
Smart IDE	Yes	No	Productivity

2.2. Dynamic Registration

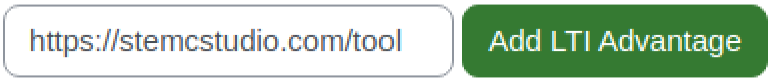
Dynamic Registration is an automated process for establishing secure communication parameters for the tool (STEMCstudio or STEMCviewer in this case) and the platform (e.g. Moodle, Canvas, Blackboard, Sakai). Once registration is complete, the tool can be launched securely using the identity of the user logged into the LMS.

The URL to use for dynamically registering STEMCstudio is `https://stemcstudio.com/tool`.

> You will use STEMCstudio for the creation of learning resources. STEMCstudio provides both the design-time environment for resource creation as well as the run-time environment for executing and previewing your creations. However, if all you want to do is to execute your creations either standalone or in a Learning Management System then it is more efficient to execute them using STEMCviewer. Not only will STEMCview load faster, but it will also prevent access to the source code for your project.

The URL to use for dynamically registering STEMCviewer is `https://stemcviewer.com/tool`.

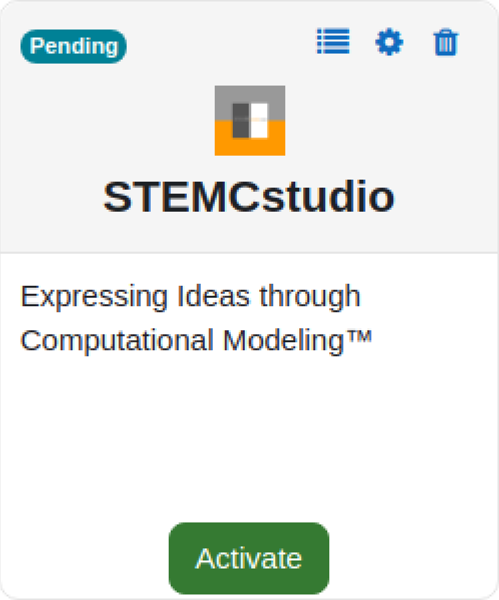
The following image depicts the registration using the Moodle LMS.



https://stemcstudio.com/tool

Figure 8. Registering STEMCstudio in Moodle using LTI Advantage

The registration process currently requires no interaction and results in a badge for STEMCstudio in the Pending state.



Pending

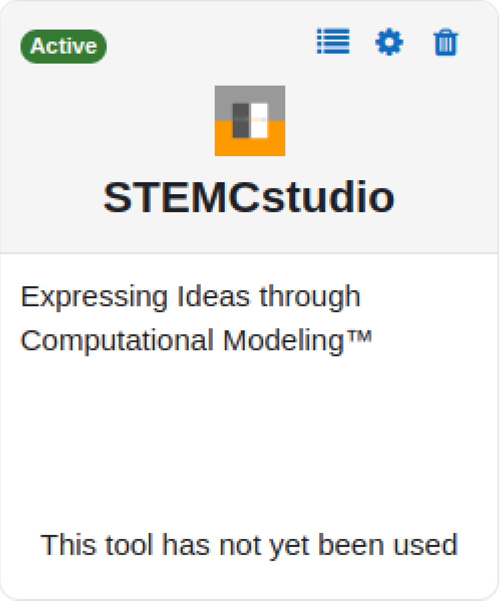
STEMCstudio

Expressing Ideas through Computational Modeling™

Activate

Figure 9. STEMCstudio tool with Pending status in Moodle

The tool must be activated before it can be used.



Active

STEMCstudio

Expressing Ideas through Computational Modeling™

This tool has not yet been used

Figure 10. STEMCstudio tool with Active status in Moodle

2.3. Deep Linking

Deep Linking allows Activities and Resources defined in the tool to be linked to the LMS with minimal manual configuration. The user interface of the tool itself is used to configure the link. In the case of STEMCstudio, this means selecting the project (Gist) and setting the parameters that determine how the workspace appears.

Creating a new Activity in Moodle requires the following steps:

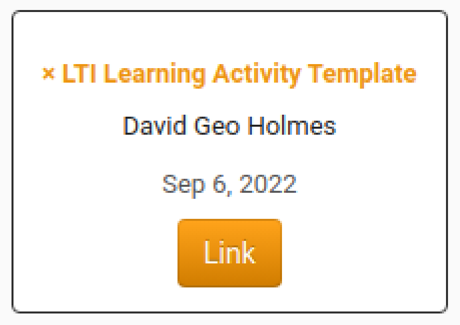
1. Select a Course.
2. Add an activity or resource.

3. Select *External Tool* as the activity type.
4. Choose *STEMCstudio* from the list of preconfigured tools (STEMCviewer can also be chosen and is a simpler process)
5. Press the *Select content* button.

This will navigate you to the STEMCstudio Home Page. From here you can select your STEMCstudio project in two ways.

1. Log in to GitHub in STEMCstudio and *download* your project.
2. Find the project in the *STEMCarXiv* (assuming you have published it).

If the workspace opens with your project, you will need to navigate back to the STEMCstudio Home Page in order to see your project with a *Link* button.



× LTI Learning Activity Template

David Geo Holmes

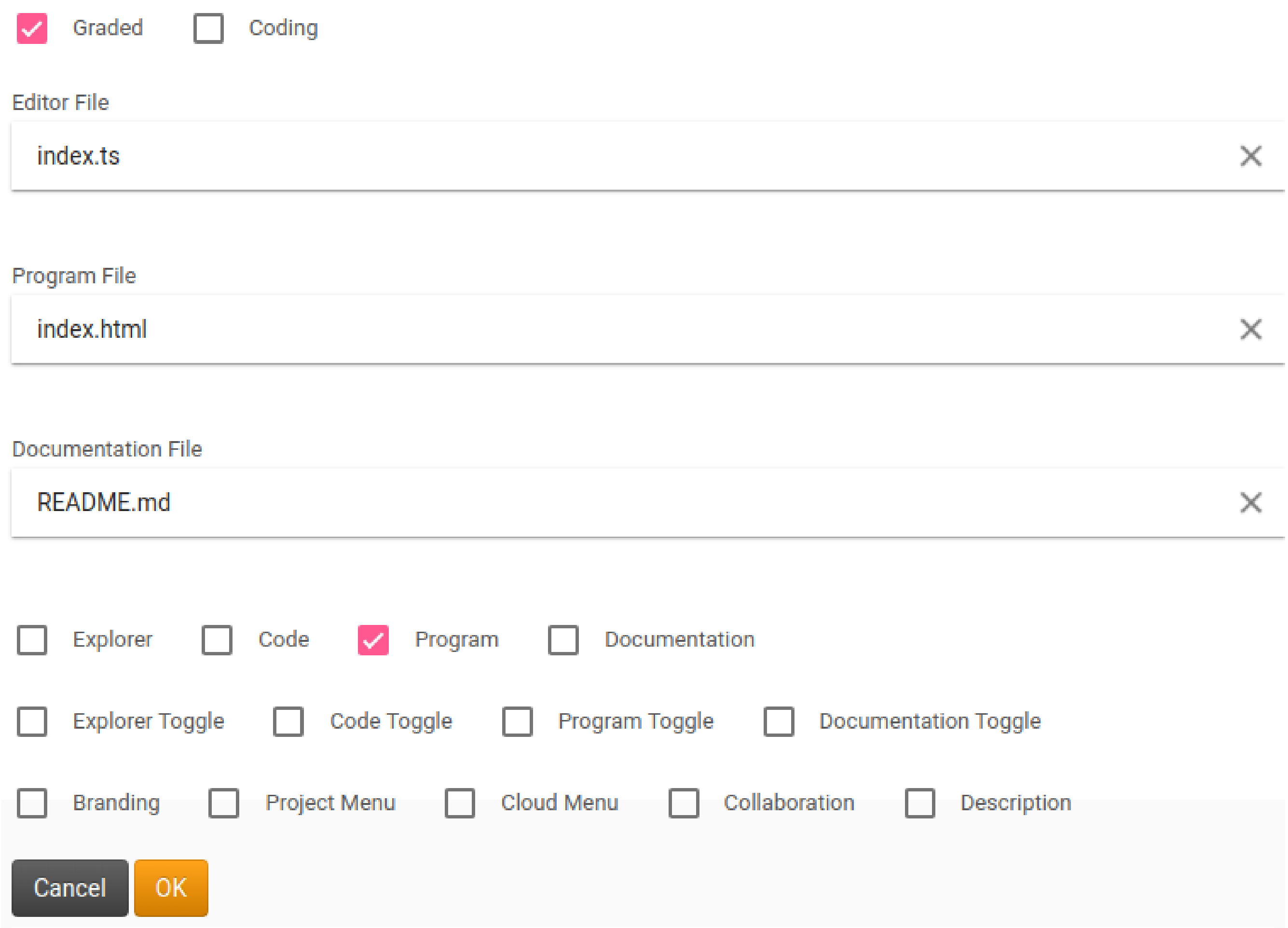
Sep 6, 2022

Figure 11. STEMCstudio Home Page with Project available for Deep Linking

Press the *Link* button to begin the *Deep Linking* process.

STEMCstudio will now present you with a dialog that will enable you to configure your resource.

The top part of the dialog offers configuration parameters.



☑ Graded ☐ Coding

Editor File: index.ts

Program File: index.html

Documentation File: README.md

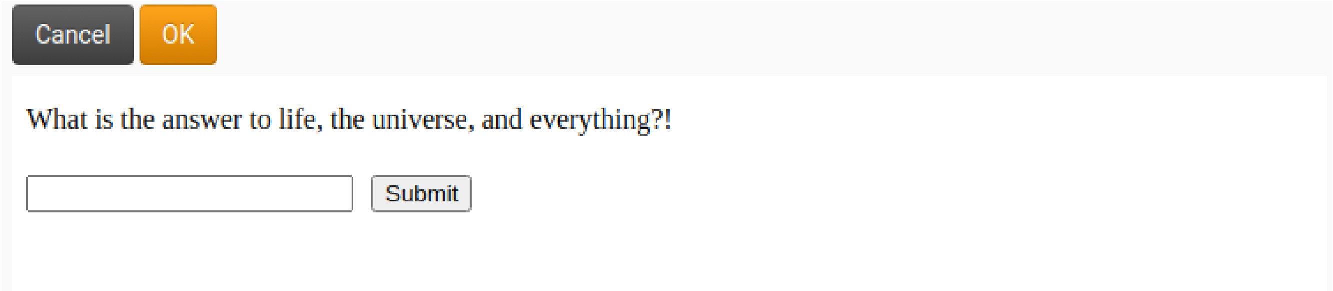
☐ Explorer ☐ Code ☑ Program ☐ Documentation

☐ Explorer Toggle ☐ Code Toggle ☐ Program Toggle ☐ Documentation Toggle

☐ Branding ☐ Project Menu ☐ Cloud Menu ☐ Collaboration ☐ Description

Figure 12. STEMCstudio Deep Linking Resource Configuration

The bottom part of the dialog shows you what will be displayed.



What is the answer to life, the universe, and everything?!

Figure 13. STEMCstudio Deep Linking WYSIWYG (What You See Is What You Get)

Use the *Graded* and *Coding* checkboxes to define the coarse behavior. Use the remaining options to fine tune the activity. You can think of the *Graded* and *Coding* options as dividing the space of activity into four kinds:

Graded	Coding	Kind of Activity
No	No	Demonstration
No	Yes	Student Coding Exploration
Yes	No	Quiz or Question
Yes	Yes	Graded Coding Exercise

There is no intrinsic meaning to the *Graded* and *Coding* options; they merely cause

> sensible choices to be made for the options below them.

Once you are satisfied with your choice of options, press the *OK* button.

This will return you to the LMS where you can Save the activity.

2.4. Programming API

STEMCstudio provides an Application Programming Interface (API) that allows you to access the following LTI services:

Assignments and Grades Service, and Names and Roles Service.

Interaction with these services from your STEMCstudio application happens through the 'stemcstudio-tunnel' package. This package provides abstractions that represent the Gradebook, the User, and the Cohort defined by the users' registration in a course.

This diagram illustrates the intermediaries involved in accessing the *Gradebook* from *Your Code*:

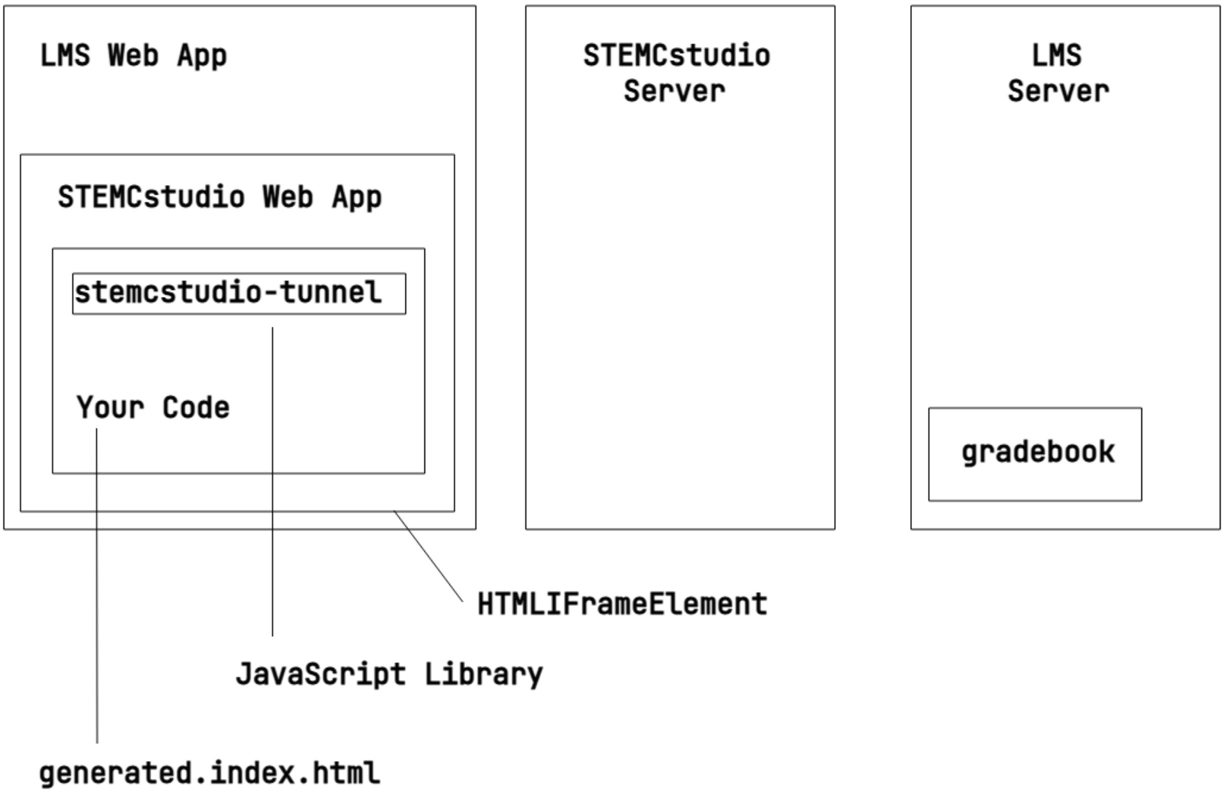


Figure 14. Your Code communicating with LMS Server through the stemcstudio-tunnel library.

> The communication between *Your Code* and the *STEMCstudio App* is performed - behind the scenes - as message passing because your application is running in an *HTMLIFrameElement*. Additionally, the communication between the *STEMCstudio App* and both servers happens using *HTTPS* web service calls. For both these reasons, the communication between *Your Code* and the *gradebook* must be asynchronous. This will require some familiarity with either the JavaScript

Promise API or the *async/await* syntax.

To get started we must make `stemcstudio-tunnel` available as a dependency. Having done that, the following import makes the LTI services available as JavaScript objects:

```
import { gradebook, Item, cohort, Score, user } from 'stemcstudio-tunnel'
```

The following code examples are taken from the STEMCstudio project https://www.stemcstudio.com/gists/96ea9435cb61fa7ba342bb226fb99623. When you run this project in STEMCstudio outside of the LMS, STEMCstudio will simulate some responses to aid in application development. When you deploy your application to your LMS it will use the real LTI services.

Be sure to be signed on with the correct roles when testing your LTI activity. If your *admin* or *instructor* is not enrolled in the course you are developing then requests such as submitting scores will fail.

Get the Gradebook Items (Columns) for the current Activity.

You'll need to do this if you want to submit scores to the gradebook. The zeroth element of the `Item[]` array returned by `gradebook.getItems()` contains the gradebook column for the current activity and has an identifier property called `id`.

The zeroth element of the `Item[]` array is the default gradebook column. We will see later that it is possible to add additional gradebook columns for the current activity.

```
async function getItems() {
    try {
        const items: Item[] = await gradebook.getItems()
        console.log('GRADEBOOK ITEMS')
        console.log('===============')
        console.log(JSON.stringify(items, null, 2))
        user.alert({ title: "Get Items", message: JSON.stringify(items, null, 2) })
    } catch (e) {
        console.warn(`${e}`)
    }
}
```

Submit Score for the current Activity.

Submitting a score for the current activity requires that you provide the identifier of the gradebook column.

```
async function submitScore() {
    try {
        const comment = await user.prompt({ title: 'Submit Score', message: 'Please add a comment.', text: 'This is outstanding work!', label: 'comment', hint: "" })
        if (gradebookItems.length > 0) {
            const score: Score = {
                scoreGiven: Math.random(),
                scoreMaximum: 1,
                activityProgress: 'Completed',
                gradingProgress: 'FullyGraded',
                comment
            }
            try {
                await gradebook.submitScore(gradebookItems[0].id, score)
                console.log('SCORE SUBMIT OK')
                console.log('===============')
            } catch (e) {
                console.warn(`${e}`)
            }
        }
    } catch (e) {
        console.warn(e)
    }
}
```

Get Results for the current Activity

Getting the results for the current activity and student is

```
async function getResults() {
    if (gradebookItems.length > 0) {
        const results = await gradebook.getResults(gradebookItems[0].id)
        console.log('GRADEBOOK RESULTS')
        console.log('=================')
        console.log(JSON.stringify(results, null, 2))
        user.alert({ title: "Get Results", message: JSON.stringify(results, null, 2) })
    }
}
```

Creating a new Gradebook Item (Column) for the current Activity.

Every activity has, by default, exactly one gradebook item when it is created. However, the LTI Advantage API allows you to create new columns on-the-fly for the current activity. This is how it is done:

```
async function createItem() {
    const itemdef: Omit<Item, 'id'> = {
        scoreMaximum: 23.5,
        label: 'My New Item',
        tag: 'My Tagge',
        // startDateTime: '2022-10-01T00:00:00',   // Has no effect but must be syntactically correct
        // endDateTime: '2022',
        resourceLinkId: 'RLID', // Can't be changed
        ltiLinkId: 'LLIID', // Can't be changed
        resourceId: "RID2"   // Can be changed
    }
    try {
        const item = await gradebook.createItem(itemdef)
        console.log('CREATED ITEM')
        console.log('============')
        console.log(JSON.stringify(item, null, 2))
    } catch (e) {
        console.warn(`${e}`)
        user.alert({ title: "Create Item", message: `${e}` })
    }
}
```

Get Membership for the current Activity Context.

You probably won't need to do this. The Names and Roles Service allows tools to synchronize with the course membership in the LMS.

```
async function getMembers() {
    try {
        const response = await cohort.getMembers()
        console.log('COHORT MEMBERS')
        console.log('==============')
        console.log(JSON.stringify(response, null, 2))
        for (const member of response.members) {
            console.log(`${member.name} ${JSON.stringify(member.roles)}`)
        }
    } catch (e) {
        console.warn(`${e}`)
    }
}
```

2.5. Summary

LTI 1.3, a.k.a. LTI Advantage, is a technical standard that describes how Learning Management Systems can securely link to external Content Providers to provide content for courses.

STEMCstudio understands the LTI 1.3 protocol; STEMCstudio applications can be launched from an LMS and can interact with the Gradebook.

Chapter 3. Useful STEM Libraries

This chapter will explore some useful libraries that can make you more productive in producing your application. The purpose of this chapter is not to provide an in-depth tutorial of each library but rather to introduce and show the concept behind various libraries that may be useful in constructing a STEM Learning Activity.

