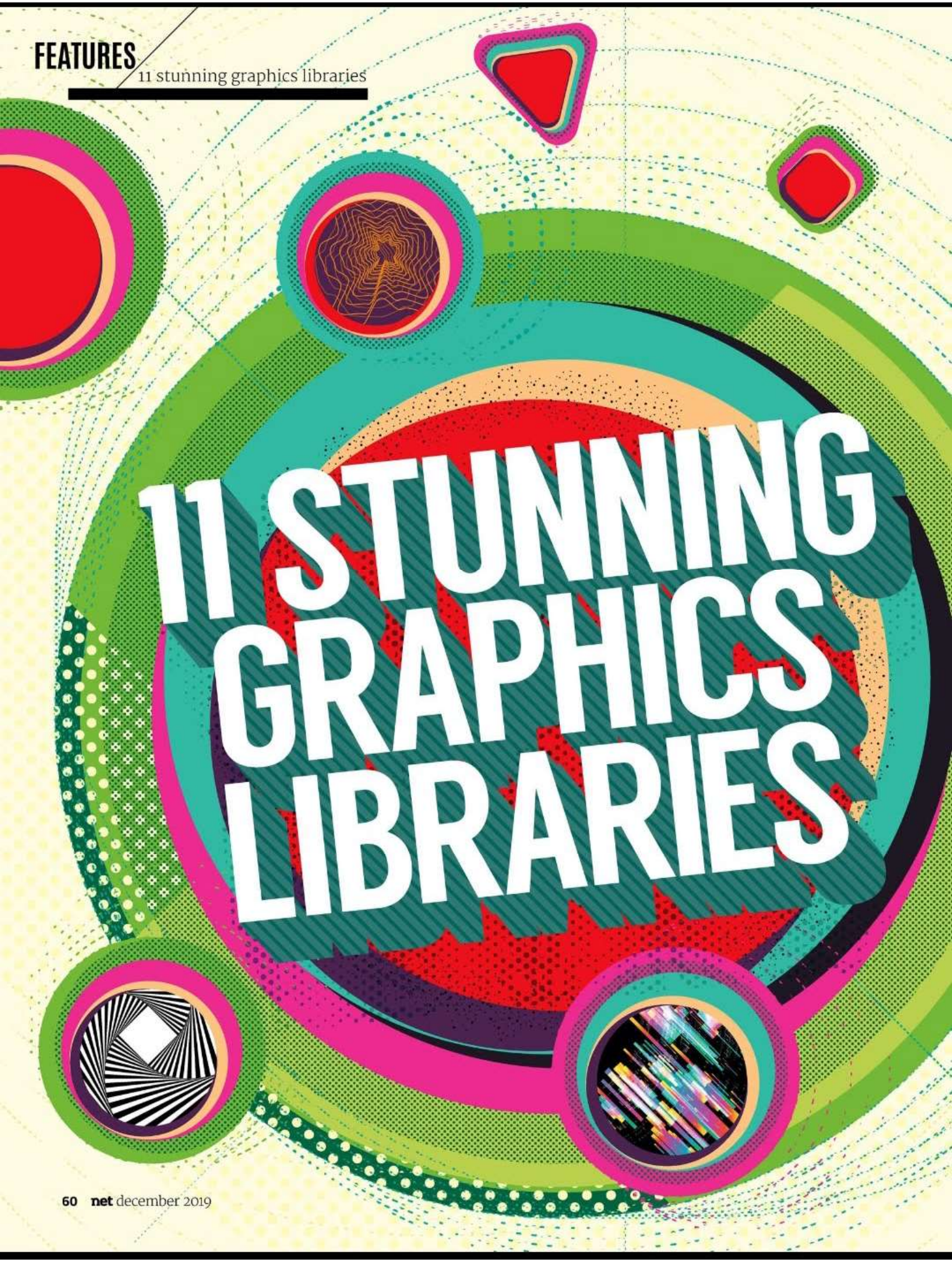
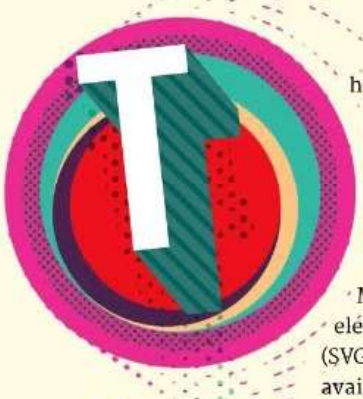


11 STUNNING GRAPHICS LIBRARIES

BURKE HOLLAND REVEALS GREAT GRAPHICAL JAVASCRIPT FRAMEWORKS AND SHOWS HOW YOU CAN USE THEM TO PRODUCE PERFECT ANIMATIONS, ILLUSTRATIONS AND DATA VIZ



AUTHOR

BURKE HOLLAND

Senior developer advocate for Microsoft, Holland is a big fan of sarcasm and JavaScript, the latter of which makes him the perfect person to assess the various frameworks available.

t: @burkeholland

They say that a picture is worth a thousand words. Coincidentally, that's about how many lines of code it takes to create that picture in a web browser. Fortunately, the browser offers several high-powered drawing APIs and surfaces. Most notably, these are the canvas element and scalable vector graphics (SVG). Both of these features are now available in almost all desktop and mobile browsers but the APIs required to use them are rather low level and 'low level' typically translates into a lot of tedious and redundant code just to do simple things. Since writing tedious and redundant code is not high on the list of things that developers typically enjoy doing, there are thankfully several libraries available to help you with all of your browser drawing requirements.

In this article, I'll take a look at some of the options available. We'll explore the most popular libraries and talk about what their strengths and weaknesses are. We're also going to do a little spelunking into some of the lesser known projects that fill some very interesting niche needs – the kind that you don't know you need until you need them. ▶

D3.JS

https://d3js.org/

D3.js is the big boss of JavaScript graphics libraries. With over 875K weekly downloads on npm (*https://www.npmjs.com/package/d3*), it is quite possibly the most well known and widely used of all the graphics libraries. It even has its own Wikipedia page (*https://en.wikipedia.org/wiki/D3.js*). And, let's face it, isn't that when you know you've made it to the big time?

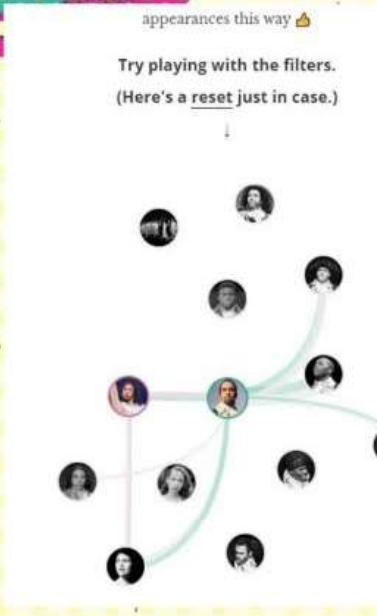
D3 enables you to build data visualisations of any kind. You only need to glance through its examples page (*https://github.com/d3/d3/wiki/Gallery*) to see the world of possibilities. Better yet, visit Shirley Wu's interactive visualisation of every line in the musical Hamilton (*https://pudding.cool/2017/03/hamilton/*) if you really want to have your mind blown.

D3 is an all-encompassing tool. It has its own DOM selection, AJAX capabilities and even a proprietary random number generator. Each component of D3 is its own Node module that must be imported. For instance, the selection module is called d3-selection. There are also modules for arrays, shapes, colours, drag-and-drop, time and much more (*https://www.npmjs.com/search?q=d3*).

The power of D3 comes with the trade-off of complexity. The learning curve can be steep and the code can still feel verbose. Building something as simple as a bar chart requires you to manually assemble the axis, scales, ticks and even draw the rectangles that will represent the bars. Developers often complain about the low-level understanding required to be effective with D3.

This is largely because creating complex data visualisations requires you to have a low-level understanding of the visualisation you want to create. D3 is not the best option for pre-baked charts. For that, there are several other choices that will find you in the 'pit of success' much faster (*https://netm.ag/2mFisDC*).