3.1. 2D Scalable Vector Graphics with g20

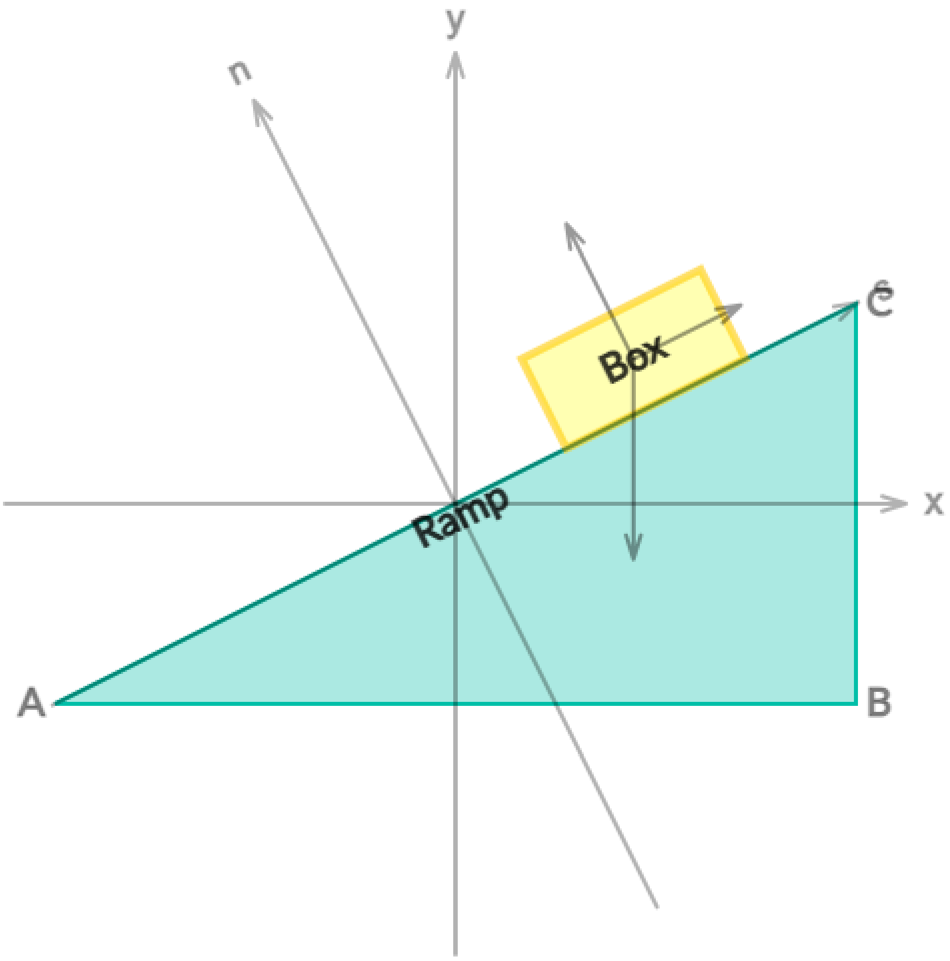


Figure 15. g20 being used to render a Block on an inclined Ramp

Introduction

g20 provides a modern JavaScript (TypeScript) suite of ESM modules for rendering 2D graphics using Scalable Vector Graphics (SVG) in the browser. This suite of modules was expressly designed for educational purposes and use within STEMCstudio.

The home page is at https://github.com/geometryzen/g20mono.

You can view the application at the following URL:

https://www.stemcviewer.com/gists/38aa01dfe4eca3a22d3f972d17c17df2

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/38aa01dfe4eca3a22d3f972d17c17df2

The official *npm* page for the core package, `@g20/core`, is https://www.npmjs.com/package/@g20/core. This page informs you of the latest runtime version and provides a link to the GitHub repository. All applications using `@g20/core` will also need the shared reactive signals package, `@g20/reactive`. Other packages described on the GitHub Monorepo Home Page are optional. When using the packages in your project, ensure that the version numbers are all the same; the packages are all published together and with the same version number to guarantee that they are compatible.

Configuration

Configuring your STEMCstudio project to use @g20 involves creating a dependency on '@g20/core', '@g20/reactive', and '@g20/svg' in your `package.json` file. There is no need to specify any overrides (which would be in `studio.config.json`). In the following example, there is also a dependency of `g20/grid` so that we can create a Grid.

package.json

```json
{
    "description": "@g20 Block on Ramp",
    "dependencies": {
        "@g20/core": "1.0.0-alpha.47",
        "@g20/grid": "1.0.0-alpha.47",
        "@g20/svg": "1.0.0-alpha.47"
    },
    "name": "g20-block-on-ramp",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "keywords": [
        "@g20",
        "STEMCstudio",
        "2D",
        "Graphics",
        "Geometric",
        "Algebra",
        "Block",
        "Ramp"
    ],
    "private": true
}
```

As a result of this configuration, the generated *system.config.json* and *types.config.json* files will look like:

system.config.json

```json
{
    "map": {
        "@g20/core": "https://cdn.jsdelivr.net/npm/@g20/core@1.0.0-
alpha.47/package.json",
        "@g20/grid": "https://cdn.jsdelivr.net/npm/@g20/grid@1.0.0-alpha.47/package.json",
        "@g20/svg": "https://cdn.jsdelivr.net/npm/@g20/svg@1.0.0-alpha.47/package.json",
        "@g20/reactive": "https://cdn.jsdelivr.net/npm/@g20/reactive@1.0.0-alpha.47/package.json"
    }
}
```

Notice that the runtime modules are found automatically from each *package.json* file.

Similarly, the TypeScript type definitions are found automatically from each *package.json* file:

types.config.json

```
{
    "map": {
        "@g20/core": "https://cdn.jsdelivr.net/npm/@g20/core@1.0.0-alpha.47/package.json",
        "@g20/grid": "https://cdn.jsdelivr.net/npm/@g20/grid@1.0.0-alpha.47/package.json",
        "@g20/svg": "https://cdn.jsdelivr.net/npm/@g20/svg@1.0.0-alpha.47/package.json",
        "@g20/reactive": "https://cdn.jsdelivr.net/npm/@g20/reactive@1.0.0-alpha.47/package.json"
    }
}
```

Usage

In this example, the *index.html* file provides a container for the rendered drawing (which will be an `svg` element) by providing a `div` element with an identifier so that it may be found from JavaScript code. The `class` attribute has been set to `board` so that it may be styled in the `style.css` file. Notice that the size of the rendering has been set in this file to 500 pixels by 500 pixels.

index.html

```
<!DOCTYPE html>
<html>

<head>
    <base href='/'>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel='stylesheet' href="style.css" />
</head>

<body>
    <div class="board" style="width:502px; height:502px">
```

```
        <div id='my-board' style='width:500px; height:500px'></div>
    </div>
</body>

</html>
```

In the *index.ts* file, the `initBoard` function call constructs the svg element that holds the drawing. The `boundingBox` attribute specifies the user coordinate system. Various shapes are then constructed and added to the board. The shapes in the core (`@g20/core`) package may either be created explicitly and then added to the board, or the convenience methods on the Board can be used to combine these two steps.

index.ts

```
import { Arrow, G20, Polygon, Rectangle } from '@g20/core';
import { Axes, Grid } from '@g20/grid';
import { initBoard } from '@g20/svg';

const size = 1; // The unit length in user coordinates

const board = initBoard("my-board", {
    boundingBox: { left: -size, top: size, right: size, bottom: -size }
});

board.defaults.text.fontFamily = "Lato";
board.defaults.text.fontSize = 20;

const gridXY = new Grid(board);
board.add(gridXY);

const axesXY = new Axes(board);
board.add(axesXY);
axesXY.opacity = 0.3;
axesXY.xAxis.strokeColor = 'black';
axesXY.xAxis.strokeWidth = 2 / board.sx;
axesXY.yAxis.strokeColor = 'black';
axesXY.yAxis.strokeWidth = 2 / board.sx;

const axesSN = new Axes(board, {});
board.add(axesSN);
axesSN.xAxis.strokeColor = 'black';
axesSN.xAxis.strokeWidth = 2 / board.sx;
axesSN.yAxis.strokeColor = 'black';
axesSN.yAxis.strokeWidth = 2 / board.sx;
axesSN.opacity = 0.3;
axesSN.visibility = 'visible';

const A = board.point([0.0, 0.0], {
    id: 'A',
    visibility: 'visible',
    text: "A",
    anchor: "end",
    baseline: "middle",
    dx: -5,
    hideIcon: true
});

const B = board.point([size * 8 / 5, 0.0], {
    id: 'B',
    visibility: 'visible',
    text: "B",
    anchor: "start",
    baseline: "middle",
    dx: 5,
    hideIcon: true
});
const C = board.point([size * 8 / 5, size * 4 / 5], {
    id: 'C',
    visibility: 'visible',
    text: "C",
    anchor: "start",
    baseline: "middle",
    dx: 5,
    dy: -10,
    hideIcon: true
});

const AB = B.X - A.X;
const AC = C.X - A.X;
const S = AC.normalize();
const N = S * G20.I;

const ramp = new Polygon(board, [A.X, B.X, C.X], {
    id: 'ramp',
    fillColor: 'rgb(0, 191, 168)',
    fillOpacity: 0.3,
    strokeColor: 'rgb(0, 191, 168)',
    strokeWidth: 0.016 * size
});
board.add(ramp);
ramp.center();
ramp.visibility = 'visible';

const block = new Rectangle(board, {
    id: 'box',
    width: size * 2 / 5,
    height: size * 1 / 5,
    fillColor: "#FFFF00",
    fillOpacity: 0.3,
    strokeColor: "#FFCC00",
    strokeOpacity: 0.6,
    strokeWidth: 4 / board.sx
});
board.add(block);
block.R.rotorFromDirections(AB, AC);
block.X = A.X + AC * 0.75 + N * block.height / 2;

axesSN.R.rotorFromDirections(AB, AC);
axesSN.xLabel.content = 's';
axesSN.yLabel.content = 'n';
// gridSN.R.rotorFromDirections(AB, AC);

const textD = board.text("Block", {
    id: 'text-D',
    anchor: 'middle',
    baseline: 'middle',
    opacity: 0.7,
    position: block.X
});
textD.R.rotorFromDirections(AB, AC);

const textE = board.text("Ramp", {
    id: 'text-E',
    anchor: 'middle',
    baseline: 'hanging',
    opacity: 0.7,
    position: ramp.X
});
textE.R.rotorFromDirections(AB, AC);

const Fg = board.arrow(- G20.ey * 2.0 * size / 5, {
    position: block.X,
    strokeColor: 'black',
    headLength: 0.05 * size
});
Fg.strokeOpacity = 0.4;
Fg.strokeWidth = 3 / board.sx;

const Fn = new Arrow(board, N * 1.5 * size / 5, {
    position: block.X,
    headLength: 0.05 * size,
    strokeColor: 'black',
    strokeWidth: 3 / board.sx
});
board.add(Fn);
Fn.strokeOpacity = 0.4;

const Fs = board.arrow(S * 1.2 * size / 5, {
    position: block.X,
    headLength: 0.05 * size,
    strokeColor: 'black',
    strokeWidth: 3 / board.sx
```

```
});
Fs.strokeOpacity = 0.4;

function animate(_timestamp: number) {
    // const x = (Math.sin(_timestamp / 2000) + 1) / 2;
    // box.X = A.X + AC * x + (N * box.height / 2);
    window.requestAnimationFrame(animate);
}

window.requestAnimationFrame(animate);

window.onunload = function() {
    try {
        board.dispose();
    }
    catch (e) {
        console.warn(`${e}`);
    }
};
```

You may want to provide some styling of your Board's container element as has been done here:

style.css

```
body {
    background-color: #cccccc;
    margin: 8px;
}

.board {
    background-color: #ffffff;
    border-style: solid;
    border-width: 2px;
    border-color: #0066ff;
    border-radius: 10px;
}
```

Consult the official documentation for further details on how to use @g20.

3.2. 2D Diagramming with JsxGraph

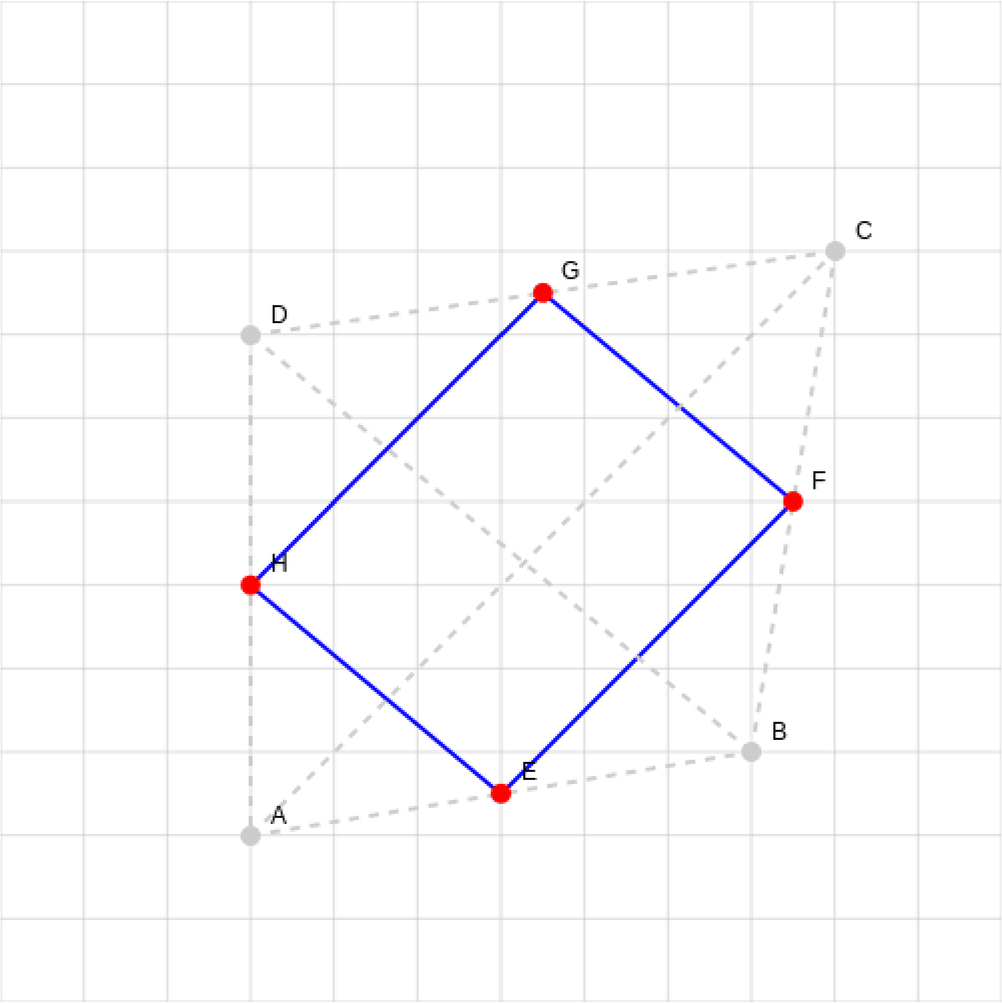


Figure 16. JsxGraph being used to demonstrate a Euclidean theorem

JsxGraph is desribed as a JavaScript library for interactive geometry. It is known especially for 2D rendering but can also render in 3D. The underlying rendering technology is pluggable but the most common technology used is Scalable Vector Graphics (SVG). The programming API consists of defining geometric elements, usually starting with points, and connecting them together to create geometric constructions. JsxGraph is able to render diagrams as well as text. JsxGraph is also an excellent utility for constructing specialized diagrams e.g. mechanics, electronics, and electrodynamics.

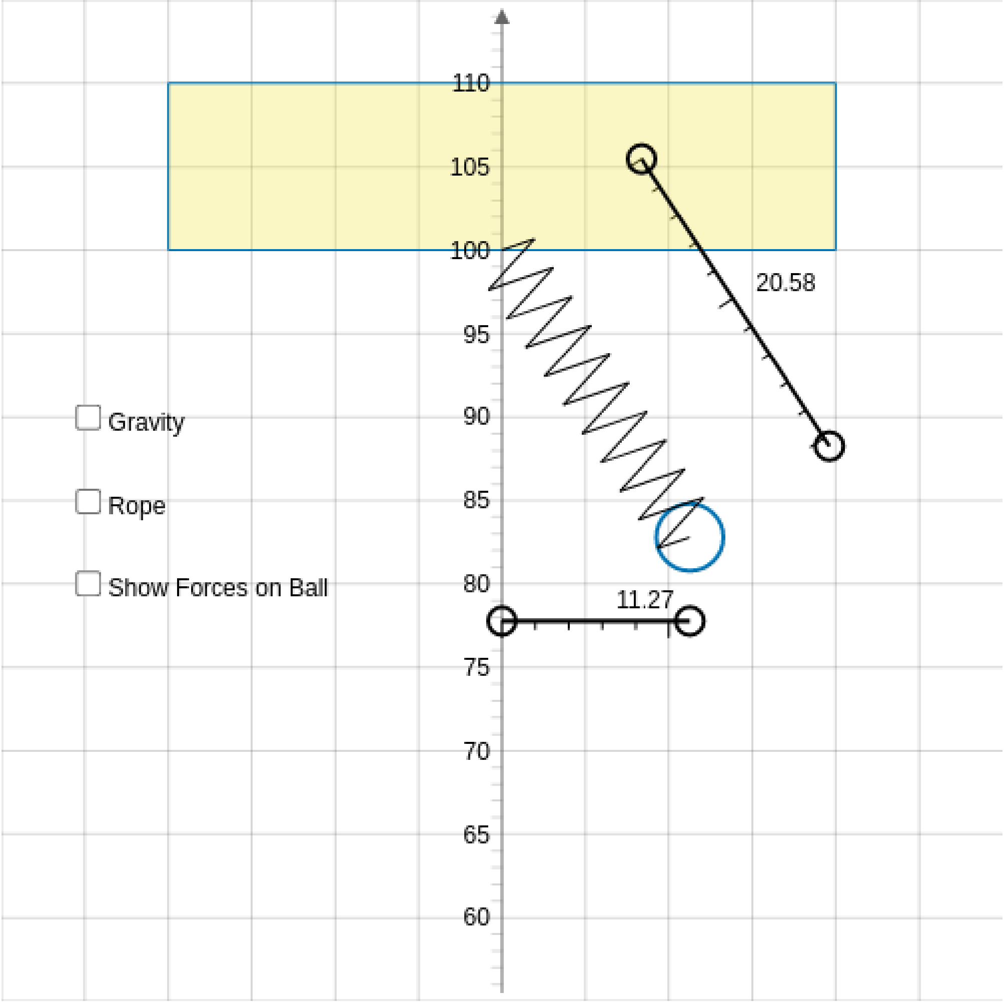


Figure 17. JsxGraph being used to render a Ball on a Spring Physics Simulation

Using JsxGraph in STEMCstudio

The starting point for using JsxGraph in STEMCstudio is to ensure that `jsxgraph` exists as a dependency in your project `package.json` file. You can either add this dependency manually be directly editing the `package.json` file or by using the (Add Dependency) button above the explorer view.

```
{
    "description": "JSXGraph Template",
    "dependencies": {
        "jsxgraph": "1.10.1"
    },
    "name": "jsxgraph-template",
    "version": "1.0.0",
    "keywords": [
        "JSXGraph",
        "template",
        "STEMCstudio",
        "Beginner"
    ],
    "private": true,
    "author": "David Geo Holmes"
}
```

It is common to define a `div` HTML tag in your `index.html` file as a placeholder for the creation of the `Board`.

```html
<!DOCTYPE html>
<html lang='en'>

<head>
    <meta charset="UTF-8">
    <title>JSXGraph template</title>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
    <base href='/'>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel='stylesheet' href="https://cdn.jsdelivr.net/npm/jsxgraph@1.10.1/distrib/jsxgraph.css" />
    <link rel='stylesheet' href="style.css" />
</head>

<body>
    <div id='my-board' class='jxgbox' style='width:500px; height:500px'></div>
</body>

</html>
```

The `jxgbox` class is defined in the `jsxgraph.css` file in the JsxGraph distribution.

The JsxGraph distribution provides a `jsxgraph.css` file with pre-defined Cascading Stylesheet Styles. This is optional but a useful starting point. Include the file by using a `link` HTML tag as shown.

You are now ready to begin coding your JsxGraph construction. The following example demonstrates the use of the programming API.

```javascript
import { JSXGraph } from 'jsxgraph'

const board = JSXGraph.initBoard("my-board", {
    axis: true,                     // Default is false
    boundingBox: [-5, 5, 5, -5],    // Default is [-5, 5, 5, -5]
    showCopyright: true,            // Default is true
    showFullscreen: true,           // Default is false
    showNavigation: true,           // Default is true
    showScreenshot: true            // Default is false
})

const A = board.create('point', [0, 0])
const B = board.create('point', [4, 0])
const C = board.create('point', [0, 4])

board.create('circle', [A, B])

const a = board.create('angle', [B, A, C], { radius: 3 })
const s = board.create('slider', [[-2, 1], [2, 1], [0, Math.PI * 0.5, 2 * Math.PI]])

a.setAngle(function() {
    return s.Value()
})

board.update()

window.onunload = function() {
    JSXGraph.freeBoard(board)
}
```

Maintaining the version of JsxGraph in your project.

Over time, you may wish to upgrade the version of JsxGraph that is being used by your project.

You should be aware that adding `jsxgraph` as a dependency to your `package.json` file caused an override entry to be created in your `studio.config.json` file:

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "jsxgraph",
            "version": "1.10.1",
            "system": "https://cdn.jsdelivr.net/npm/jsxgraph@1.10.1/distrib/jsxgraphcore.js",
            "types": "https://cdn.jsdelivr.net/npm/jsxgraph@1.10.1/package.json"
        }
    ],
    "references": {},
    "showGeneratedFiles": true
}
```

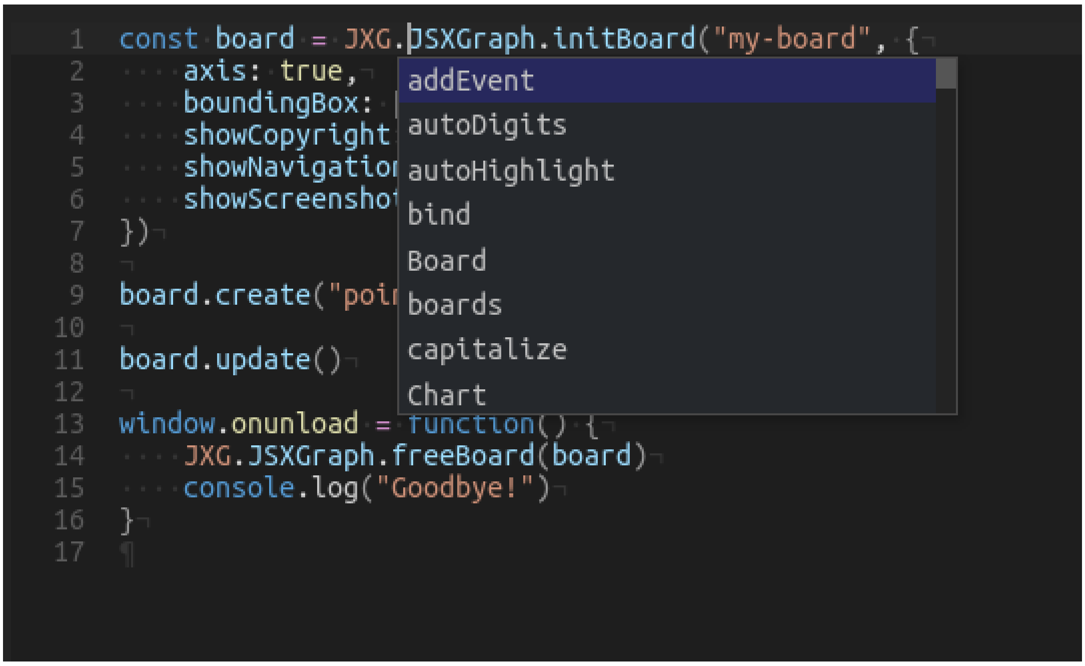
The override stipulates that the runtime implementation should come from the `@geometryzen/jsxgraph` wrapper (which provides a `system` module format) while the type definitions should come from the original `jsxgraph` package. Using the `jsxgraph` package, you are responsible for maintaining both the `package.json` file dependency as well as the `studio.config.json` override.

JXG Global Variable or ES6 Module Syntax

You have a choice in how you consume the JsxGraph library. This is because the JsxGraph library is bundled in a format called the Universal Module Definition (UMD). If you consume the library as a global variable then the variable name is *JXG* and it lives in the *window* namespace. Your code will look like this:

```
const board = JXG.JSXGraph.initBoard("my-board", {
    axis: true,
    boundingBox: [-6, 6, 6, -6],
    showCopyright: true,
    showNavigation: true,
    showScreenshot: true
})

board.create("point", [1, 1])

board.update()

window.onunload = function() {
    JXG.JSXGraph.freeBoard(board)
    console.log("Goodbye!")
}
```

To see the JavaScript resources that you can use from the *JXG* namespace, type a period after the *JXG* symbol.



```
const board = JXG.JSXGraph.initBoard("my-board", {
    axis: true,
    boundingBox:
    showCopyright
    showNavigatio
    showScreensho
})

board.create("poi

board.update()

window.onunload = function() {
    JXG.JSXGraph.freeBoard(board)
    console.log("Goodbye!")
}
```

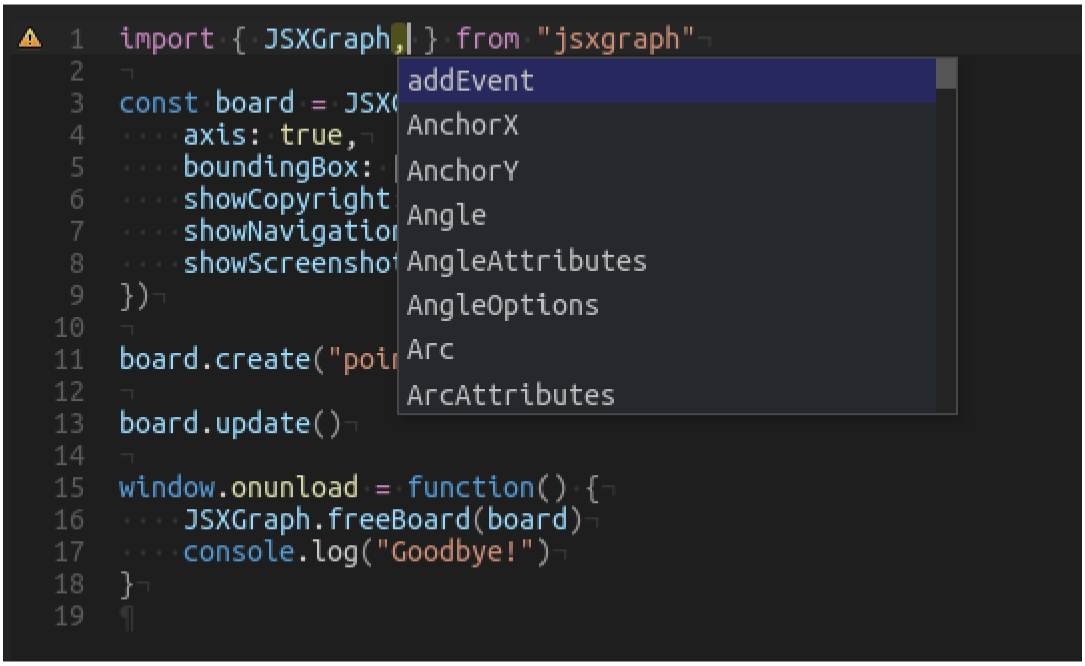
Figure 18. STEMCstudio Context Assist

If you consume the library using ES6 Module Syntax then your code will look like this:

```
import { JSXGraph } from "jsxgraph"
```

```
const board = JSXGraph.initBoard("my-board", {
    axis: true,
    boundingBox: [-6, 6, 6, -6],
    showCopyright: true,
    showNavigation: true,
    showScreenshot: true
})

board.create("point", [1, 1])

board.update()

window.onunload = function() {
    JSXGraph.freeBoard(board)
    console.log("Goodbye!")
}
```

To see the JavaScript resources that you can import from the *jsxgraph* module, type a comma after the *JSXGraph* symbol inside the curly braces of the *import* statement and press the *Ctrl+Spacebar* keys.



```
import { JSXGraph, } from "jsxgraph"

const board = JSX
    axis: true,
    boundingBox:
    showCopyright
    showNavigation
    showScreenshot
})

board.create("poi

board.update()

window.onunload = function() {
    JSXGraph.freeBoard(board)
    console.log("Goodbye!")
}
```

Figure 19. STEMCstudio Import Help

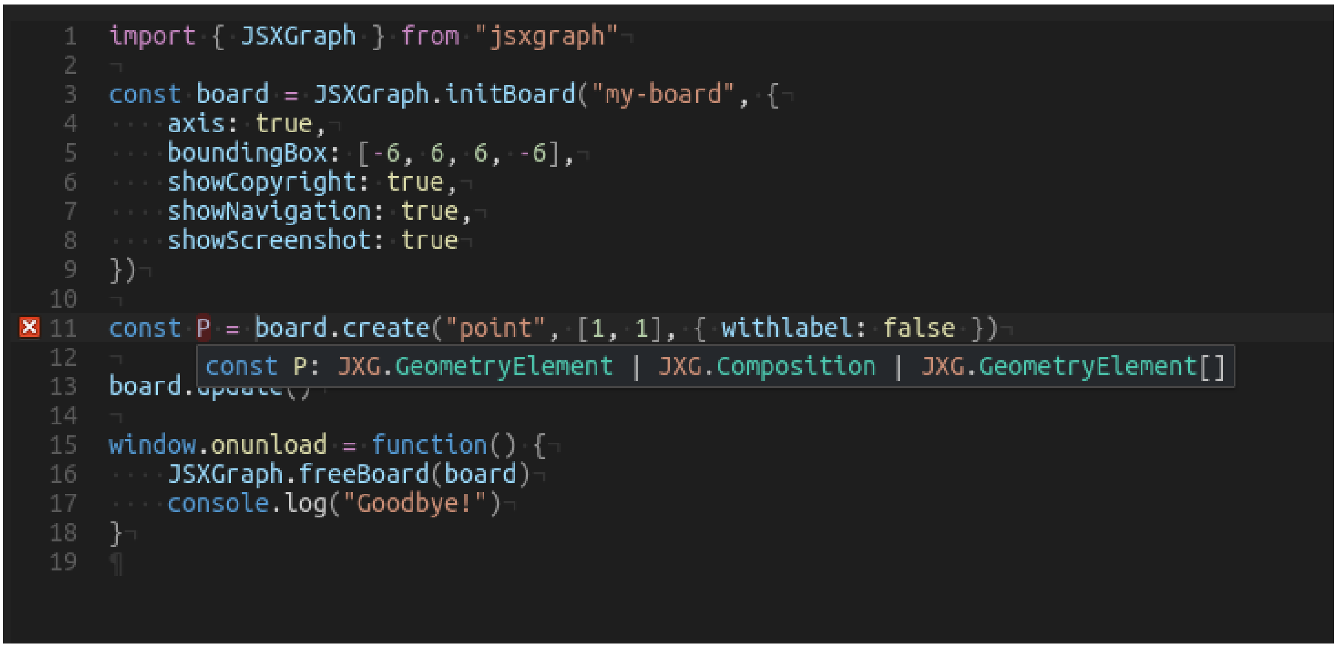
The choice as to whether you use the global *JXG* variable or ES6 Module Syntax is most likely going to be determined by your execution environment. For example, if you are using a legacy LMS plugin architecture then most likely JsxGraph will be available as the global *JXG* variable. Otherwise you should default to the more flexible ES6 Module Syntax and runtime architecture.

The overloaded *Board.create* method and pitfalls when using TypeScript

The *create* method of the *Board* class is overloaded to return different types according to the *elementType* parameter. Additionally, the second *parents* parameter may allow more than one type, and the optional third parameter usually has property names which are case-insensitive.

Frequently, a developer may report that the code runs correctly but the editor can't help with the semantic checking.

Let's look at a concrete example:



```
import { JSXGraph } from "jsxgraph"

const board = JSXGraph.initBoard("my-board", {
    axis: true,
    boundingBox: [-6, 6, 6, -6],
    showCopyright: true,
    showNavigation: true,
    showScreenshot: true
})

const P = board.create("point", [1, 1], { withlabel: false })

board.update()

window.onunload = function() {
    JSXGraph.freeBoard(board)
    console.log("Goodbye!")
}
```

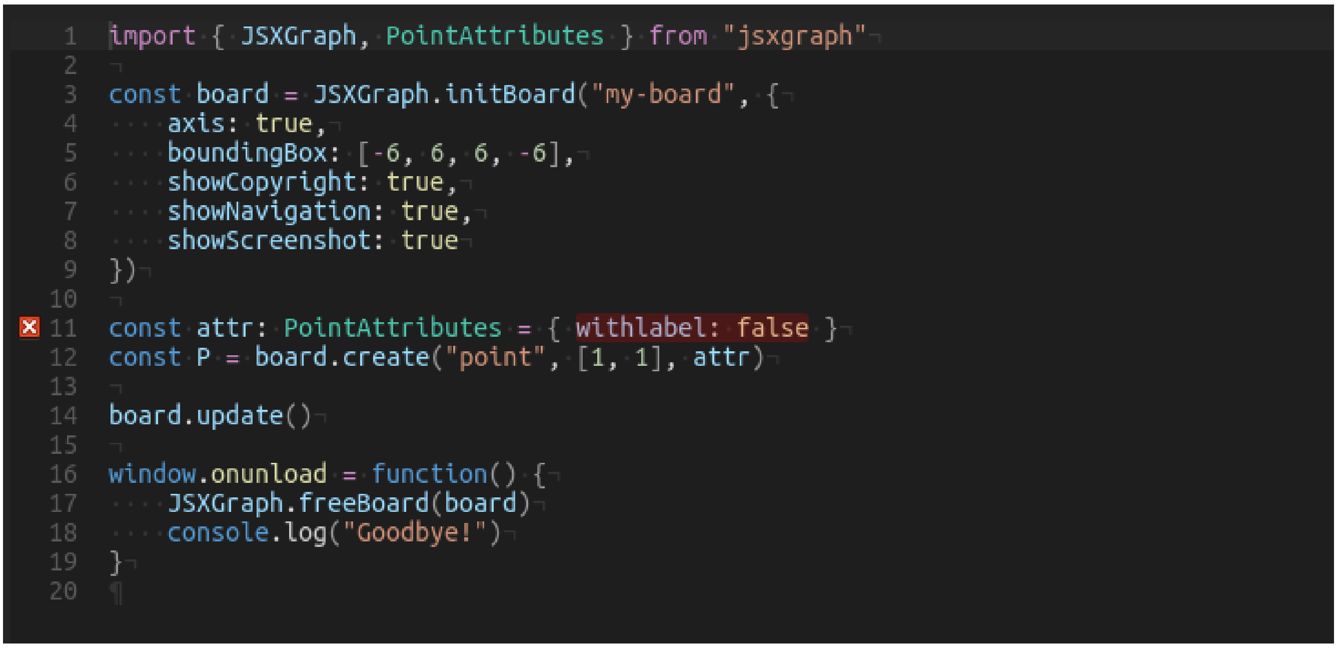
```
const P: JXG.GeometryElement | JXG.Composition | JXG.GeometryElement[]
```

Figure 20. JsxGraph Board.create overloaded issue

What is going on here?

We see that the developer intent is to create a *JXG.Point* instance, but the editor is confused and can only establish the general return type of the *create* method. The problem is difficult to spot.

If we break up the *JXG.Point* construction into two lines then the cause becomes apparent:



```
import { JSXGraph, PointAttributes } from "jsxgraph"

const board = JSXGraph.initBoard("my-board", {
    axis: true,
    boundingBox: [-6, 6, 6, -6],
    showCopyright: true,
    showNavigation: true,
    showScreenshot: true
})

const attr: PointAttributes = { withlabel: false }
const P = board.create("point", [1, 1], attr)

board.update()

window.onunload = function() {
    JSXGraph.freeBoard(board)
    console.log("Goodbye!")
}
```

Figure 21. JsxGraph Board.create debugging

The property name *withlabel* is incorrect and should be *withLabel* (according to the TypeScript type definitions). The code runs correctly because the runtime allows case insensitivity in the property names. But at design time, the difference is sufficient to throw off the matching that establishes the return type. There is a trade-off going on here. If the typing of the *create* method is very lax then it can be made to match based on the *elementType* parameter, but the definitions will not provide useful information on the second and third parameters. If the typing of the *create* method is exhaustive then it will not match when there are common mistakes.

I believe the pragmatic solution is to proceed by breaking the construction into multiple lines, as in the previous example. In this case you will get more help for the third *attributes* parameter.

3.3. 3D Graphics with Eight

When we think of 3D Graphics, we are normally referring to a high performance processing pipeline that takes advantage of the parallel processing of the Graphics Processor Unit (GPU). An example of this is *WebGL*. The *WebGL* API is very low level and not very practical for most STEM education needs where the desire is to use common geometric objects to render scenes. Higher level APIs are available in JavaScript libraries that hide the implementation details of WebGL and provide suitable abstractions for building 3D scenes.

While the `three` package (https://www.npmjs.com/package/three) is well known and provides more than adequate functionality for rendering 3D graphics, an alternative is available that is designed to be more suitable for educational purposes. This package is called `davinci-eight`. The core ideas behind this package are:

- Uniform representation of position and attitude using Geometric Algebra. This avoids the ad-hoc approaches to rotations based on Euler angles and quaternions.
- Written in TypeScript with generated type definitions. The type definitions are guaranteed to be accurate and the library code is easier to maintain.
- Designed for extension at all levels in the architecture. This allows the user to create more sophisticated graphics and new objects by having clearly defined extension points.

While the `davinci-eight` package has different design objectives, the programming metaphor is very similar to `three`.

Try It

Here is an example of some of the out-of-the-box components available.

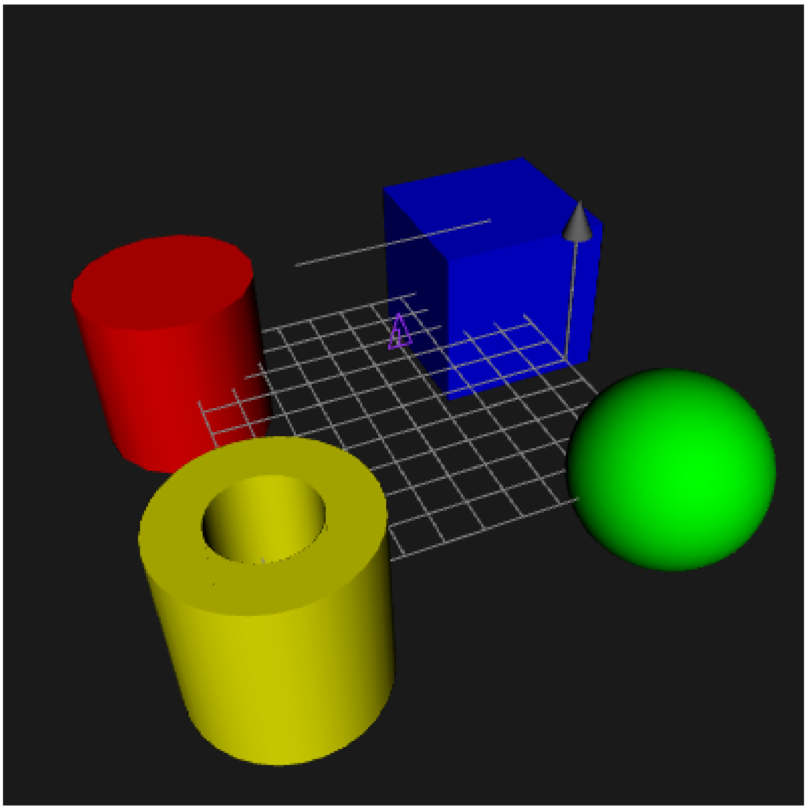


Figure 22. Graphics using the @geometryzen/eight WebGL Library

You can try the application at the following URL:

https://www.stemcviewer.com/gists/b58dd9a292ab3c34044a6231d7c00b4a

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/b58dd9a292ab3c34044a6231d7c00b4a

Configuration

The `package.json` file in your project must include a dependency on `davinci-eight`.

package.json

```json
{
    "description": "DaVinci eight Component Guide",
    "dependencies": {
        "davinci-eight": "8.4.57"
    },
    "name": "eight-visual-component-guide",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "keywords": [
        "STEMCstudio",
        "stemcbook"
    ]
}
```

There is no need to add an override to the `studio.config.json` file because the `davinci-eight` package is fully compatible with STEMCstudio.

Inspection of the generated `system.config.json` file will reveal that the runtime module is defined using the following mapping:

system.config.json

```
{
    "map": {
        "davinci-eight": "https://cdn.jsdelivr.net/npm/davinci-eight@8.4.57/package.json"
    }
}
```

Likewise, in the `types.config.json` file the type definitions are defined using the following mapping:

types.config.json

```
{
    "map": {
        "davinci-eight": "https://cdn.jsdelivr.net/npm/davinci-eight@8.4.57/package.json"
    }
}
```

How It Works

We'll look at a slightly simpler example to understand how the various parts fit together:

https://www.stemcstudio.com/gists/394d7777f6d3c37bd6fc6a1fe35748bf

The *index.html* file is quite simple. We essentialy need an HTML *canvas* element on which to construct a WebGL *Rendering Context*.

index.html

```
<!DOCTYPE html>
<html>

<head>
    <base href='/'>
    <link rel='stylesheet' href='style.css'>
</head>

<body>
    <canvas id='my-canvas'></canvas>
</body>

```

```
</html>
```

The *index.ts* file looks a bit daunting. Let's break it down. The *Engine* is constructed on the HTML *canvas* element. This essentially says that we are going to use the *canvas* element for *WebGL* purposes. We then create a *PerspectiveCamera* and a *DirectionalLight*. These are not part of the scene but they do modify the appearance of the scene. Each of these objects implements a *Facet* interface that provides information when the scene is rendered. The various facets are collected in an *ambients* array for use during rendering. The *TrackballControls* instance is an adapter that takes events from your pointer device and modifies the state of the camera. A `Drawable` Box is constructed. The appearance of the Box can be changed by setting various `BoxOptions`. Rendering is performed by the *animate* function, which calls the `render` method of the `Box`.

index.ts

```
import {
    Box,
    BoxOptions,
    Capability,
    Color,
    DirectionalLight,
    Engine,
    Facet,
    PerspectiveCamera,
    TrackballControls
} from "davinci-eight"

const engine = new Engine("my-canvas")
    .size(500, 500)
    .clearColor(0.1, 0.1, 0.1, 1.0)
    .enable(Capability.DEPTH_TEST)

const ambients: Facet[] = []

const camera = new PerspectiveCamera()
camera.eye.z = 5
ambients.push(camera)

const dirLight = new DirectionalLight()
ambients.push(dirLight)

const options: BoxOptions = { color: Color.green, mode: "mesh" }
const box = new Box(engine, options)

const trackball = new TrackballControls(camera, window)
// Subscribe to mouse events from the canvas.
trackball.subscribe(engine.canvas)

/**
 * animate is the callback point for requestAnimationFrame.
 * This has been initialized with a function expression in order
 * to avoid issues associated with JavaScript hoisting.
 */
const animate = function(timestamp: number) {
    engine.clear()

    // Update the camera based upon mouse events received.
    trackball.update()

    // Keep the directional light pointing in the same direction as the camera.
    dirLight.direction.copy(camera.look).sub(camera.eye)

    const t = timestamp * 0.001

    box.R.rotorFromGeneratorAngle({ xy: 0, yz: 1, zx: 0 }, t)

    box.render(ambients)

    // This call keeps the animation going.
    requestAnimationFrame(animate)
}

// This call "primes the pump".
requestAnimationFrame(animate)
```

Extending the Library

In this example we will take a look at one way in which the library can be extended to create new objects.

In this case the extension is defined in a STEMCstudio project and so can only be used by this project. However, you could write the extension into an external library.

You can also extend the library by creating new `Facet`(s). The `PerspectiveCamera` and `DirectionalLight` are examples of predefined `Facet`(s).

index.ts