D3 is capable of rendering to canvas and SVG. However, the real magic of D3 is in its ability to 'data bind' to the graphics it generates. Think of a chart that changes as the incoming data changes. With SVG, each graphical item is an individual element that can be selected and updated. This is not possible with canvas and, since D3 is fundamentally about powering data visualisations, SVG is usually the preferred output format.



Right D3.js provides boundless possibilities, as Shirley Wu's interactive Hamilton visualisation demonstrates

CHART.JS

https://www.chartis.org/

Chart.js is an open-source project for building robust charts with JavaScript. The big difference between Chart.js and D3 is that while you can build just about anything with D3, Chart.js limits you to eight pre-built chart types: line, bar, pie, polar, bubble, scatter, area and mixed. While this seems limiting, it's what makes Chart.js simpler to get started with. This is especially true for those who aren't experts in data vis but know their way around a basic chart.

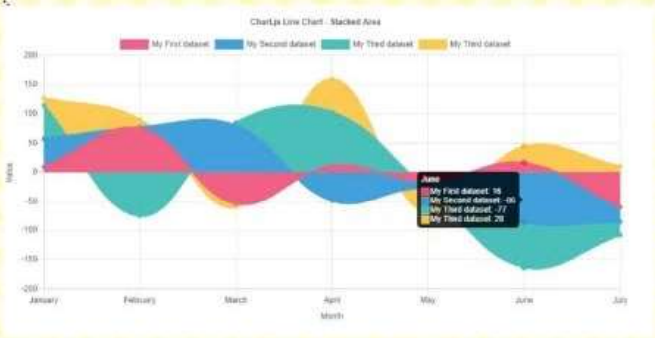
The syntax is all built around a `chart` type. You initialise a new chart on an existing canvas element, set the chart type and then set the chart options. Chart.js only renders to canvas. This is not a problem since all modern browsers support the HTML `canvas` element but it might be a hangup for developers who have requirements for SVG support.

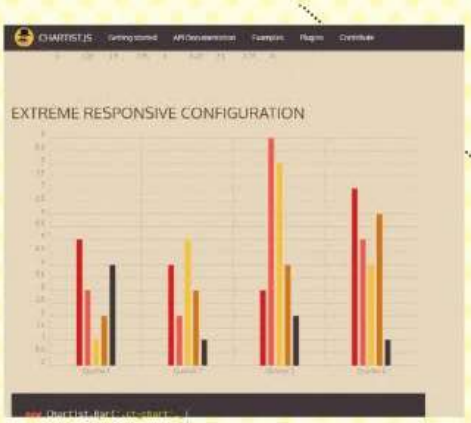
It also means you are limited in the animations that are possible. Out of the box, Chart.js has support for all easing equations and animations are specified with one property setting. While that makes it quick and easy to get an animated chart, not having individual SVG elements prevents you from being able to do complex animations using CSS3 transitions and animations.

Unlike D3, Chart.js is not modular, so it only takes one JavaScript include to get support for all functions and chart types. This makes it easier to get started but means your assets can be much larger. This is especially true if you require time axes – Chart.js then requires Moment.js, which is ~51kb minified and zipped.

It's far easier to create a bar chart with Chart.js than with D3. However, there is a ceiling that comes with the simplicity.

Right Chart.js has eight prebuilt chart types, which means users can get up and running quickly





Left One big bonus of using Chartist is that it is responsive by default

© Shirley Wu / The Pudding: https://pudding.cool/2017/03/hamilton

CHARTIST RENDERS TO SVG INSTEAD OF CANVAS, MAKING IT MUCH MORE CUSTOMISABLE

You may find you hit the limit of what it will do out of the box. Often, developers start with a solution like Chart.js and then graduate to D3.

If the simplicity of Chart.js appeals to you, you might really like the next option: Chartist.

CHARTIST

https://gionkunz.github.io/chartist-js/

Chartist aims to be a simple, streamlined charting library that is small in size and easy to get started with. It is also designed to be responsive by default. This is a bigger deal than it sounds, as frameworks like D3 do not resize charts automatically but require the developer to tie into events and redraw graphics.

Chartist is also tiny in comparison to Chart.js. It weighs in at a mere 10KB with zero dependencies. That might be because it only offers three chart types: line, bar and pie. There are variations within these types (ie scatter plot is a line type in Chartist) but the tiny size and ease of configuration is countered by the lack of out-of-the-box chart types.