```
import {
    BeginMode,
    Capability,
    Color,
    Engine,
    Facet,
    GeometryArrays,
    Mesh,
    PerspectiveCamera,
    Primitive,
    ShaderMaterial,
    TrackballControls
} from 'davinci-eight'
import { black, blue, cyan, green, magenta, red, white, yellow } from './colors'
import { e2, e3 } from './space'
import { windowResizer } from './windowResizer'

const engine = new Engine('canvas')
    .size(500, 500)
    .clearColor(0.1, 0.1, 0.1, 1.0)
    .enable(Capability.DEPTH_TEST)

const ambients: Facet[] = []

const camera = new PerspectiveCamera()
camera.eye.copy(e3 - e2).normalize().scale(5)
camera.up.copy(e3)
camera.projectionMatrixUniformName = 'P'
camera.viewMatrixUniformName = 'V'
ambients.push(camera)

const controls = new TrackballControls(camera, window)
controls.subscribe(engine.canvas)

/**
 * Coordinates of the cube vertices.
 */
const vertices = [
    [-0.5, -0.5, +0.5],
    [-0.5, +0.5, +0.5],
    [+0.5, +0.5, +0.5],
    [+0.5, -0.5, +0.5],
    [-0.5, -0.5, -0.5],
    [-0.5, +0.5, -0.5],
    [+0.5, +0.5, -0.5],
    [+0.5, -0.5, -0.5]
]

/**
 * Colors used for the faces.
 * Only six of the eight colors are used.
 */
const colors = [
    black,
    red,
    yellow,
    green,
    blue,
    magenta,
    cyan,
    white
]
```

```typescript

const aPositions: number[] = []
const aColors: number[] = []

/**
 * Pushes positions and colors into the the aPositions and aColors arrays.
 * A quad call pushes two triangles, making a square face.
 * The dimensionality of each position is 3, but could be changed.
 * The first parameter, a, is used to pick the color of the entire face.
 */
function quad(a: number, b: number, c: number, d: number): void {
    const indices = [a, b, c, a, c, d]

    for (const index of indices) {
        for (const vertex of vertices[index]) {
            aPositions.push(vertex)
        }
        const color: Color = colors[a]
        aColors.push(color.r)
        aColors.push(color.g)
        aColors.push(color.b)
    }
}

quad(1, 0, 3, 2)
quad(2, 3, 7, 6)
quad(3, 0, 4, 7)
quad(6, 5, 1, 2)
quad(4, 5, 6, 7)
quad(5, 4, 0, 1)

const primitive: Primitive = {
    mode: BeginMode.TRIANGLES,
    attributes: {
        aPosition: { values: aPositions, size: 3 },
        aColor: { values: aColors, size: 3 }
    }
}

const vertexShaderSrc = (document.getElementById('vs') as HTMLScriptElement).textContent as string
const fragmentShaderSrc = (document.getElementById('fs') as HTMLScriptElement).textContent as string

const geometry = new GeometryArrays(engine, primitive)
const material = new ShaderMaterial(vertexShaderSrc, fragmentShaderSrc, [], engine)
const cube = new Mesh(geometry, material, engine)
cube.modelMatrixUniformName = 'M'
cube.normalMatrixUniformName = 'N'

```

```
/**
 * Animates the scene.
 */
function animate() {

    engine.clear()

    controls.update()

    cube.render(ambients)

    requestAnimationFrame(animate)
}

windowResizer(engine, camera).resize()

// Get things going by requesting the first animation frame.
requestAnimationFrame(animate)
```

vs.glsl

```
attribute vec3 aPosition;
attribute vec3 aColor;
uniform mat4 M;
uniform mat3 N;
uniform mat4 P;
uniform mat4 V;
varying highp vec4 vColor;

void main(void) {
  gl_Position = P * V * M * vec4(aPosition, 1.0);
  vColor = vec4(aColor, 1.0);
}
```

fs.glsl

```
precision mediump float;
varying highp vec4 vColor;

void main(void) {
  gl_FragColor = vColor;
}
```

3.4. Data Visualization with Plotly

Plotly

Plotly is a JavaScript Open Source Graphing Library.

JavaScript examples for various chart types are documented by the *plotly* maintainers at https://plotly.com/javascript/. It is usually fairly straightforward to adapt these to TypeScript and use them in STEMCstudio.

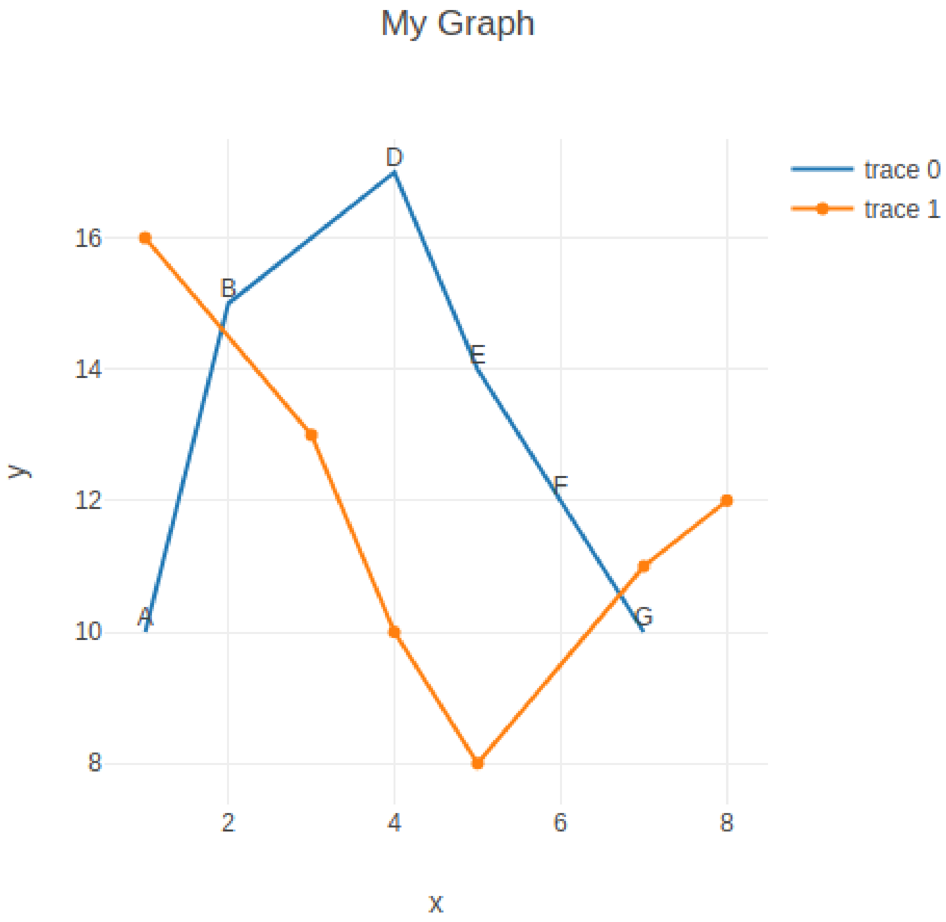


Figure 23. Plotly

You can try the application at the following URL:

https://www.stemcviewer.com/gists/8191c1070bc5d68cd223a33f01ce4d53

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/8191c1070bc5d68cd223a33f01ce4d53

The official *npm* page is https://www.npmjs.com/package/plotly.js. This page informs you of the runtime versions and locations on the CDN. Unfortunately, the developers behind *plotly* don't provide the TypeScript type definitions. However, a number of versions are hosted on the STEMCstudio server.

Configuring STEMCstudio to use plotly involves creating a dependency in `package.json` and having an override in `studio.config.json`.

package.json

```json
{
    "description": "plotly.js",
    "dependencies": {
        "plotly.js": "2.28.0"
    },
    "name": "graphing-with-plotly",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "keywords": [
        "STEMCstudio",
        "template",
        "stemcbook"
    ]
}
```

studio.config.json

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": false,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "plotly.js",
            "version": "2.28.0",
            "system": "https://cdn.jsdelivr.net/npm/plotly.js@2.28.0/dist/plotly.js",
            "types": "https://stemcstudio.com/vendor/plotly.js@2.17.0/index.d.ts"
        }
    ],
    "references": {},
    "showGeneratedFiles": true
}
```

> The runtime implementation for plotly is provided by a UMD module.

As a result of this configuration, the generated *system.config.json* and *types.config.json* files will look like:

system.config.json

```
{
    "map": {
        "plotly.js": "https://cdn.jsdelivr.net/npm/plotly.js@2.28.0/dist/plotly.js"
    }
}
```

types.config.json

```json
{
    "map": {
        "plotly.js": "https://stemcstudio.com/vendor/plotly.js@2.17.0/index.d.ts"
    }
}
```

The *index.html* file provides a placeholder for the chart by using a `div` element:

index.html

```html
<!DOCTYPE html>
<html>

<head>
    <base href='/'>
    <link rel='stylesheet' href='style.css'>
</head>

<body>
    <div id='my-graph' style='width: 500px; height:500px;'></div>
</body>

</html>
```

The *index.ts* file configures and constructs the chart. The code differs from the official JavaScript documentation by having a ES Module import for the package `plotly.js`, `const` variables, and the creation of the plot using the exported `newPlot` function rather than the global `Plotly` variable:

index.ts

```typescript
import { Layout, newPlot, PlotConfig, Trace } from 'plotly.js'

const trace1: Trace = {
    x: [1, 2, 3, 4, 5, 6, 7, 8],
    y: [10, 15, null, 17, 14, 12, 10, null, 15],
    text: ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
    textposition: 'top center',
    mode: 'lines+text',
    connectgaps: true
}

const trace2: Trace = {
    x: [1, 2, 3, 4, 5, 6, 7, 8],
    y: [16, null, 13, 10, 8, null, 11, 12],
    mode: 'lines+markers',
    connectgaps: true
}

```

```
const layout: Layout = {
    title: "My Graph",
    xaxis: { title: "x" },
    yaxis: { title: "y" },
    showlegend: true
}

const config: PlotConfig = {
    displayModeBar: false,
    scrollZoom: true
}

const graphDiv = document.getElementById('my-graph')
if (graphDiv) {
    newPlot(graphDiv, [trace1, trace2], layout, config)
}
```

3.5. Charting with Chart.js

Chart.js

Chart.js is described as

Simple yet flexible JavaScript charting library for the modern web

The official web page is

https://www.chartjs.org

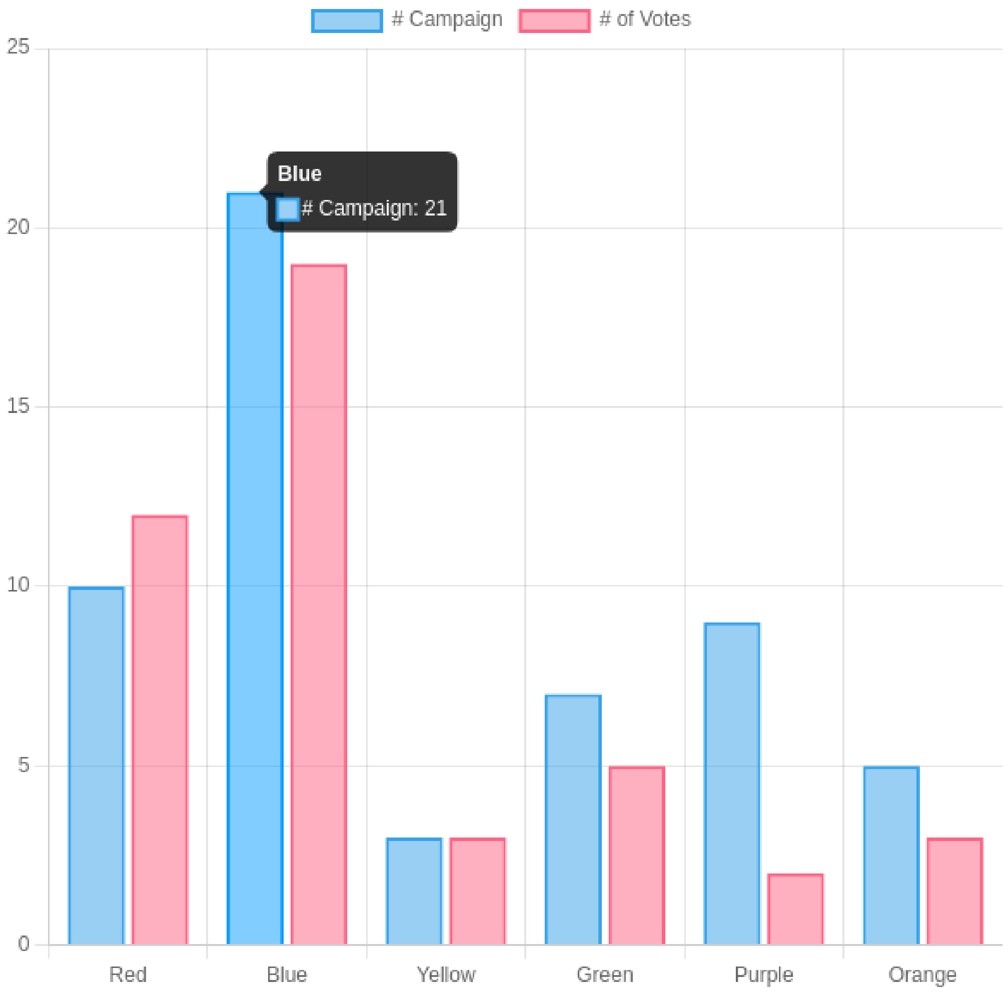


Figure 24. Chartjs

You can view a STEMCstudio sample application at the following URL:

https://www.stemcviewer.com/gists/bf1b63181f7192f104fd883af1219afb

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/bf1b63181f7192f104fd883af1219afb

The web page contains extensive examples. Here I will only describe the setup required to allow Chart.js to run in STEMCstudio.

Your project will contain the obligatory *index.html* file.

Because Chart.js uses the HTMLCanvasElement, your HTML document must contain a canvas element.

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div>
        <canvas id="myChart" height="500" width="500" style='width:500px; height:500px' aria-label="Hello ARIA World" role="img">
            <p>Fallback</p>
        </canvas>
    </div>
</body>

</html>
```

The *index.ts* file does the work of configuring the chart and loading the data.

index.ts

```
import { Chart } from 'chart.js'

const canvasElement = document.getElementById('myChart') as HTMLCanvasElement
if (canvasElement) {
    /* const chart = */ new Chart(canvasElement, {
        type: 'bar',
        data: {
            labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
            datasets: [
                {
                    label: '# Campaign',
                    data: [10, 21, 3, 7, 9, 5],
                    borderWidth: 2
                },
                {
                    label: '# of Votes',
                    data: [12, 19, 3, 5, 2, 3],
                    borderWidth: 2
                }
            ]
        },
        options: {
            scales: {
                y: {
                    beginAtZero: true
                }
            }
        }
    })
}
```

When you add a dependency for Chart.js, you will search for *chart.js*. One the dependency has been added, it will be stored in the *package.json* file under the *dependencies* property.

package.json

```json
{
    "description": "chart.js",
    "dependencies": {
        "chart.js": "4.4.2"
    },
    "name": "chart.js",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "private": true
}
```

STEMCstudio has an override for *chart.js* because the package cannot be consumed directly by only looking at the *package.json* file for *chart.js*. The override gets stored in the *studio.config.json* file. You may modify this file if you know what you are doing.

studio.config.json

```json
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "chart.js",
            "version": "4.4.2",
            "system": "https://cdn.jsdelivr.net/npm/chart.js@4.4.2/dist/chart.umd.js",
            "types": "https://cdn.jsdelivr.net/npm/chart.js@4.4.2/package.json"
        }
    ],
    "references": {},
    "showGeneratedFiles": true
}
```

From the *studio.config.json* file, STEMCstudio generates the files used for design-time and run-time.

types.config.json is used at design-time and provides the location of the TypeScript type definitions.

In the case of *chart.js*, we only need to know the location of its *package.json* file to find the type definitions.

types.config.json

```json
{
    "map": {
        "chart.js": "https://cdn.jsdelivr.net/npm/chart.js@4.4.2/package.json"
    }
}
```

system.config.json is used at run-time and provides the location of the executable JavaScript code.

In the case of *chart.js*, we ignore the *package.json* file and elect to use the UMD module format code.

system.config.json

```
{
    "map": {
        "chart.js": "https://cdn.jsdelivr.net/npm/chart.js@4.4.2/dist/chart.umd.js"
    }
}
```

3.6. Symbolic Mathematics using STEMCmicro

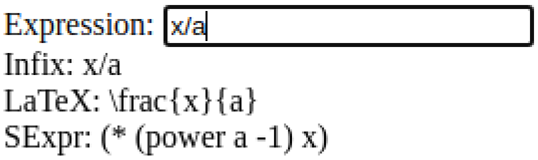
Using Symbolic Mathematics in an educational activity provides the opportunity to create more interesting problems.

Up till now this has been achieved by using specialized LMS plugins that invoke remote servers to transform expressions e.g. STACK. In STEMCstudio, the approach taken is to use a library that runs in the web browser. The ideas behind this new approach are to provide greater flexibility, to avoid the scaling issues caused by multiple users, and avoid the installation and versioning issues with a server-side implementation.

The definition of **Symbolic Mathematics** is rather broad. In the context of authoring STEMC educational activities, we may be interested in being able to verify that the input from a student matches some expression without regard to ambiguities in the ordering of terms and factors. One way to do this is to take advantage of the fact that symbolic mathematics processors typically normalize expressions and convert them to a canonical representation. This will be the focus of this example.

There are several JavaScript libraries available, each with its own strengths and weaknesses. In this example we will look at a library that is still under development that aims to fulfill this niche.

Our demonstration example allows the user to enter a mathematical expression. The parsed and normalized expression tree is then rendered in multiple formats:



Expression: x/a
Infix: x/a
LaTeX: \frac{x}{a}
SExpr: (* (power a -1) x)

Figure 25. Rendering a Mathematical Expression

You can try the application at the following URL:

https://www.stemcviewer.com/gists/245c2b74174423139930622611 3a576c

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/245c2b741744231399306226113a576c

Adding *stemcmicro* to your project is rather easy because *stemcmicro* natively supports the *system* runtime format and the TypeScript type definitions are generated and bundled with the NPM package. Simply search for *stemcmicro* when adding the dependency. Once found, the dependency will be added to the *package.json* file of your project.

package.json

```
{
    "description": "Symbolic Math with STEMCmicro",
    "dependencies": {
        "@stemcmicro/engine": "0.9.104",
        "@stemcmicro/js-parse": "0.9.104"
    },
    "name": "symbolic-math-jsxmath",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "keywords": [
        "stemcbook",
        "stemcmicro"
    ]
}
```

There are no overrides because the design-time and run-time artifacts are all in appropriate formats and locatable through the *stemcmicro* package.json file. Hence, the *studio.config.json* overrides property is empty.

studio.config.json

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": false,
    "linting": true,
    "noLoopCheck": false,
    "operatorOverloading": false,
    "overrides": [],
    "references": {},
    "showGeneratedFiles": true
}
```

STEMCstudio determines how to locate the run-time (executable) JavaScript code and places that information in *system.config.json*.

system.config.json

```json
{
    "map": {
        "@stemcmicro/js-parse": "https://cdn.jsdelivr.net/npm/@stemcmicro/js-parse@0.9.104/package.json",
        "@stemcmicro/engine": "https://cdn.jsdelivr.net/npm/@stemcmicro/engine@0.9.104/package.json",
        "@stemcmicro/context": "https://cdn.jsdelivr.net/npm/@stemcmicro/context@0.9.104/package.json",
        "@stemcmicro/atoms": "https://cdn.jsdelivr.net/npm/@stemcmicro/atoms@0.9.104/package.json",
        "@stemcmicro/directive": "https://cdn.jsdelivr.net/npm/@stemcmicro/directive@0.9.104/package.json",
        "@stemcmicro/em-parse": "https://cdn.jsdelivr.net/npm/@stemcmicro/em-parse@0.9.104/package.json",
        "@stemcmicro/native": "https://cdn.jsdelivr.net/npm/@stemcmicro/native@0.9.104/package.json",
        "@stemcmicro/tree": "https://cdn.jsdelivr.net/npm/@stemcmicro/tree@0.9.104/package.json",
        "@stemcmicro/stack": "https://cdn.jsdelivr.net/npm/@stemcmicro/stack@0.9.104/package.json"
    }
}
```

STEMCstudio determines how to locate the design-time TypeScript type definitions and places that information in *types.config.json*.

types.config.json

```json
{
    "map": {
        "@stemcmicro/engine": "https://cdn.jsdelivr.net/npm/@stemcmicro/engine@0.9.104/package.json",
        "@stemcmicro/js-parse": "https://cdn.jsdelivr.net/npm/@stemcmicro/js-parse@0.9.104/package.json",
        "@stemcmicro/context": "https://cdn.jsdelivr.net/npm/@stemcmicro/context@0.9.104/package.json",
        "@stemcmicro/atoms": "https://cdn.jsdelivr.net/npm/@stemcmicro/atoms@0.9.104/package.json",
        "@stemcmicro/directive": "https://cdn.jsdelivr.net/npm/@stemcmicro/directive@0.9.104/package.json",
        "@stemcmicro/em-parse": "https://cdn.jsdelivr.net/npm/@stemcmicro/em-parse@0.9.104/package.json",
        "@stemcmicro/native": "https://cdn.jsdelivr.net/npm/@stemcmicro/native@0.9.104/package.json",
        "@stemcmicro/stack": "https://cdn.jsdelivr.net/npm/@stemcmicro/stack@0.9.104/package.json",
        "@stemcmicro/tree": "https://cdn.jsdelivr.net/npm/@stemcmicro/tree@0.9.104/package.json"
    }
}
```

The HTML file for this example has an HTMLInputElement for input and three HTMLDivElement(s) for output.