Chartist renders to SVG instead of canvas, making it much more customisable in terms of look and feel, as well as providing far more control over interactivity and animations. However, not having rendering access to a canvas means that you might have a harder time doing certain actions. For instance, there is an API for rendering a canvas to an image (toDataURI). That option does not exist for SVG, so exporting a chart as an image will prove to be much trickier. In an ideal world, you would have the option to render to both modes.

Chartist charts are easier to configure than Chart.js, as there are fewer options available. While it's possible to extend these charts with quite a bit of functionality, their focus on simplicity means that they are, by definition, simple. Chartist is a great solution for those who need a basic charting solution. Charts are inherently hard to configure as they require some sort of knowledge about how to set up the data along certain axis and grouped in certain ways. Chartist makes the charting portion as simple as possible but you may find yourself in need of a more powerful solution as you become more comfortable with generating your charts.

Chartist also lists open-source framework support, including React and Angular. There is no mention of a Vue package on its site. ▶

BRITECHARTS

http://eventbrite.github.io/britecharts/

Britecharts is a charting library that wraps D3. It was made by Eventbrite, who then open-sourced the project under the permissive Apache V2 license.

It offers a very minimal, yet aesthetically pleasing set of charts. While it can be quite a task to create a bar chart with vanilla D3, Britecharts' wrapping makes it as simple as creating a new `barChart` object and then setting its width and height.

```
barChart.width(500).height(500);
```

Britecharts has support for all of the basic chart types: line, bar, donut, bullet, scatter plot, sparkline and step, which is more than those offered by libraries like Chartist. It also provides basic tooltip and legend functionality. The animations for the charts are built-in and Eventbrite has provided some beautiful colour schemes.

Ultimately, Britecharts is a great option for basic charting functionality. The configuration objects are quite simple and you still get the power of D3 under the covers without having to know anything about D3 itself. Many developers will find this a more compelling option than simply building a complete chart from the ground up with D3.

It also has a focus on the data-binding aspects of D3, making it quite good for charts that need to change as underlying data changes. While somewhat restrictive in the available types, it also has a base chart type that you can extend to create new chart types of your own.

TAUCHARTS

https://taucharts.com

Taucharts is another charting solution that wraps the complexity of D3 in an easy to implement API. It is built on the concepts from *The Grammar of Graphics*, a book by author Leland Wilkinson (https://netm.ag/2mAFUBK). It provides understanding of when and how to use which data visualisations to display different kinds of data.

Right Taucharts teaches users about the fundamentals of data visualisation

Out of the box, it features line, bar, scatterplot, area and facet charts. However, it implements the concepts from *The Grammar of Graphics* into 'Taucharts Language' which provides a framework in which to implement your own data visualisations.

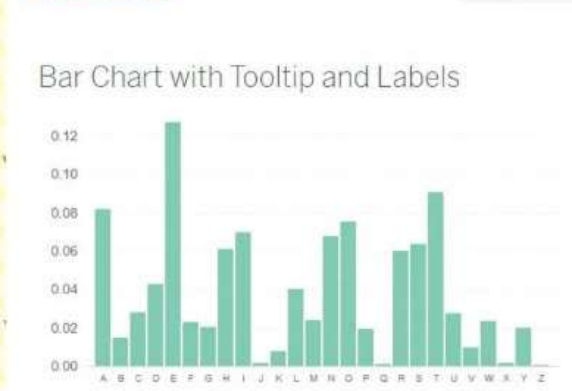
Taucharts looks quite compelling and the fact that it's built on D3 makes it an attractive and powerful option. There is the feeling, though, that the developer also needs to read *The Grammar of Graphics* in order to fully leverage its power.

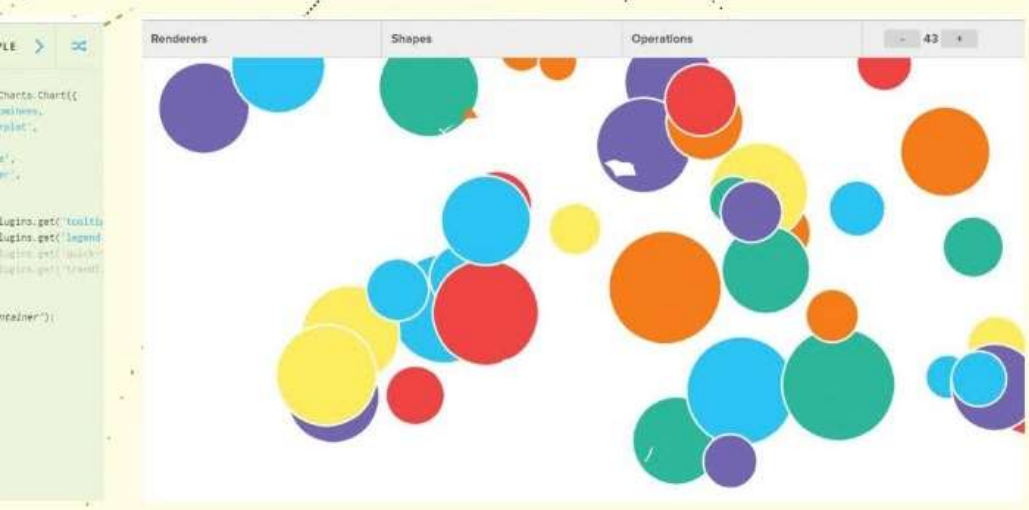
So far we've covered only charting and data visualisation. This is because drawing charts is the most common use-case for a graphics library in a browser. But it is by no means the only one. Another quite common scenario for graphics is, of course, animation.

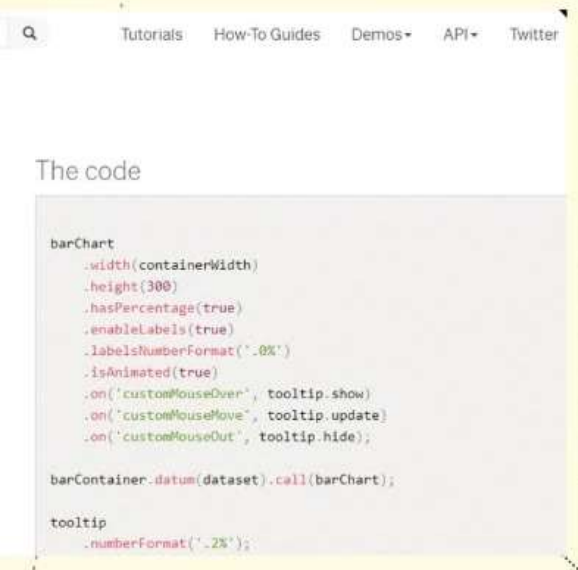
TWO.JS

https://two.js.org

Not to be confused with D3.js, two.js is an open-source JavaScript library for two-dimensional drawing on the web. It is also able to target all three graphics options in modern browsers: SVG, Canvas and WebGL.







Above Twos.js comes into its own when handling 2D animation

Left The predetermined methodology of Pts is good for interactive visualisations, as well as animations

Below Britecharts wraps D3, so you can enjoy D3 functionality without having to learn how it works

Two.js is somewhat similar to D3 in that it is strictly focused on drawing and does not have pre-baked charts or interactive structures to choose from. This means that, just like D3, you need an underlying understanding of the type of drawing you are trying to do and how to achieve that with the constructs two.js provides. Drawing a circle is rather straightforward. Building out a detailed animation, on the other hand, is a much more complicated endeavour. Two.js only abstracts the tedium of drawing shapes, not the tedium of the overall drawing.

Two.js also keeps track of all of the objects that you create, so you can reference and animate them at any time. This is particularly important if you are doing game development and you have assets that need to be tracked for things like collision detection. It has a built-in animation loop, which relieves you from having to worry about animation frames, and makes it easier to tie in an animation library such as GreenSock (*https://greensock.com/*).

While two.js is powerful, its free-form nature might leave some developers unsure of how to begin and it's more of a niche tool for 2D drawing and animation. Another excellent option is pts.js.