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <label for="textBox">Expression:</label>
    <input id="textBox" type="text" autocomplete="off" />
    <div><span>Ascii: </span><span id="ascii"></span></div>
    <div><span>Human: </span><span id="human"></span></div>
    <div><span>Infix: </span><span id="infix"></span></div>
    <div><span>LaTeX: </span><span id="latex"></span></div>
    <div><span>SExpr: </span><span id="sexpr"></span></div>
    <div><span>SVG  : </span><span id="svgex"></span></div>
</body>

</html>
```

The TypeScript file for the application simply uses the *stemcmicro* package as a parser and renders the parsed output in various formats.

index.ts

```typescript
import { ExprEngine, create_engine } from '@stemcmicro/engine'
import { js_parse } from '@stemcmicro/js-parse'
const textBox = document.querySelector("#textBox") as HTMLInputElement

const divAscii = document.querySelector("#ascii") as HTMLSpanElement
const divHuman = document.querySelector("#human") as HTMLSpanElement
const divInfix = document.querySelector("#infix") as HTMLSpanElement
const divLaTeX = document.querySelector("#latex") as HTMLSpanElement
const divSExpr = document.querySelector("#sexpr") as HTMLSpanElement
const divSVGEx = document.querySelector("#svgex") as HTMLSpanElement

function keyupListener(this: HTMLInputElement, _ev: KeyboardEvent) {

    divAscii.textContent = ""
    divHuman.textContent = ""
    divInfix.textContent = ""
    divLaTeX.textContent = ""
    divSExpr.textContent = ""
    divSVGEx.textContent = ""

    try {
        const engine: ExprEngine = create_engine()

        const { trees, errors } = js_parse(textBox.value)

        for (const error of errors) {
            divInfix.textContent = `${error}`
            return
        }

        for (const tree of trees) {
            const value = engine.valueOf(tree)
            divAscii.textContent = engine.renderAsString(value, { format: 'Ascii' })
            divHuman.textContent = engine.renderAsString(value, { format: 'Human' })
            divInfix.textContent = engine.renderAsString(value, { format: 'Infix' })
            divLaTeX.textContent = engine.renderAsString(value, { format: 'LaTeX' })
            divSExpr.textContent = engine.renderAsString(value, { format: 'SExpr' })
            divSVGEx.innerHTML = engine.renderAsString(value, { format: 'SVG' })
        }
        engine.release()
    }
    catch (e) {
        console.warn(e)
    }
}

textBox.addEventListener('keyup', keyupListener)

window.onunload = function() {
    textBox.removeEventListener('keyup', keyupListener)
}
```

There is much more that you can do with the *stemcmicro* library.

The NPM page is https://www.npmjs.com/package/stemcmicro

The GitHub pages contain API documentation https://geometryzen.github.io/stemcmicro

3.7. Rendering Mathematics in STEMCstudio

The underlying Web technology that allows STEMCstudio to support the rendering of mathematics is called MathML. This technology is supported by most modern web browsers.

https://www.w3.org/Math

So the most direct way of rendering mathematics is simply to write MathML in your `index.html` file:

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <base href="/">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <h1>MathML Example</h1>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
        <mi>a</mi>
        <msup>
            <mi>x</mi>
            <mn>2</mn>
        </msup>
        <mo>+</mo>
        <mi>b</mi>
        <mi>x</mi>
        <mo>+</mo>
        <mi>c</mi>
    </math>
</body>

</html>
```

The approach above requires no special support from STEMCstudio and no additional libraries because it is built in to all browsers. However, this approach can be tedious and so it is common for authors to express their mathematics to be rendered in a language called `TeX` and have some library convert the `TeX` to HTML. In this book we will look at two popular libraries for converting `TeX`to HTML; MathJax and KaTeX.

3.8. Rendering Mathematics with MathJax

https://www.mathjax.org

Rendering mathematical notation in your application can be done by using *MathJax*. There are two use cases to consider: `auto rendering` and `manual rendering`.

MathJax auto rendering

Auto rendering means that you will define static TeX expressions in your `index.html` file that will be asynchronously processed by MathJax into MathML.

Implementing auto rendering in STEMCstudio is achieved by following the MathJax documentation. This essentially involves including a `script` tag in your `index.html` file to load MathJax

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="style.css">
    <script id="MathJax-script" async src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js">
    </script>
</head>

<body>
    <h1>MathJax auto rendering example</h1>
    <p>
        When \(a \ne 0\), there are two solutions to \(ax^2 + bx + c = 0\) and they are
        \[x = {-b \pm \sqrt{b^2-4ac} \over 2a}.\]
    </p>
</body>

</html>
```

For MathJax auto rendering it is permissable to load the MathJax script asynchronously using the `async` attribute.

This example was taken from the following project:

https://www.stemcstudio.com/gists/db8545fe1cc107dc7c2e7521185df501

MathJax manual rendering

Manual rendering means that you will define dynamic TeX expressions in your `*.ts` files that will be processed by MathJax into MathML.

Implementing MathJax manual rendering in STEMCstudio requires that you trigger MathJax to render expressions on the page using the `typeset` function on the global `MathJax` object.

index.ts

```
const element = document.getElementById('xyz') as HTMLDivElement

element.innerHTML = "When \\(a \\ne 0\\), there are two solutions to \\(ax^2 + bx + c = 0\\) and they are \\[x = {-b \\pm \\sqrt{b^2-4ac} \\over 2a}.\\]"

window['MathJax'].typeset()

export { }
```

In the example above the MathJax global variable is accessed through the global `window` object. This prevents type errors being reported by short-circuiting the type checking.

In this example the loading of the MathJax script must be performed *synchronously* so that the `MathJax` global variable is available when the `typeset` function is called.

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="style.css">
    <script id="MathJax-script" src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js">
    </script>
</head>

<body>
    <h1>MathJax manual rendering example</h1>
    <div id='xyz'></div>
</body>

</html>
```

The `async` attribute is removed from the `script` tag so that the `MathJax` script is loaded synchronously.

This example was taken from the following project:

https://www.stemcstudio.com/gists/8df357eac890105178b6c733704d1c64

3.9. Rendering Mathematics with KateX

KaTeX is mathematical markup rendering library developed by Khan Academy.

https://katex.org

Using KaTeX in STEMCstudio is fairly straightforward. There are two use cases to consider: `auto rendering` and `manual rendering`.

KaTeX auto rendering

In `auto-rendering`, KaTeX is being used to transform TeX syntax that is embedded in your `index.html` document into MathML (Mathematics Markup Language) that can be rendered by the browser. To do this you should use the Auto-render extension as described in https://katex.org/docs/autorender.html

Implementing auto rendering is a matter of following the KaTex auto rendering documentation by adding appropriate `link` and `script` tags to your `index.html` file as follows:

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <base href="/">
    <title></title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/katex@0.16.7/dist/katex.min.css" integrity="sha384-3UiQGuEI4TTMaFmGIZumfRPtfKQ3trwQE2JgosJxCnGmQpL/lJdjpcHkaaFwHlcI" crossorigin="anonymous">
    <script defer src="https://cdn.jsdelivr.net/npm/katex@0.16.7/dist/katex.min.js" integrity="sha384-G0zcxDFp5LWZtDuRMnBkk3EphCK1lhEf4UEyEM693ka574TZGwo4IWwS6QLzM/2t" crossorigin="anonymous"></script>
    <script defer src="https://cdn.jsdelivr.net/npm/katex@0.16.7/dist/contrib/auto-render.min.js" integrity="sha384-+VBxd3r6XgURycqtZ117nYw44OOcIax56Z4dCRWbxyPt0Koah1uHoK0o4+/RRE05" crossorigin="anonymous"></script>
    <script>
        document.addEventListener("DOMContentLoaded", function() {
            renderMathInElement(document.body, {
              // customised options
              // • auto-render specific keys, e.g.:
              delimiters: [
                  {left: '$$', right: '$$', display: true},
                  {left: '$', right: '$', display: false},
                  {left: '\\[', right: '\\]', display: true},
                  {left: '\\(', right: '\\)', display: false}
              ],
              throwOnError : false
            });
        });
    </script>
</head>

<body>
    <p>
        When \(a \ne 0\), there are two solutions to \(ax^2 + bx + c = 0\) and they are
        \[x = {-b \pm \sqrt{b^2-4ac} \over 2a}.\]
    </p>
    <p>1. auto-render with $$ escaping and display:true ...</p>
    <p>$$a=b$$</p>
    <p>2. auto-render with $ escaping and display:false</p>
    <p>$a=b$</p>
    <p>auto-render with [ ] escaping and display:true</p>
    <p>\[a=b\]</p>
    <p>auto-render with ( ) escaping and display:false</p>
    <p>\(a=b\)</p>
</body>

</html>
```

You can try the application at the following URL:

https://www.stemcviewer.com/gists/d36b89fde29ed60774d71f5af64c39cd

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/d36b89fde29ed60774d71f5af64c39cd

KaTeX manual rendering

When rendering some TeX string in your JavaScript code so that it is visible in your HTML page you will need to use the KaTeX API directly. Unfortunately, KaTeX is not distributed in a module format that is compatible with STEMCstudio. This can be resolved by using a wrapper package around the KaTeX page that exposes the code in `system` format. Such a package already exists and is called `@geometryzen/katex`.

Implementing KaTeX manual rendering in your application involes the following steps.

1. Add `katex` as a dependency to `package.json`
2. Create an override in `studio.config.json` to replace the runtime with `@geometryzen/katex`.
3. Define the element in your HTML where you want to output MathML.
4. Write the code to render your TeX string into the HTML element.

Here is what that looks like in the various files:

package.json

```json
{
    "description": "KaTeX demo",
    "dependencies": {
        "katex": "0.16.11"
    },
    "name": "katex-demo",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "private": true
}
```

studio.config.json

```json
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [],
    "references": {},
    "showGeneratedFiles": true
}
```

You only need to override the `system` property that affects the runtime. STEMCstudio is able to find TypeScript type definitions for KaTeX in the package `@types/katex`.

You can find a version of `@geometryzen/katex` to use from the https://www.npmjs.com site.

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <base href="/">
    <title></title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <p>This expression is rendered from index.ts</p>
    <div id="katex"></div>
</body>

</html>
```

index.ts

```typescript
import { render } from 'katex'

const element = document.getElementById('katex') as HTMLElement

render("c = \\pm\\sqrt{a^2 + b^2}", element, {
    displayMode: true,
    output: 'mathml',
    throwOnError: false
})
```

You can try the application at the following URL:

https://www.stemcviewer.com/gists/005b55eee6797297dd7b056597726f12

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/005b55eee6797297dd7b056597726f12

KaTeX manual rendering a third way

KaTeX is an example of a library that does not expose a granular API. This makes it feasible to entirely encapsulate the KaTeX API using a library that defines its own API. We haven't done this in the case of KaTeX because the TypeScript type definitions work quite well with STEMCstudio and have good JSDoc comments which futher enhances the STEMCstudio developer experience.

3.10. Code Editing using monaco-editor

monaco-editor is a browser based code editor.

You can try the application at the following URL:

https://www.stemcviewer.com/gists/1b9ac72f9381d7ccf68add45f147aa58

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/1b9ac72f9381d7ccf68add45f147aa58

The first step is to add *monaco-editor* as a dependency to your project. This can either be done by using the *Add Dependency* button in the *Explorer* toolbar, or by editing the *package.json* file directly. The end result should be the same and the *package.json* file will be as follows:

package.json

```
{
    "description": "monaco-editor",
    "dependencies": {
        "monaco-editor": "0.50.0"
    },
    "name": "monaco-editor",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "private": true,
    "keywords": [
        "monaco",
        "editor",
        "STEMCstudio"
    ]
}
```

The purpose of adding the dependency is to get access to the *monaco.d.ts* type definitions. We'll need an override in *studio.config.json* to tell STEMCstudio where to find the *monaco.d.ts* file.

studio.config.json

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "references": {},
    "showGeneratedFiles": true,
    "overrides": [
        {
            "name": "monaco-editor",
            "version": "0.50.0",
            "types": "https://cdn.jsdelivr.net/npm/monaco-editor@0.50.0/monaco.d.ts"
        }
    ]
}
```

We now have to load the runtime for the *monaco-editor*. This is done in our *index.html* file.

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <base href="/">
    <title></title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <script src="https://cdn.jsdelivr.net/npm/monaco-editor@0.50.0/min/vs/loader.js"></script>
    <script>
        require.config({
            paths: {
                vs: 'https://cdn.jsdelivr.net/npm/monaco-editor@0.50.0/min/vs'
            }
        });

        require(['vs/editor/editor.main'], function() {
            System.defaultJSExtensions = true
            System.import('./index.js').catch(function(e) {
                console.error(e)
            })
        });
    </script>
    <div id="container" style="width: 600px; height: 400px; border: 1px solid grey"></div>
    <button id='btn'>Press Me</button>
    <button id='btn-reset'>Reset</button>
</body>

</html>
```

An important detail in the *index.html* file is that we have taken control of the loading of our *index.js* file using *System.import*. This ensures that the *index.js* file loads only after the *monaco-editor* global variable 'monaco' has been set.

Finally, we are ready in *index.ts* to create the editor...

index.ts

```typescript
import { completions } from './completions'

const code = [
    `const greeting = "Hello, World"`,
    `console.log(greeting)`
].join('\n')

const editor = monaco.editor.create(document.getElementById('container')!, {
    value: code,
    language: 'javascript'
})

monaco.editor.defineTheme("myTheme", {
    base: "vs",
    inherit: true,
    colors: {
        "editor.foreground": "#000000",
        "editor.background": "#FFFFFF"
    },
    rules: [
        { foreground: "#000000", token: "" }
    ]
})

monaco.editor.setTheme("myTheme")

// Syntax Highlighting using Monarch
monaco.languages.setMonarchTokensProvider("stemcmicro", {
    tokenizer: {
        // etc
    }
})

monaco.languages.typescript.javascriptDefaults.setDiagnosticsOptions({
    noSemanticValidation: false,
    noSyntaxValidation: false
})

// Disable default autocompletion for javascript
// Unfortunately this seems to kill the hover over support.
/*
monaco.languages.typescript.javascriptDefaults.setCompilerOptions({
    noLib: false
})
*/

// What's the best way to get some type
const extraLib = monaco.languages.typescript.javascriptDefaults.addExtraLib('')

```

```
// Auto Completion and Documentation.
monaco.languages.registerCompletionItemProvider('typescript', {
    // triggerCharacters: ["$"],
    provideCompletionItems: completions
})

// Keyboard Shortcuts
editor.addCommand(monaco.KeyMod.CtrlCmd | monaco.KeyCode.KeyS, () => {
    const action = editor.getAction("editor.action.formatDocument")
    if (action) {
        action.run()
    }
})

editor.addCommand(monaco.KeyMod.CtrlCmd | monaco.KeyCode.KeyR, () => {
    // Knock yourself out...
})

const btn = document.getElementById('btn') as HTMLButtonElement

btn.onclick = function() {
    alert(editor.getValue())
}

const btnReset = document.getElementById('btn-reset') as HTMLButtonElement

btnReset.onclick = function() {
    editor.setValue(code)
}

window.onunload = function() {
    editor.dispose()
    extraLib.dispose()
}

monaco.languages.typescript.javascriptDefaults.setDiagnosticsOptions({
    noSemanticValidation: true,
    noSyntaxValidation: true
})

monaco.languages.registerDocumentFormattingEditProvider('javascript', {
    async provideDocumentFormattingEdits(model, _options, _token) {

        return [
            {
                range: model.getFullModelRange(),
                text: "Formatted..."
            }
        ]
    }
})
```

```

// This is important. It causes this file to be loaded as a module.
export { }
```

An important detail in this file is the *export* at the bottom that ensures that the *index.ts* file is loaded as an EcmaScript module.

You should find that *IntelliSense* is working and that the 'monaco' global variable is recognized as a *namespace*.

Congratulations! You now have *monaco-editor* working in your STEMCstudio application.

For more information on *monaco-editor*, please visit the Microsoft GitHub repository:

https://github.com/microsoft/monaco-editor?tab=readme-ov-file

3.11. LMS Gradebook Integration

You can try the application at the following URL:

https://www.stemcviewer.com/gists/7378bd81ed8c8601d378f52c7ccb22fe

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/7378bd81ed8c8601d378f52c7ccb22fe

system.config.json

```json
{
    "map": {
        "stemcstudio-tunnel": "https://cdn.jsdelivr.net/npm/stemcstudio-tunnel@1.2.3/package.json",
        "@microsoft/fast-components": "https://cdn.jsdelivr.net/npm/@geometryzen/fast-components@0.9.6/dist/system/index.min.js",
        "@microsoft/fast-element": "https://cdn.jsdelivr.net/npm/@geometryzen/fast-element@0.9.9/dist/system/index.min.js",
        "@microsoft/fast-foundation": "https://cdn.jsdelivr.net/npm/@geometryzen/fast-foundation@0.9.6/dist/system/index.min.js"
    }
}
```

types.config.json

```json
{
    "map": {
        "stemcstudio-tunnel": "https://cdn.jsdelivr.net/npm/stemcstudio-tunnel@1.2.3/package.json",
        "@microsoft/fast-components": "https://cdn.jsdelivr.net/npm/@microsoft/fast-
components@2.30.6/package.json",
        "@microsoft/fast-element": "https://cdn.jsdelivr.net/npm/@microsoft/fast-element@1.12.0/package.json",
        "@microsoft/fast-foundation": "https://cdn.jsdelivr.net/npm/@microsoft/fast-foundation@2.49.0/package.json"
    }
}
```

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id="container" class="container">
        <div class="header">
            <h2></h2>
            <fast-switch id="toggle">
                Luminance
                <span slot="checked-message">Light</span>
                <span slot="unchecked-message">Dark</span>
            </fast-switch>
        </div>
        <div class="controls" id="controls">
            <fast-number-field id='score-given' autocomplete='off' min="0">Score Given</fast-number-field>
            <br />
            <fast-number-field type='text' id='score-maximum' autocomplete='off' min="0">Score Maximum</fast-number-field>
            <br />
            <!--label for="activity-progress">Activity Progress:</label-->
            <fast-select name="Activity Progress" id="activity-progress">
                <fast-option value="Completed" selected>Completed</fast-option>
                <fast-option value="Initialized">Initialized</fast-option>
                <fast-option value="InProgress">InProgress</fast-option>
                <fast-option value="Started">Started</fast-option>
                <fast-option value="Submitted">Submitted</fast-option>
            </fast-select>
            <br />
            <!--label for="grading-progress">Grading Progress:</label-->
            <fast-select name="Grading Progress" id="grading-progress">
                <fast-option value="Failed">Failed</fast-option>
                <fast-option value="FullyGraded" selected>Fully Graded</fast-option>
                <fast-option value="Pending">Pending</fast-option>
                <fast-option value="PendingManual">Pending Manual</fast-option>
                <fast-option value="NotReady">Not Ready</fast-option>
            </fast-select>
            <br />
            <fast-text-field id='comment' autocomplete='off'>Comment</fast-text-field>
            <br />
            <fast-button id='btn-submit-answer'>Submit</fast-button>
            <pre id="results"></pre>
        </div>
    </div>
</body>

</html>
```

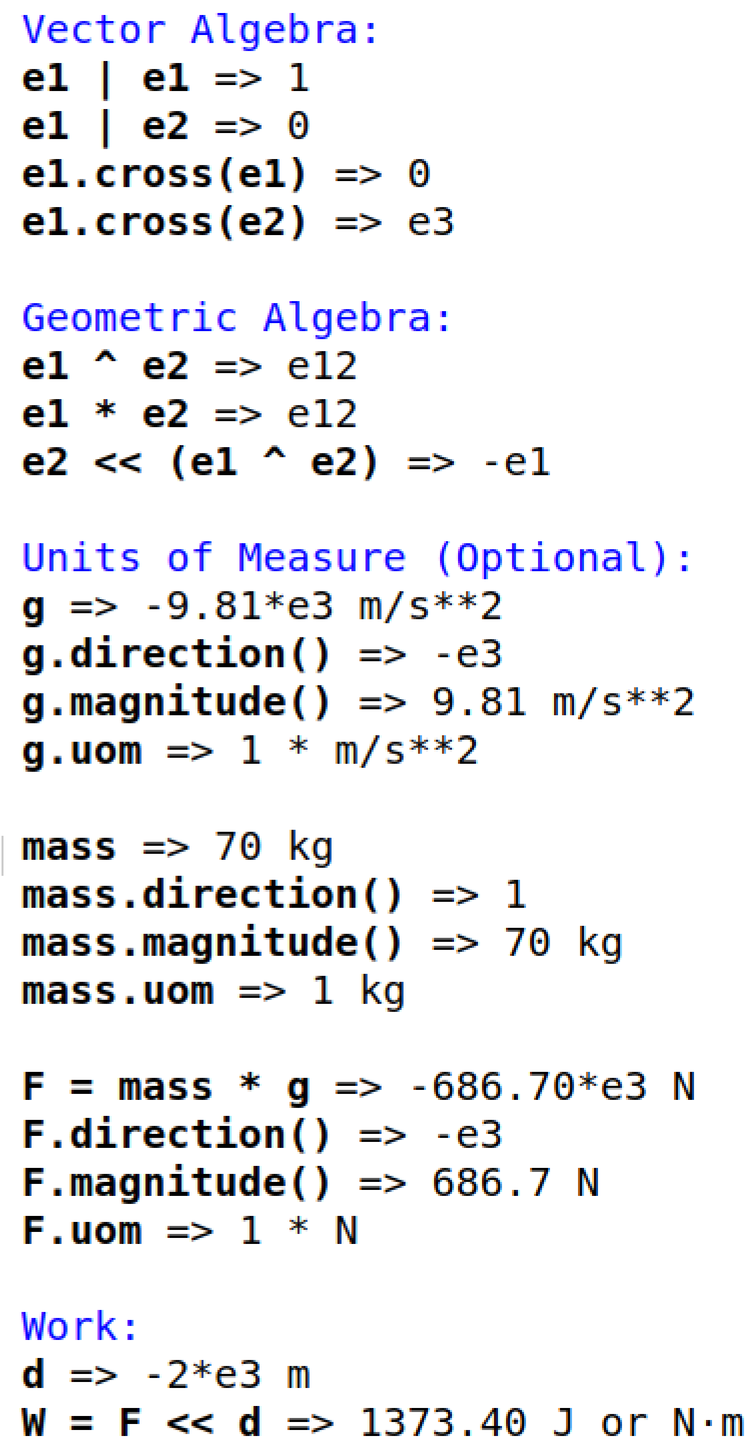
index.ts

```typescript
import { gradebook, Item, Score, user } from 'stemcstudio-tunnel'

import {
    allComponents,
    baseLayerLuminance,
    Button,
    NumberField,
    provideFASTDesignSystem,
    Select,
    StandardLuminance,
    Switch,
    TextField
} from "@microsoft/fast-components"

// We are not using a tree-shaking bundler,
// so there is no advantage in registering individual components.
provideFASTDesignSystem(document.body).register(allComponents)

const toggle = document.getElementById('toggle') as Switch

const toggleLightMode = function() {
    baseLayerLuminance.setValueFor(
        document.body,
        toggle.checked ? StandardLuminance.LightMode : StandardLuminance.DarkMode
    )
}

toggle.addEventListener('click', toggleLightMode)

let submit: Button
let txtScoreGiven: NumberField
let txtScoreMaximum: NumberField
let selActivity: Select
let selGrading: Select
let txtComment: TextField

const gradebookItems: Item[] = []

function scoreFromUI(): Score {
    const score: Score = {
        activityProgress: selActivity.value as any,
        gradingProgress: selGrading.value as any
    }
    const scoreGiven = parseInt(txtScoreGiven.value, 10)
    if (scoreGiven) {
        score.scoreGiven = scoreGiven
    }
    const scoreMaximum = parseInt(txtScoreMaximum.value, 10)
    if (scoreMaximum) {
        score.scoreMaximum = scoreMaximum
    }
    const comment = txtComment.value
    if (comment) {
        score.comment = comment
    }
    return score
}

// Notice that scores are only submitted for a given Gradebook Item.
async function submitScore() {
    console.log()
    try {
        if (gradebookItems.length > 0) {
            const score: Score = scoreFromUI()
            console.log(`${JSON.stringify(score, null, 2)}`)
            try {
                await gradebook.submitScore(gradebookItems[0].id, score)
                user.alert({ title: 'Submit', message: 'Successfully submitted.' })
                updateResults()
            } catch (e) {
                user.alert({ title: 'Submit', message: `Something went wrong with your submission. Cause; ${e}` })
            }
        }
        else {
            console.warn('No items in the Gradebook')
        }
    } catch (e) {
        console.warn(e)
    }
}

```

```
async function updateResults() {
    const results = await gradebook.getResults(gradebookItems[0].id)
    const divResults = document.getElementById('results') as HTMLDivElement
    divResults.textContent = JSON.stringify(results, null, 2)
}

async function ready() {
    // This is the recommended prolog for an app using the Gradebook...
    // Ensure that the appropriate Gradebook Items are defined at startup.
    // In a real LMS there will always be a Gradebook Item for the activity.
    gradebookItems.length = 0
    const items = await gradebook.getItems()
    if (items.length === 0) {
        // This code will only run when simulating the LMS...
        // Note that the gradebook items in simulation mode are cached in Session Storage.
        const item = await gradebook.createItem({ scoreMaximum: 10, label: 'LTI Tester' })
        gradebookItems.push(item)
    }
    else {
        for (const item of items) {
            gradebookItems.push(item)
        }
    }
    updateResults()
    submit = document.getElementById('btn-submit-answer') as Button
    submit.addEventListener('click', submitScore)
    txtScoreGiven = document.getElementById('score-given') as NumberField
    txtScoreMaximum = document.getElementById('score-maximum') as NumberField
    selActivity = document.getElementById('activity-progress') as Select
    selGrading = document.getElementById('grading-progress') as Select
    txtComment = document.getElementById('comment') as TextField
}

ready()

window.onunload = function() {
    submit.removeEventListener('click', submitScore)
    toggle.removeEventListener('click', toggleLightMode)
}

export { }
```

3.12. Numeric Vector and Geometric Algebra

Manipulating geometric quantities such as scalars and vectors in a coordinate-free manner is a rite of passage for a student and an extremely useful technique, especially in the age of computers where such computations can be performed at speed, flawlessly, and with economical expression in code.



```
Vector Algebra:
e1 | e1 => 1
e1 | e2 => 0
e1.cross(e1) => 0
e1.cross(e2) => e3

Geometric Algebra:
e1 ^ e2 => e12
e1 * e2 => e12
e2 << (e1 ^ e2) => -e1

Units of Measure (Optional):
g => -9.81*e3 m/s**2
g.direction() => -e3
g.magnitude() => 9.81 m/s**2
g.uom => 1 * m/s**2

mass => 70 kg
mass.direction() => 1
mass.magnitude() => 70 kg
mass.uom => 1 kg

F = mass * g => -686.70*e3 N
F.direction() => -e3
F.magnitude() => 686.7 N
F.uom => 1 * N

Work:
d => -2*e3 m
W = F << d => 1373.40 J or N·m
```

Figure 26. Multivectors

You can try the application at the following URL:

https://www.stemcviewer.com/gists/6d337555572454c211182c5b45aed418

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/6d337555572454c211182c5b45aed418

This example uses the '@geometryzen/multivectors' package to provide the computations:

package.json

```json
{
    "description": "Units of Measure",
    "dependencies": {
        "@geometryzen/multivectors": "0.9.11"
    },
    "name": "uom",
    "version": "1.0.0",
    "keywords": [
        "S.I. Units",
        "base",
        "UNITS",
        "Dimensions",
        "davinci-newton",
        "NEWTON",
        "Geometric3",
        "meter",
        "kilogram",
        "second",
        "ampere",
        "STEMCstudio",
        "stemcbook"
    ],
    "author": "David Geo Holmes"
}
```

The '@geometryzen/multivectors' package provides ideal resources for STEMCstudio and so no overrides are required in the `studio.config.json` file. STEMCstudio generates the following entries in `system.config.json` and `types.config.json`:

system.config.json

```json
{
    "map": {
        "@geometryzen/multivectors": "https://cdn.jsdelivr.net/npm/@geometryzen/multivectors@0.9.11/package.json"
    }
}
```

types.config.json

```json
{
    "map": {
        "@geometryzen/multivectors": "https://cdn.jsdelivr.net/npm/@geometryzen/multivectors@0.9.11/package.json"
    }
}
```

The 'index.html' file defines a 'pre' (preserve) element in which to place the result.

index.html

```html
<!doctype html>
<html>

<head>
    <base href='/'>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <pre id='info'></pre>
</body>

</html>
```

The 'index.ts' makes various calulations and prints the results. Note the (optional) use of operator overloading.

index.ts

```typescript
import { Geometric3 } from '@geometryzen/multivectors'
import { blue } from './colors'
import { println, printvar } from './display'

const e1 = Geometric3.e1
const e2 = Geometric3.e2
const e3 = Geometric3.e3
const newton = Geometric3.newton
const kilogram = Geometric3.kilogram
const meter = Geometric3.meter

const g = -9.81 * e3 * newton / kilogram
const mass = 70 * kilogram
const F = mass * g
const d = -2 * e3 * meter
const W = F << d

println('Vector Algebra:', blue)
printvar('e1 | e1', e1 | e1)                    // 1
printvar('e1 | e2', e1 | e2)                    // 0
printvar('e1.cross(e1)', e1.cross(e1))          // 0
printvar('e1.cross(e2)', e1.cross(e2))          // e3
println('')
println('Geometric Algebra:', blue)
printvar('e1 ^ e2', e1 ^ e2)                    // e12
printvar('e1 * e2', e1 * e2)                    // e12
printvar('e2 << (e1 ^ e2)', e2 << (e1 ^ e2))    // -e1
println('')
println('Units of Measure (Optional):', blue)
printvar('g', g)                                // -9.81*e3 m/s**2
printvar('g.direction()', g.direction())        // -e3
printvar('g.magnitude()', g.magnitude())        // 9.81 m/s**2
printvar('g.uom', g.uom)                        // 1 * m/s**2
println('')
printvar('mass', mass)                          // 70 kg
printvar('mass.direction()', mass.direction())  // 1
printvar('mass.magnitude()', mass.magnitude())  // 70 kg
printvar('mass.uom', mass.uom)                  // 1 kg
println('')
printvar('F = mass * g', F.toFixed(2))          // -686.70*e3 N
printvar('F.direction()', F.direction())        // -e3
printvar('F.magnitude()', F.magnitude())        // 686.7 N
printvar('F.uom', F.uom)                        // 1 * N
println('')
println('Work:', blue)
printvar('d', d)                                // -2*e3 m
printvar('W = F << d', W.toPrecision(6))        // 1373.40 J or N·m
```

3.13. Simulations and the Physics Engine

This example provides an accurate simulation of elastic collisions. It uses a physics engine to compute the motion of colliding blocks. The custom collision handling code detects when a collision occurs and backtracks the physics engine to the collision point, applies the momenta exchanges, and then advances the simulation. The JsxGraph package is used to render the simulation, and JsxGraph is wrapped by another external package as blocks with a visual representation and a model that comes from the physics engine.

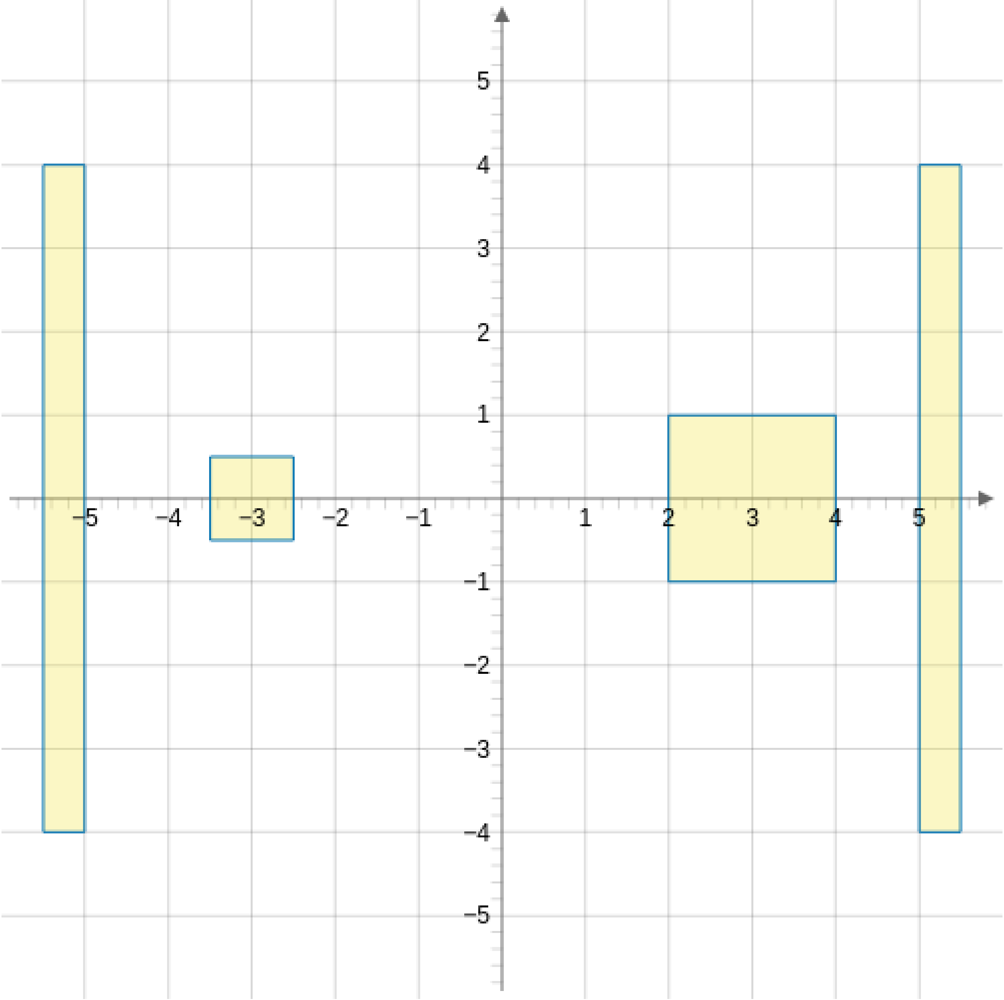


Figure 27. Collision Handling

You can try the application at the following URL:

https://www.stemcviewer.com/gists/0436144a07ae10e84a8619a17c4cb4ee

The code for this example can be found at the following URL:

https://www.stemcstudio.com/gists/0436144a07ae10e84a8619a17c4cb4ee

system.config.json

```
{
    "map": {
        "@geometryzen/multivectors":
"https://cdn.jsdelivr.net/npm/@geometryzen/multivectors@0.9.11/package.json",
        "@geometryzen/newton":
"https://cdn.jsdelivr.net/npm/@geometryzen/newton@0.9.2/package.json",
        "jsxgraph":
"https://cdn.jsdelivr.net/npm/jsxgraph@1.10.1/distrib/jsxgraphcore.js",
        "@geometryzen/newton-jsxgraph-widgets":
"https://cdn.jsdelivr.net/npm/@geometryzen/newton-jsxgraph-widgets@0.9.1/package.json"
    }
}
```

types.config.json

```json
{
    "map": {
        "@geometryzen/multivectors": "https://cdn.jsdelivr.net/npm/@geometryzen/multivectors@0.9.11/package.json",
        "@geometryzen/newton": "https://cdn.jsdelivr.net/npm/@geometryzen/newton@0.9.2/package.json",
        "jsxgraph": "https://cdn.jsdelivr.net/npm/jsxgraph@1.10.1/package.json",
        "@geometryzen/newton-jsxgraph-widgets": "https://cdn.jsdelivr.net/npm/@geometryzen/newton-jsxgraph-widgets@0.9.1/package.json"
    }
}
```

index.html

```html
<!DOCTYPE html>
<html lang='en'>

<head>
    <meta charset="UTF-8">
    <title>JSXGraph template</title>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
    <base href='/'>
    <link rel='stylesheet' href="https://cdn.jsdelivr.net/npm/jsxgraph@1.10.0/distrib/jsxgraph.css" />
    <link rel='stylesheet' href="style.css" />
</head>

<body>
    <div id='jxgbox' class='jxgbox' style='width:500px; height:500px'></div>
</body>

</html>
```

index.ts

```typescript
import {
    Block2,
    Engine2,
    Geometric2
} from '@geometryzen/newton'
import { JsxBlock } from '@geometryzen/newton-jsxgraph-widgets'
import { JSXGraph } from 'jsxgraph'
import { interactBodies } from './interactBodies'
import { writeToDOM } from './writeToDOM'

const e1 = Geometric2.e1
const kg = Geometric2.kilogram
const m = Geometric2.meter
const s = Geometric2.second

const sim = new Engine2()
const Δt = 0.01 * s

const bodies: Block2[] = []
const width = m
const height = m
const blockA = new Block2(width, height)
bodies.push(blockA)
const blockB = new Block2(width * 2, height * 2)
bodies.push(blockB)
const wallL = new Block2(width * 0.5, height * 8)
bodies.push(wallL)
const wallR = new Block2(width * 0.5, height * 8)
bodies.push(wallR)

blockA.M = 1 * kg
blockA.X = (-3 * e1) * m
blockA.P = 10 * e1 * kg * m / s

blockB.M = 4 * kg
blockB.X = (0 * e1) * m
blockB.X = 0 * m // The direction is not important when the value is zero but the units are.

wallL.M = 100000000 * kg
wallL.X = (-5.25 * e1) * m

wallR.M = 100000000 * kg
wallR.X = (5.25 * e1) * m

sim.addBody(blockA)
sim.addBody(blockB)
sim.addBody(wallL)
sim.addBody(wallR)

const board = JSXGraph.initBoard('jxgbox', {
    axis: true,
    boundingBox: [-6, 6, 6, -6],
    showCopyright: true,
    showNavigation: false,
    showFullscreen: false,
    showScreenshot: true
})

// The widgets (JsxBlock) are coupled rather tightly to the Physics Engine package.
// In this case the JsxBlock gets its dimensions and position from the Block2.
// It would be better to have a decoupled user interface.
const viewA = new JsxBlock(board, blockA)
const viewB = new JsxBlock(board, blockB)
const viewL = new JsxBlock(board, wallL)
const viewR = new JsxBlock(board, wallR)

const animation = function() {
    try {
        interactBodies(bodies, sim)
        sim.advance(Δt.a, Δt.uom)
        viewA.update()
        viewB.update()
        viewL.update()
        viewR.update()
        board.update()
        // Keep the animation going.
        window.requestAnimationFrame(animation)
    }
    catch (e) {
        writeToDOM(e)
    }
}
// Prime the pump...
window.requestAnimationFrame(animation)
```

3.14. Summary

We have been introduced to some useful libraries that will make it possible to produce powerful STEM Learning Activities with less code.

Chapter 4. Application Frameworks

This chapter will be about how to consider whether or not to use an application framework, what choices are available, and what is recommended.

4.1. What is a Web Application Framework?

A Web Application Framework provides an alternative way to interact and update the DOM that may streamline the developer experience of DOM updating and may also encourage a more modular approach to assembling a web application. Increasingly, modern frameworks will use an alternate representation for the DOM from pure HTML and will require a compiler to convert the developer source code into JavaScript for execution.

4.2. Nothing

It's not the name of the latest JavaScript application framework. What I mean by this is that you code directly to the Document Object Model (DOM) in order to create your dynamic user interface.

This will work OK for very small applications but is likely to become tedious and repetitive for size and quality of applications that we are trying to achieve.

It is a good approach to take if you are just getting started with STEMCstudio and are new to web development.

An excellent resource for learning web development is *https://developer.mozilla.org/en-US/docs/Learn*.

4.3. Web Components

Again, not a framework, but a very attractive way to build large-grained components. When these components are deployed in external packages there is considerable opportunity for reuse and application simplification.

4.4. React

https://react.dev

Introduction

React is a popular approach to building web user interfaces. This section describes the steps you must take to use React in STEMCstudio. It is not a general React tutorial.

How it Works

React is described as a library for building user interfaces. It provides an excellent Developer Experience (DX) because of its technical underpinnings; React uses the *jsx* standard as the file format which is an extension of JavaScript to support HTML creating through the DOM. Furthermore, this format is understood by TypeScript and extended to give a *tsx* standard, which is *jsx_* with TypeScript type annotations. What this amounts to is that the STEMCstudio development environment is able to give you lots of help in authoring your user interface code.

Getting Started

The package dependencies required for running React in STEMCstudio are `react`, `react-dom`, `csstype`, and `prop-types`.

```
{
    "description": "Modern React Template",
    "dependencies": {
        "csstype": "3.1.3",
        "prop-types": "15.8.1",
        "react": "18.3.1",
        "react-dom": "18.3.1"
    },
    "name": "modern-react-template",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "private": true,
    "keywords": [
        "React",
        "function"
    ]
}
```

In addition to adding the dependencies to your *package.json* file, it will be necessary to define overrides in *studio.config.json*:

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "csstype",
            "version": "3.1.2",
            "system": "https://cdn.jsdelivr.net/npm/csstype@3.1.2/package.json",
            "types": "https://cdn.jsdelivr.net/npm/csstype@3.1.2/package.json"
        },
        {
            "name": "prop-types",
            "version": "15.8.1",
            "system": "https://cdn.jsdelivr.net/npm/prop-types@15.8.1/umd/prop-types.js",
            "types": "https://cdn.jsdelivr.net/npm/@types/prop-types@15.7.5/package.json"
        },
        {
            "name": "react",
            "version": "18.3.1",
            "system": "https://cdn.jsdelivr.net/npm/react@18.3.1/umd/react.development.js",
            "types": "https://cdn.jsdelivr.net/npm/@types/react@18.3.3/package.json"
        },
        {
            "name": "react-dom",
            "version": "18.3.1",
            "system": "https://cdn.jsdelivr.net/npm/react-dom@18.3.1/umd/react-dom-development.js",
            "types": "https://cdn.jsdelivr.net/npm/@types/react-dom@18.0.10/package.json"
        }
    ],
    "references": {},
    "showGeneratedFiles": true
}
```

Finally the `jsx` property in *tsconfig.json* should be configured to compile JavaScript extensions for `react`.

```
{
    "allowJs": false,
    "allowSyntheticDefaultImports": true,
    "declaration": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "jsx": "react",
    "module": "system",
    "noImplicitAny": true,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "preserveConstEnums": true,
    "removeComments": true,
    "sourceMap": false,
    "strict": true,
    "strictNullChecks": true,
    "suppressImplicitAnyIndexErrors": true,
    "target": "esnext",
    "traceResolution": true
}
```

You are now ready to create your STEMCstudio application using the React framework.

The boilerplate code for bootstrapping your application is in the *index.tsx* file.

```
import * as React from 'react'
import { createRoot } from 'react-dom/client'
import { App } from './App'

const container = document.getElementById('root')
if (container) {
    const root = createRoot(container)
    root.render(<App message="World" />)

    window.onunload = function() {
        root.unmount()
    }
}
```

The *index.html* file contains an `HTMLDivElement`, which is the mounting point for the `App`.

```
<!DOCTYPE html>
<html>

<head>
    <base href='/'>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel='stylesheet' href='style.css'>
</head>

<body>
    <div id='root'></div>
</body>

</html>
```

The *App.tsx* file contains the top-level component.

In this example, `App` is a function component, which is the modern and recommended way to implement React applications. The legacy approach was to extends a base class `Component` that is provided by the `react` package.

```
import * as React from 'react'
import { CSSProperties, FunctionComponent, ReactNode } from 'react'
import { MyButton } from './MyButton'

interface AppProps {
    message: string,
    style?: CSSProperties,
    children?: ReactNode
}

export const App: FunctionComponent<AppProps> = (props) => {
    return (
        <div>
            <h1 style={props.style}>Hello, {props.message ?? "World"}!</h1>
            <MyButton />
            <div>{props.children}</div>
        </div>
    )
}
```

4.5. SolidJS

https://solidjs.com

Introduction

SolidJS is a relative newcomer to the application framework space.

How it Works

SolidJS uses a paradigm known as *signals* to propagate changes in values to become DOM updates. It also makes use of JavaScript extensions (*jsx*) for the authoring of HTML components. The approach is described by the author as *fine-grained reactivity* with the goal of high performance.

Getting Started

A template exists if you are creating a new project. What follows is a description of the unique SolidJS features of that the template, or how to modify your project so that it supports SolidJS.

The package dependencies required for running SolidJS in STEMCstudio are `solid-js`, and `csstype`.

```
{
    "description": "SolidJS Template",
    "dependencies": {
        "csstype": "3.1.3",
        "solid-js": "1.8.17"
    },
    "name": "solid-js-template",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "private": true,
    "keywords": [
        "Solid",
        "JS",
        "Reactive",
        "JavaScript",
        "JSX",
        "TSX"
    ]
}
```

In addition to adding the dependencies to your *package.json* file, it will be necessary to define overrides in *studio.config.json*:

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "csstype",
            "version": "3.1.3",
            "system": "https://cdn.jsdelivr.net/npm/csstype@3.1.3/package.json",
            "types": "https://cdn.jsdelivr.net/npm/csstype@3.1.3/package.json"
        },
        {
            "name": "solid-js",
            "version": "1.8.17",
            "system": "https://cdn.jsdelivr.net/npm/@geometryzen/solid-js@1.8.17/package.json",
            "types": "https://cdn.jsdelivr.net/npm/solid-js@1.8.17/package.json"
        }
    ],
    "references": {},
    "showGeneratedFiles": true
}
```

Finally the `jsx` and `jsxImportSource` properties in *tsconfig.json* should be configured to compile JavaScript extensions appropriate for SolidJS.

```
{
    "allowJs": true,
    "allowUnreachableCode": false,
    "checkJs": false,
    "declaration": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "forceConsistentCasingInFileNames": true,
    "jsx": "preserve",
    "jsxImportSource": "solid-js",
    "module": "system",
    "noImplicitAny": true,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "preserveConstEnums": true,
    "removeComments": false,
    "skipLibCheck": true,
    "sourceMap": false,
    "strict": true,
    "strictNullChecks": true,
    "suppressImplicitAnyIndexErrors": true,
    "target": "esnext",
    "traceResolution": true
}
```

You are now ready to create your STEMCstudio application using the SolidJS framework.

The boilerplate code for bootstrapping your application is in the *index.tsx* file.

```
import { render } from "solid-js/web"
import { App } from "./App.js"

const cleanup: () => void = render(() => <App />, document.getElementById('app')!)

window.onunload = function() {
    cleanup()
}
```

The *index.html* file contains an `HTMLDivElement`, which is the mounting point for the `App`.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
```

```
</head>

<body>
    <div id="app"></div>
</body>

</html>
```

```
import type { JSX } from 'solid-js'

export function App(): JSX.Element {
    return <h1>Hello, World!</h1>
}
```

4.6. Svelte

https://svelte.dev

Introduction

Svelte is relative newcomer to the web application framework space.

> ⚠ The Svelte implementation in STEMCstudio is experimental. Issues can happen if the version of Svelte used to compile your `*.svelte` files does not match the version of the Svelte dependency in your project. This should not affect any projects that have been compiled and deployed. For now, only the `3.59.1` version of Svelte is supported.

How it Works

Svelte uses its own proprietary *svelte* format to define the user interface. Svelte employs a compiler to convert the propietary format into executable JavaScript. The proprietary *svelte* format looks like HTML with embedded JavaScript in `script` tags and is designed to reduce the amount of code needed to create fragments of HTML, but it is at the expense of less powerful tooling. To date, there is no first-class parser and analyzer for *svelte* files.

Getting Started

A template exists if you are creating a new project. What follows is a description of the unique Svelte features of that the template, or how to modify your project so that it supports Svelte.

The package dependencies required for running Svelte in STEMCstudio are `svelte`.

```json
{
    "description": "Svelte 5 Template",
    "dependencies": {
        "svelte": "5.0.0-next.142"
    },
    "name": "svelte-template",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "keywords": [
        "Svelte",
        "STEMCstudio"
    ],
    "private": true
}
```

In addition to adding the dependencies to your *package.json* file, it will be necessary to define overrides in *studio.config.json*:

```json
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "svelte",
            "version": "5.0.0-next.142",
            "system": "https://cdn.jsdelivr.net/npm/@geometryzen/svelte@5.0.0-next.142/package.json",
            "types": "https://cdn.jsdelivr.net/npm/svelte@5.0.0-next.142/package.json"
        }
    ],
    "references": {},
    "showGeneratedFiles": true
}
```

You are now ready to create your STEMCstudio application using the SolidJS framework.

The boilerplate code for bootstrapping your application is in the *index.ts* file.

```typescript
import { mount, unmount } from 'svelte'
import App from './App.svelte'

const app = mount(App as any, {
    target: document.getElementById("app")!,
    props: {}
})

window.onunload = function() {
    unmount(app)
}

export default app
```

The *index.html* file contains an `HTMLDivElement`, which is the mounting point for the `App`.

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <base href="/">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/modern-css-reset/dist/reset.min.css" />
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id="app"></div>
</body>

</html>
```

Components are created their own files and must have the extension `.svelte`

```
<script>
    import {
        onMount,
        onDestroy
    } from 'svelte'

    let name = $state("Svelte")
    let count = $state(0)

    function increment() {
        count += 1
    }

    let double = $derived(count * 2)

    onMount(() => {
        // console.log("App.onMount()")
    });

    onDestroy(() => {
        // console.log("App.onDestroy()")
    });
</script>

<h1>Hello, {name}!</h1>

<button on:click={increment}>
    clicks: {count}
</button>

<p>Double that is {double}</p>

<style>
h1 {
    color: blue;
}
</style>
```

4.7. Summary

STEMCstudio applications can built with any combination of the supported application frameworks (React, SolidJS, and Svelte), Web Components, and native DOM APIs.

Appendix A: Authoring JavaScript Libraries

This chapter will provide a concrete, step-by-step, example of how to author a modern JavaScript library. Nothing in this chapter is specific to STEMCstudio. However, a library incorporating modern best practices will provide the best developer experience for someone using your library.

A.1. Best Practices

This section describes the features that are considered best practices in modern JavaScript libraries.

Smart Editor Support using Type definitions

We would like the consumer of the library to be supported with smart editor features such as type checking and autocompletion. We will accomplish this by shipping a type definition file, `index.d.ts` with our library that contains TypeScript type definitions for our JavaScript library. Furthermore, we will generate this file so that it always stays synchronized with the implementation code.

Documentation Generation

We would like to provide the consumer of the library with web pages that describe the consumption of the library in detail. We will automatically generate the documentation from the code and make it available as GitHub pages.

Comprehension depends on Implementation Language

We would like the maintainers of the library to be able to easily understand the implementation code whether they are the original author or not, and even if some time has elapsed since the code was written. The best practice for doing this is to use TypeScript rather than JavaScript as the implementation language. TypeScript is easily learned because it is simply JavaScript with additional type information sprinkled on top. Utilities exist for automatically stripping the type information to reveal the executable JavaScript code.

Reliability aided by Automated Testing

We would like the maintainers of the code to be able to easily run test suites to verify the integrity of the code. Our test runner of choice will be `mocha` because it is agnostic as to the assertion framework. We will use `chai` as the assertion framework and library because it provides a choice of several assertion styles. We will also add coverage testing so that we can see what has been tested.

Module Formats and Code Bundling

There is a gradual move to composing libraries from EcmaScript modules (ESM), and often with many files. While this provides opportunity for tree-shaking in applications, some consumers will benefit from bundled code in particular formats. We will offer several formats; `ESM`, `UMD`, and `System`. STEMCstudio is able to consume any of these formats as long as they are bundled. STEMCstudio uses the `System` format for execution and so this is the most efficient way to consume your library in STEMCstudio. Our choice of bundler will be `rollup`. Another popular choice is `webpack`, but this tends to be used more for web applications, with rollup being used for libraries.

Bundling Types

STEMCstudio requires the type information to be bundled into one file. We will do this automatically using a rollup plugin.

Publishing and Consumption

We would like our library to be readily available for download from a Content Delivery Network. We will use `https://unpkg.com` for that purpose. Simply publishing our library to `npm` will make it available on this CDN. STEMCstudio can then pull in the implementation code and the type definitions by specifying their URLs in your STEMCstudio application. The consumer gets to choose the version of the library that is appropriate for their application.

Shared Dependencies

We would like to make sure that if our library depends on another library that we can choose whether to bundle that dependency or leave the choice of dependency version to the application developer. The latter is usually preferable for reducing code bloat. We will achieve this using the `peerDependencies` feature in our `package.json` file.

A.2. Step by Step Guide

I find it easier to create my GitHub repository first. Let's do that.

Sign into GitHub as the account that will own the library.

Select the `Repositories` tab.

Click the `New` button.

Give the repository a name. e.g. `my-lib`.

Add a `README.md` file by checking the check box.

Choose a license. `MIT` is usually a good choice.

Click the `Create repository` button. The repository will be created with a `LICENSE` file and `README.md` file.

Click the `Code` button and copy the SSH URL to the clipboard.

Open a new terminal. Navigate to the folder that you want to be the parent of your local repository folder. Clone the repository by executing a command similar to the following. Substitute your *owner* name for *geometryzen*, and your *repository* name for *my-lib*.

```
git clone git@github.com:geometryzen/my-lib.git
```

Change directory to enter the repository folder.

```
cd my-lib
```

Initialize npm to create the `package.json` file using the following command:

```
npm init
```

Accept the defaults and/or make any changes. You will be able to update the values later.

Now is a good time to fire up your favorite Integrated Development Environment (IDE). Mine is Visual Studio Code.

```
code .
```

Install TypeScript as a development dependency.

```
npm i -D typescript
```

Create the `tsconfig.json` file with the command:

```
npx tsc --init
```

Modify the contents of the `tsconfig.json` file to have the following property values.

```json
{
    "compilerOptions": {
        "target": "esnext",
        "module": "esnext",
        "moduleResolution": "node",
        "declaration": true,
        "declarationDir": "types",
        "emitDeclarationOnly": true,
        "sourceMap": true,
        "outDir": "build",
        "esModuleInterop": true,
        "forceConsistentCasingInFileNames": true,
        "strict": true,
        "noImplicitAny": true,
        "strictNullChecks": false,
        "noImplicitThis": true,
        "noUnusedLocals": true,
        "noUnusedParameters": false,
        "noImplicitReturns": true,
        "skipLibCheck": true,
        "removeComments": false,
        "resolveJsonModule": true
    },
    "include": ["./src/**/*.ts", "./rollup.config.mts", "./rollup-plugin-dts.d.ts"],
    "exclude": ["node_modules"]
}
```

It’s now time to start adding some implementation source files. Let’s create the source folder and add an empty TypeScript file.

```
mkdir src
cd src
touch index.ts
cd ..
```

Modify the `index.ts` file to have the following content. This will enable us to test a thin vertical slice from our library to STEMCstudio.

```
/**
 * Constructs a personalized string that can be used to greet a person.
 * @param name The name of the person receiving the greeting.
 * @returns a greeting string containing the name of the person receiving the greeting.
 */
export function greeting(name: string): string {
    return `Hello, ${name}!`;
}
```

We’ll now verify that it builds correctly by adding a script to `package.json`.

```
  "scripts": {
    "build": "tsc",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Now execute the following command.

```
npm run build
```

This should create a file `index.d.ts` in the `types` folder with the contents…

```
/**
 * Constructs a personalized string that can be used to greet a person.
 * @param name The name of the person receiving the greeting.
 * @returns a greeting string containing the name of the person receiving the greeting.
 */
declare function greeting(name: string): string;

export { greeting };
```

Module Formats and Bundling.

It's now time to install packages associated with `rollup`, define a configuration file `rollup.config.mts` and wire it all together. Let's start with creating the configuration file in the project root.

```
touch rollup.config.mts
```

Give this file the following contents

```
import commonjs from '@rollup/plugin-commonjs';
import resolve from '@rollup/plugin-node-resolve';
import terser from '@rollup/plugin-terser';
import typescript from '@rollup/plugin-typescript';
import { RollupOptions } from 'rollup';
import dts from 'rollup-plugin-dts';
import peer_deps_external from 'rollup-plugin-peer-deps-external';
import pkg from './package.json' assert { type: 'json' };

function non_minified_file(path: string): string {
    return path.replace(".min.js", ".js");
}

/**
* Comment with library information to be appended in the generated bundles.
*/
const banner = `/**
* ${pkg.name} ${pkg.version}
* (c) ${pkg.author.name} ${pkg.author.email}
* Released under the ${pkg.license} License.
*/
`.trim();

const options: RollupOptions[] = [
    {
        input: './src/index.ts',
        output: [
            {
                banner,
                file: non_minified_file(pkg.exports['.'].import),
                format: 'esm',
                sourcemap: true
            },
            {
                file: pkg.exports['.'].import,
                format: 'esm',
                sourcemap: true,
                plugins: [terser()]
            },
            {
                banner,
                file: non_minified_file(pkg.exports['.'].system),
                format: 'system',
                sourcemap: true
            },
            {
                file: pkg.exports['.'].system,
                format: 'system',
                sourcemap: true,
                plugins: [terser()]
            },
            {
                banner,
                file: pkg.exports['.'].require,
                format: 'commonjs',
                sourcemap: true
            },
            {
                file: pkg.browser,
                format: 'umd',
                name: 'MYLIB',
                sourcemap: true
            }
        ],
        plugins: [
            // Allows us to consume libraries that are CommonJS.
            commonjs(),
            peer_deps_external() as Plugin,
            resolve(),
            typescript({ tsconfig: './tsconfig.json', exclude: ['**/*.spec.ts'], noEmitOnError: true })
        ]
    },
    // Bundle the generated ESM type definitions.
    {
        input: './dist/esm/types/src/index.d.ts',
        output: [{ file: pkg.types, format: "esm" }],
        plugins: [dts()]
    }
];

```

```
export default options;
```

Install the rollup related packages

```
npm i -D @rollup/plugin-commonjs
npm i -D @rollup/plugin-node-resolve
npm i -D @rollup/plugin-terser
npm i -D @rollup/plugin-typescript
```

```
npm i -D rollup-plugin-dts
```

```
npm i -D rollup-plugin-peer-deps-external
```

I'm going to change the *name* property in *package.json* to show how the name of the package is distinct from the name of the repository.

> If you do this and create your package under an organization like `@geometryzen` then you will have to either pay for the privilege to be private or ensure that your package is publicly accessible.

Now lets fix up the properties in `package.json` that define the relative locations of the distribution resources. We do this using the "exports" property. The "module" property is the ESM entry point.

```
{
    "name": "@geometryzen/my-lib",
    "version": "0.9.26",
    "description": "My TypeScript Library Template",
    "exports": {
        ".": {
            "types": "./dist/index.d.ts",
            "import": "./dist/esm/index.min.js",
            "require": "./dist/commonjs/index.js",
            "system": "./dist/system/index.min.js",
            "default": "./dist/esm/index.min.js"
```

Finally, change the build script in `package.json` to read

```
"build": "rollup -c"
```

Let's give our bundled build a try...

```
npm run build
```

The `dist` folder should now contain subfolders for the formats `cjs`, `esm`, `system`, and `umd`. Each of those folders should contain at least an `index.js` file, which is the bundled implementation. The `dist` folder should also contain an `index.d.ts` file, which is our bundled type definitions.

Unit Testing

Testing your library is always a good idea. However, depending upon the nature of your library, the payback from automatic testing may vary. In general, JavaScript resources that don't use the Document Object Model (DOM) are easier to test. Code that does use the DOM can be tested using various test runners or DOM simulators, but may benefit from manual visual inspection of the output. In this example we will assume that the testing required does not involve the DOM.

Our testing approach will be to use *Jest* as our test runner and assertion library.

Install the following dependencies:

```
npm i -D jest
npm i -D @types/jest
npm i -D ts-jest
npm i -D ts-jest-resolver
```

Create the folder that will contain our tests.

```
mkdir tests
```

Now let's add the configuration file for *Jest* which is called *jest.config.ts*:

```
touch jest.config.ts
```

And here are the contents of *jest.config.ts*:

```
import type { Config } from "jest";

const config: Config = {
    preset: "ts-jest",
    resolver: "ts-jest-resolver"
};

export default config;
```

Change the *package.json* script for *test* to read:

```
"test": "jest"
```

And now you can run your unit tests with the command:

```
npm run test
```

You may find it useful to add the following script to *package.json*:

```
"pretest": "npm run build",
```

Coverage

A report of what you have tested and to what extent is called a coverage report. It is important because it will help you decide where you need to add testing before refactoring or simply to improve confidence in your code.

Add a script to the *package.json* to create the coverage report when the tests are run:

```
"coverage": "npm run test -- --coverage"
```

And now you can obtain a coverage report with the command:

```
npm run coverage
```

Documentation

```
npm i -D typedoc
npm i -D trash
npm i -D open-cli
```

```
touch typedoc.json
```

```json
{
    "entryPoints": ["src/index.ts"],
    "out": "docs",
    "exclude": "src/**/*.spec.ts",
    "excludePrivate": true,
    "theme": "default"
}
```

Add a script to the *package.json* to create the documentation:

```
"docs": "npm run docs:typedoc && open-cli docs/index.html",
"docs:typedoc": "trash docs && typedoc --options typedoc.json",
```

And now you can create documentation with the command:

```
npm run docs
```

The coverage report is an HTML page, *index.html*, that can be found in the *coverage* folder.

GitHub Pages

It would be nice to deploy our documenttaion to the web so that others can read it. For this we will use a technology called GitHub Pages.

Install the utilty that will transfer our documentation to GitHub.

```
npm i -D gh-pages
```

Add a script to *package.json*:

```
"pages": "npm run docs:typedoc && gh-pages -d docs"
```

Try it out:

```
npm run pages
```

The documentation should be visible at `https://owner.github.io/repo`.

Linting

```
npm i -D eslint
```

```
npm i -D @typescript-eslint/eslint-plugin
npm i -D @typescript-eslint/parser
```

eslint.config.js

```
touch eslint.config.js
```

```
import eslintJs from "@eslint/js";
import typescriptEslint from "@typescript-eslint/eslint-plugin";
import typescriptParser from "@typescript-eslint/parser";
import eslintConfigPrettier from "eslint-config-prettier";
import globals from "globals";
import eslintTs from "typescript-eslint";

export default [
    {
        ignores: ["build/**", "coverage/**", "dist/**", "docs/**", "node_modules/**", "types/**", "www/**"]
    },
    {
        files: ["**/*.ts"],
        languageOptions: {
            parser: typescriptParser,
            parserOptions: {},
            globals: globals.browser
        },
        plugins: {
            "@typescript-eslint": typescriptEslint
        },
        rules: {
            // Make this "error" when releasing.
            "brace-style": [2, "stroustrup"],
            "no-console": "warn",
            "no-param-reassign": "off",
            semi: [2, "always"]
        }
    },
    eslintJs.configs.recommended,
    ...eslintTs.configs.recommended,
    eslintConfigPrettier
];
```

Add a script to *package.json*:

```
"lint": "eslint . --ext .ts"
```

Try it out:

```
npm run lint
```

Test Driven Development

I’ll add the following script to *package.json*:

```
"dev": "rollup -c -w",
```

Here is the complete *package.json* file:

```json
{
    "name": "@geometryzen/my-lib",
    "version": "0.9.26",
    "description": "My TypeScript Library Template",
    "exports": {
        ".": {
            "types": "./dist/index.d.ts",
            "import": "./dist/esm/index.min.js",
            "require": "./dist/commonjs/index.js",
            "system": "./dist/system/index.min.js",
            "default": "./dist/esm/index.min.js"
        }
    },
    "browser": "./dist/umd/index.js",
    "main": "./dist/commonjs/index.js",
    "module": "./dist/esm/index.min.js",
    "type": "module",
    "types": "./dist/index.d.ts",
    "files": [
        "dist/commonjs/index.js",
        "dist/commonjs/index.js.map",
        "dist/esm/index.js",
        "dist/esm/index.js.map",
        "dist/esm/index.min.js",
        "dist/esm/index.min.js.map",
        "dist/index.d.ts",
        "dist/system/index.js",
        "dist/system/index.js.map",
        "dist/system/index.min.js",
        "dist/system/index.min.js.map",
        "dist/umd/index.js",
        "dist/umd/index.js.map"
    ],
    "keywords": [
        "geometryzen",
        "my",
        "lib"
    ],
    "publishConfig": {
        "access": "public"
    },
    "scripts": {
        "build": "npm run clean && rollup --config rollup.config.mts --configPlugin @rollup/plugin-typescript",
        "check": "npx package-check",
        "clean": "rm -rf coverage && rm -rf dist && rm -rf docs && rm -rf es2015 && rm -rf system && rm -rf types && rm -rf build",
        "coverage": "npm run test -- --coverage",
        "dev": "rollup --config rollup.config.mts --configPlugin @rollup/plugin-typescript -w",
        "docs": "npm run docs:typedoc && open-cli docs/index.html",
        "docs:typedoc": "rm -rf docs && typedoc --options typedoc.json",
        "format:check": "prettier --check '**/*.{js,ts,tsx,css,yml,json}'",
        "format:write": "prettier --write '**/*.{js,ts,tsx,css,yml,json}'",
        "lint": "eslint .",
        "pages": "npm run docs:typedoc && gh-pages -d docs",
        "release": "release-it",
        "test": "jest"
    },
    "repository": {
        "type": "git",
        "url": "git+https://github.com/geometryzen/my-lib.git"
    },
    "author": {
        "name": "David Geo Holmes",
        "email": "david.geo.holmes@gmail.com"
    },
    "license": "MIT",
    "bugs": {
        "url": "https://github.com/geometryzen/my-lib/issues"
    },
    "homepage": "https://github.com/geometryzen/my-lib#readme",
    "devDependencies": {
        "@rollup/plugin-commonjs": "^28.0.0",
        "@rollup/plugin-node-resolve": "^15.3.0",
        "@rollup/plugin-terser": "^0.4.4",
        "@rollup/plugin-typescript": "^11.1.6",
        "@skypack/package-check": "^0.2.2",
        "@types/jest": "^29.5.13",
        "@types/rollup-plugin-peer-deps-external": "^2.2.5",
        "@typescript-eslint/eslint-plugin": "^8.8.1",
        "@typescript-eslint/parser": "^8.8.1",
        "eslint": "^8.57.0",
        "eslint-config-prettier": "^9.1.0",
        "gh-pages": "^6.1.1",
        "jest": "^29.7.0",
        "open-cli": "^8.0.0",
        "prettier": "^3.3.3",
        "release-it": "^17.7.0",
        "rollup": "^4.24.0",
        "rollup-plugin-dts": "^6.1.1",
        "rollup-plugin-peer-deps-external": "^2.2.4",
        "ts-jest": "^29.2.5",
        "ts-jest-resolver": "^2.0.1",
        "ts-node": "^10.9.2",
        "tslib": "^2.7.0",
        "typedoc": "^0.26.8",
        "typescript": "^5.6.3",
        "typescript-eslint": "^8.8.1"
    }
```

```
}
```

Integration Testing

How can we test our library before actually publishing it to *npm*?

We will make use of the *npm link* command to create symbolic links.

See *https://github.com/geometryzen/my-app* for a test harness example.

Publishing to npm

We're now almost ready to publish.

The following script in *package.json* can be used to execute other scripts prior to publishing:

```
"prepublishOnly": "npm run build && npm run test && npm run lint && npm run pages",
```

We use the *.npmignore* file to control what artifacts are published to npm.

Create the *.npmignore* file in the root of your project.

```
touch .npmignore
```

The *npm pack* command is used to pack a *tarball* that contains the artifacts in your project. The following command can be used to inspect the artifacts that will be published to npm without actually creating a *tarball* file.

```
npm pack --dry-run
```

The contents of your *.npmignore* file should look something like:

```
CODE_OF_CONDUCT.md
CONTRIBUTING.md
coverage/
docs/
src/
tests/
types/
.eslintignore
.eslintrc
.mocharc.json
.nycrc.json
jest.config.js
register.js
rollup.config.js
rollup.config.mjs
rollup.config.mts
tsconfig.json
typedoc.json
dist/commonjs/index.d.ts
dist/commonjs/src/index.d.ts
dist/commonjs/tests/**/*.*
dist/esm/index.d.ts
dist/esm/src/index.d.ts
dist/esm/tests/**/*.*
dist/system/index.d.ts
dist/system/src/index.d.ts
dist/system/tests/**/*.*
dist/umd/index.d.ts
dist/umd/src/index.d.ts
dist/umd/tests/**/*.*
```

Publish your package to *npm* using the command:

```
npm publish
```

Consuming the Library

The published package contains bundled JavaScript files and one TypeScript type definition file.

Add a dependency to the `package.json` file that refers to the `@geometryzen/my-lib` package. You can either add the dependency manually by editing the `package.json` file or use the `Add Dependency` tool in the workspace explorer. If you add the dependency manually then you will need to obtain the latest version from the `https://npmjs.com` website. Your `package.json` file should look like:

```json
{
    "description": "STEMCbook my-lib Example",
    "dependencies": {
        "@geometryzen/my-lib": "0.9.26"
    },
    "name": "stemcbook-my-lib-example",
    "version": "1.0.0",
    "author": "David Geo Holmes",
    "private": true,
    "keywords": [
        "stemcbook",
        "my-lib"
    ]
}
```

Once the dependency has been added to the `package.json` file, the *system.config.json* file in your STEMCstudio project should contain the entry point for the JavaScript files:

```
{
    "map": {
        "@geometryzen/my-lib": "https://cdn.jsdelivr.net/npm/@geometryzen/my-lib@0.9.26/package.json"
    }
}
```

> The *system.config.json* file is generated and will only be visible if the `Show Generated Files` option is checked in the Project Settings. This file is readonly because it is generated by STEMCstudio.

Likewise, the *types.config.json* file in your STEMCstudio project should contain an entry:

```
{
    "map": {
        "@geometryzen/my-lib": "https://cdn.jsdelivr.net/npm/@geometryzen/my-lib@0.9.26/package.json"
    }
}
```

> The *types.config.json* file is generated and will only be visible if the `Show Generated Files` option is checked in the Project Settings. This file is readonly because it is generated by STEMCstudio.

Finally, importing and using a JavaScript resource in your STEMCstudio project will look like:

```
import { greeting } from "@geometryzen/my-lib"

const titleElement = document.getElementById('title')
if (titleElement) {
    titleElement.textContent = greeting("World")
}

window.onunload = function() {
    // Write your application cleanup code here.
}
```

The `greeting` function has been imported and is called with an appropriate argument. The result is used to set an element in the HTML DOM.

A.3. Summary

We've looked at the desirable features of a JavaScript library and have followed a step-by-step process to produce an example library. External libraries will be extremely valuable for creating more sophisticated applications and will make an application consuming the library more manageable.

Appendix B: Consuming ES6 Module format Libraries

B.1. Converting ES6 module format to System

STEMCstudio simulates ES6 modules by transpiling code to the `System` format. The `System` format is functionally equivalent to the ES6 module format and so there is no loss in capability.

> STEMCstudio does this so that it can execute code entirely in the browser without requiring a server-side capability to bundle code and serve it. Requiring a server-side capability per user can amount to expensive cloud computing resources.

STEMCstudio transpiles TypeScript files in your project on-the-fly into `System` format and can also do the same for external JavaScript libraries in ES6 module format. However, because external libraries are only loaded at execution time, the transpilation can impose a performance penalty. Moreover, the code that performs the transpilation, `typescript.js`, has to be available to the `System` loader, which would be done by adding a `script` tag to the HTML file:

```
<script src="https://path/to/typescript.js">
```

But `typescript.js` is a large (approx 10MB) file and so the loading of this file can be slow and can seriously increase the load time of your application. Using the `async` or `defer` attributes to control the load either does not work or does not affect the performance.

An additional problem that we may have with packages in ES6 module format is that the code is un-bundled. STEMCstudio is currently unable to consume these un-bundled packages.

B.2. Solution is System module format

Don’t do at runtime what you can do at design time! More concretely, convert your ESM packages to `System` format up-front by creating a package that re-exports the artifacts of the original library in `System` format. Another benefit of this approach is that you can bundle the TypeScript type definitions into a file that can be readily consumed by STEMCstudio.

B.3. Wrapping their modules

In this section we take a look at a concrete example of wrapping an ES6 module. In the next section we’ll look at how to consume it in STEMCstudio.

The `@microsoft/fast-components` library is in ES6 module format. It has been re-packaged and bundled into a package `@geometryzen/fast-components` in `System` format. You can explore this library at https://github.com/geometryzen/fast-components. Let’s take a look at some of the more important aspects of this repository.

The *index.ts* file simply re-exports everything in the original package. Nothing is subtracted or added:

```
export * from '@microsoft/fast-components';
```

The repository itself is similar in construction to those described in Appendix A. The *rollup.config.mjs* file controls the artifacts that are generated. Notice that we only build the `System` module format.

```
/* eslint-disable no-undef */
/* eslint-disable @typescript-eslint/no-var-requires */
import resolve from '@rollup/plugin-node-resolve';
import terser from '@rollup/plugin-terser';
import typescript from '@rollup/plugin-typescript';
import dts from 'rollup-plugin-dts';
import external from 'rollup-plugin-peer-deps-external';
import pkg from './package.json' assert { type: 'json' };

/**
 * Comment with library information to be appended in the generated bundles.
 */
const banner = `/**
 * ${pkg.name} ${pkg.version}
 * (c) ${pkg.author}
 * Released under the ${pkg.license} License.
 */
`.trim();

/**
 * @type {import('rollup').RollupOptions}
 */
const options =
{
    input: 'src/index.ts',
    output: [
        {
            banner,
            file: './dist/system/index.js',
            format: 'system',
            sourcemap: true
        },
        {
            banner,
            file: './dist/system/index.min.js',
            format: 'system',
            sourcemap: true,
            plugins: [terser()]
        }
    ],
    plugins: [
        external(),
        resolve(),
        typescript({ tsconfig: './tsconfig.json' })
    ]
};

export default [
    options,
    {
        input: 'node_modules/@microsoft/fast-components/dist/fast-components.d.ts',
        output: [{ file: pkg.types, format: "esm" }],
        plugins: [dts()],
    }
];
```

In the *package.json* file, the `browser` property replaces the `main` property to show that the module is meant to be used client-side.

> ⚠ By pointing the `browser` property to the minified `System` JavaScript file, the *unpkg.com* CDN will return this file by default. This is undocumented so we should be careful not to rely on it.

```
{
    "name": "@geometryzen/fast-components",
    "version": "2.30.6",
    "description": "@microsoft/fast-components as a system module",
    "browser": "./dist/system/index.min.js",
    "types": "./dist/index.d.ts",
    "publishConfig": {
        "access": "public"
    },
```

B.4. Consuming your package

If you were to look at the *package.json* file for `@microsoft/fast-components` then you would see that the package has regular *dependencies* on various other `@microsoft/fast-*` packages. Because these are not *peerDependencies*, we can expect that the package embeds the code from other packages. Thus, if we consume the module `@geometryzen/fast-components` then we don't need to also import the other modules in order to ensure that our application runs correctly.

However, if you were to look at https://unpkg.com/@geometryzen/fast-components@0.9.6/dist/index.d.ts then you would see that the *index.d.ts* file that is shipped with `@geometryzen/fast-components` imports from `@microsoft/fast-elements`, `@microsoft/fast-foundation`, and `@microsoft/fast-web-utilities`. So if we want a complete developer experience in the STEMCstudio IDE then we need to ensure that the typings files for these modules are available.

But how can we consume the `@geometryzen/fast-components` module to get the correct runtime behavior and at the same time get the corect developer experience in the STEMCstudio IDE?

The trick is to code our application *as if* it is consuming the original `@microsoft/fast-*` modules, but map these modules onto the wrapped implementations. Let's look at an example that uses both `fast-components` and `fast-elements`.

You can study this project directly at https://www.stemcstudio.com/gists/ea1b221a5193f1731e5c7b2737999b24.

The main application file, *index.ts*, uses the `@microsoft/fast-components` module directly to register FAST custom components as well as a custom component that is defined in the *NameTag.js* file.

```
import {
    allComponents,
    baseLayerLuminance,
    provideFASTDesignSystem,
    StandardLuminance,
    Switch
} from "@microsoft/fast-components"

import { NameTag } from "./NameTag.js"

const designSystem = provideFASTDesignSystem()
designSystem.register(allComponents)
designSystem.register(NameTag)

const toggle = document.getElementById('toggle') as Switch

const updateLuminance = function() {
    baseLayerLuminance.setValueFor(
        document.body,
        toggle.checked ? StandardLuminance.LightMode : StandardLuminance.DarkMode
    )
}

updateLuminance()

toggle.addEventListener('click', updateLuminance)

window.onunload = function() {
    toggle.removeEventListener('click', updateLuminance)
}
```

The *NameTag.ts* file is using the `@microsoft/fast-element` module.

```
import { attr, customElement, FASTElement, html } from "@microsoft/fast-element"
import type { ViewTemplate } from "@microsoft/fast-element"

const template: ViewTemplate<NameTag> = html<NameTag>`
```

```
  <div class="header">
    <h3>Hello, ${(x: NameTag) => x.name}!</h3>
  </div>
  <div class="body"></div>
  <div class="footer"></div>
`

@customElement({
    name: 'name-tag',
    template
})
export class NameTag extends FASTElement {
    /**
     * The name of the thing
     */
    @attr name = ''

    nameChanged() {
    }

    connectedCallback() {
        super.connectedCallback()
    }
}
```

So we need both the @microsoft/fast-components and @microsoft/fast-element modules, and we need to map them to our System module implementations.

We also need the @microsoft/fast- type definitions to be mapped to our @geometryzen/fast- implementations. We map as many modules as we need to allow the STEMCstudio IDE to infer the types correctly.

We do this in *studio.config.json*.

```
{
    "hideConfigFiles": false,
    "hideReferenceFiles": true,
    "linting": true,
    "noLoopCheck": true,
    "operatorOverloading": false,
    "overrides": [
        {
            "name": "@microsoft/fast-components",
            "version": "2.30.6",
            "system": "https://cdn.jsdelivr.net/npm/@geometryzen/fast-components@2.30.6/dist/system/index.min.js",
            "types": "https://cdn.jsdelivr.net/npm/@microsoft/fast-components@2.30.6/package.json"
        },
        {
            "name": "@microsoft/fast-element",
            "version": "1.12.0",
            "system": "https://cdn.jsdelivr.net/npm/@geometryzen/fast-element@0.9.9/dist/system/index.min.js",
            "types": "https://cdn.jsdelivr.net/npm/@microsoft/fast-element@1.12.0/package.json"
        },
        {
            "name": "@microsoft/fast-foundation",
            "version": "2.49.5",
            "system": "https://cdn.jsdelivr.net/npm/@geometryzen/fast-foundation@2.49.5/dist/system/index.min.js",
            "types": "https://cdn.jsdelivr.net/npm/@microsoft/fast-foundation@2.49.5/package.json"
        }
    ],
    "references": {},
    "showGeneratedFiles": false
}
```

B.5. Summary

We now understood the performance problem associated with loading external ES6 module packages and how to solve them by re-bundling in `System` format.

Appendix C: Operator Overloading

C.1. What is Operator Overloading?

When you perform an arithmetic operation such as `2 + 2`, or `x + 1` where `x` is a `number` type, the JavaScript interpreter is able to quite happily perform the computation. But what happens if you want to define a mathematical object (such as a vector which is therefore not a primitive `number`), and you want that object to have sensible behavior when it interacts with mathematical operators? In other words, what we are trying to do is to give the operator additional behavior when it is encountered with certain operands, which is why it is called **operator overloading**.

C.2. How it Works

Operator Overloading is typically performed by running JavaScript code through a transformation step that injects special code in place of normal operators.

Operator Overloading is not implemented in standard JavaScript. The reason for this is that JavaScript performance would suffer because JavaScript does not provide type information for the runtime system to optimize code.

Experiments by a number of implementers show the surprising result that such a bolt-on Operator Overloading implementation only affects overall performance by around 5%. This makes it very acceptable in an education or research environment.

STEMCstudio implements operator overloading in this way, but also makes it possible to turn Operator Overloading on or off using a project-level switch.

STEMCstudio recognizes a wide range of operators for overloading, and by using a standard approach makes it possible for users to implement their own mathematical objects with operator overloading. In addition, STEMCstudio adjusts the precedence of operators to conform with the notation standards of Geometric Algebra so that parenthesis may be dropped for clarity.

C.3. Code transformation for Operator Overloading

In STEMCstudio, a special transformation step takes place after the TypeScript code has been transpiled to JavaScript. This step replaces operators by function calls that inspect their arguments for special dunder (double underscore) methods on objects. For example, suppose that we have the following TypeScript code:

```
a + b
```

This code would be transformed to:

```
add(a, b)
```

The `add` function takes two arguments `a`, and `b`. `add` is a standard function built into a library that is included by STEMCstudio if Operator Overloading is requested. The `add` function looks to see if `a` is an object and whether it has the special dunder method `add`. If so, the `add` function executes the code:

```
a.__add__(b)
```

The `add` method may veto the call (effectively saying that it cannot perform the action) by returning `undefined` (or `void 0`). If the method invocation is not vetoed and a result is returned then this is the result of `a + b`. If the method invocation is vetoed then the `add` function will try:

```
b.__radd__(a)
```

This is the right-addition version of the same `a + b` expression, except that `b` gets a chance to handle the execution. This left or right handed invocation of addition makes sense when you consider that one type may know about another but not the other way around.

Finally, if these special methods don't exist or the invocations are vetoed, the `add` function falls back to simply calling `a + b`. At this point, the JavaScript runtime takes over, possibly coercing the arguments of the `+` operator to `number` before performing the addition.

Operator Overloading exists for both binary operators and unary operators.

C.4. Binary Operators

The following table summarizes the binary operators, their meaning, and the dunder methods.

`+`	addition	`__add__`	`__radd__`
`-`	subtraction	`__sub__`	`__rsub__`
`*`	multiplication	`__mul__`	`__rmul__`
`/`	division	`__div__`	`__rdiv__`
`<<`	left contraction	`__lshift__`	`__rshift__`
`>>`	right contraction	`__rshift__`	`__rrshift__`
`|`	scalar product	`__vbar__`	`__rvbar__`
`^`	exterior product	`__wedge__`	`__rwedge__`
`===`	equality	`__eq__`	`__req__`
`!==`	inequality	`__ne__`	`__rne__`
`>=`	greater than or equal	`__ge__`	`__rge__`
`>`	greater than	`__gt__`	`__rgt__`
`<=`	less than or equal	`__le__`	`__rle__`
`<`	less than	`__lt__`	`__rlt__`

C.5. Unary Operators

The following table summarizes the unary operators, their meaning, and the dunder method.

`~`	reversion	`__tilde__`
`!`	inverse	`__bang__`
`-`	unary minus	`__neg__`
`+`	unary plus	`__pos__`

C.6. Operator Precedence

When Operator Overloading is enabled in STEMCstudio, the precedence of operators is adjusted to conform to the norms for geometric algebra. In particular, the contraction operators and scalar product bind to their arguments the most tightly. The exterior or wedge product binds less tightly but more tightly than multiplication. As always, multiplication binds more tightly than addition. If in doubt, or the expression is esoteric, use parenthesis. However, as in mathematics notation, precedence rules exist to make mathematical expressions more readable.

C.7. Example Complex number class

The following complex number class implements binary addition, binary multiplication, and unary minus.

```
export class Complex {
    constructor(public readonly real: number, public readonly imag: number) {
    }
    __add__(rhs: Complex | number): Complex | undefined {
        if (typeof rhs === 'number') {
            return new Complex(this.real + rhs, this.imag)
        }
        else if (rhs instanceof Complex) {
            return new Complex(this.real + rhs.real, this.imag + rhs.imag)
        }
        else {
            return void 0
        }
    }
    __radd__(lhs: number): Complex | undefined {
        if (typeof lhs === 'number') {
            return new Complex(lhs + this.real, this.imag)
        }
        else {
            return void 0
        }
    }
    __mul__(rhs: Complex | number): Complex | undefined {
        if (typeof rhs === 'number') {
            return new Complex(this.real * rhs, this.imag * rhs)
        }
        else if (rhs instanceof Complex) {
            const x = this.real * rhs.real - this.imag * rhs.imag
            const y = this.real * rhs.imag + this.imag * rhs.real
            return new Complex(x, y)
        }
        else {
            return void 0
        }
    }
    __rmul__(lhs: number): Complex | undefined {
        if (typeof lhs === 'number') {
            return new Complex(lhs * this.real, lhs * this.imag)
        }
        else {
            return void 0
        }
    }
    __neg__(): Complex {
        return new Complex(-this.real, -this.imag)
    }
    toString(): string {
        return `${this.real} + ${this.imag} * i`
    }
}

export function complex(x: number, y: number): Complex {
    return new Complex(x, y)
}
```

Notice that the dunder methods do not mutate their object instance (`this`). This results in expected behavior and is strongly recommended. In contrast, you may sometimes wish to make mathematical objects mutable. This should only be done in performance critical applications such as graphics where the creation of temporary objects would result in extra work for the JavaScript garbage collector. Even in such cases, the dunder methods should not mutate, but you may define non-dunder custom methods that do.

In general you should make your mathematical objects immutable so that operations do not have side-effects and make it easy to reason about values.

C.8. Summary

We have seen what Operator Overloading is, how it is implemented in STEMCstudio, and a detailed example. You can now experiment with your own implementatins in STEMCstudio.