PTS.JS

https://ptsis.org

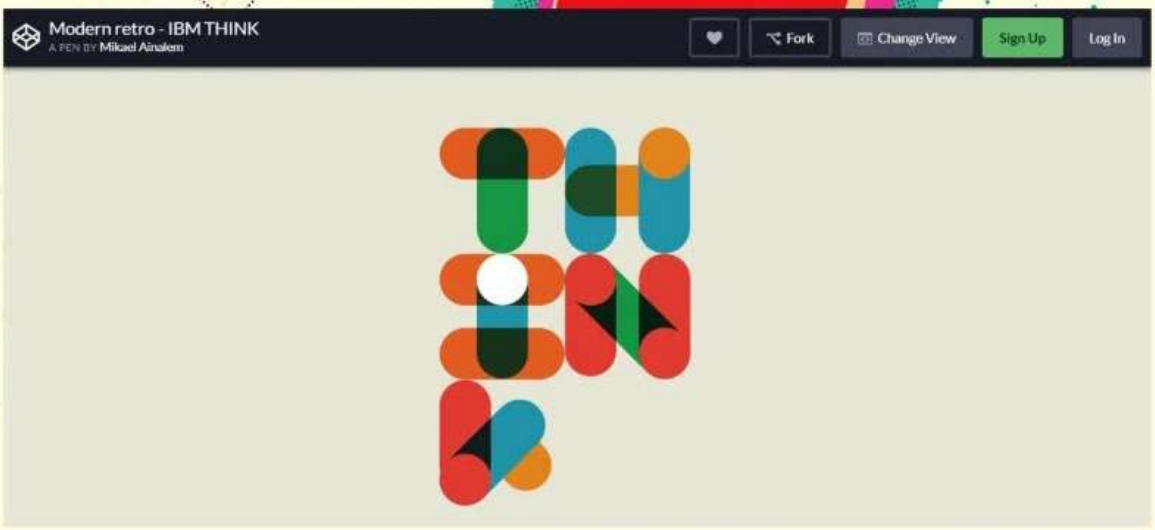
Pts is also a two-dimensional drawing library. It is fundamentally different to two.js however, as it uses a predetermined methodology for how drawings and animations should be assembled: space, form and point. The analogy its developers use to explain this is one from the physical world. Space is paper. Form is the pencil. And point is your idea.

In terms of its implementation, `space` is a canvas element. Once the canvas element is created, you can add `players` to it. These can be either functions or objects. These functions and objects must conform to the predetermined interface that a `space` has. Pts is built on TypeScript, so there's no need to guess at what those are as the tooling you use will likely suggest those with autocomplete.

TWO.JS TRACKS THE OBJECTS YOU CREATE SO YOU CAN REFERENCE THEM AT ANY TIME

For instance, a space has a `start` function that you can specify. This is code that is run when the space is ready. Within these functions, the drawing to the space occurs using the `form` object. Form objects can draw any sort of shape and the `point` is where these items are located in the space.

Pts seems to be primarily designed for creating interactive visualisations and animations. Its implementation is interesting, albeit quite abstract. Developers may have a hard time understanding the 'space, form, point' model that Pts requires. This is another mental hurdle that will have to be cleared in addition to that of simply drawing and animating shapes. ▶



Left Anime.js works on a staggered system, which helps with timing your animations

Below PixiJS is a 2D drawing library with an API that was built and used on Adobe Flash, so will be familiar to some

ANIME.JS

https://animejs.com

Anime.js is primarily an animation library. It has a built-in stagger system to make it more simple to have complex animations that overlap or are dependent upon the occurrence of another execution. It's common for animations to be timed together or to be triggered by one another. The staggering system makes this easier to implement, as it helps relieve some of the overhead of tracking everything happening on the page and manually configuring the animation timings.

Unlike the drawing libraries covered so far, anime.js doesn't have APIs for drawing shapes. Instead, it assumes your shapes already exist and that you want to animate them. This makes it great for use with libraries such as two.js. Anime.js has support for animating CSS properties, SVG, DOM and even JavaScript objects.

Anime.js is a good option for animating existing drawings and will likely be combined with another library. It should be considered an alternative to something like GreenSock and not a replacement for other drawing libraries. Anime.js would likely be used for more complex animations that need to happen as part of an interactive web experience.

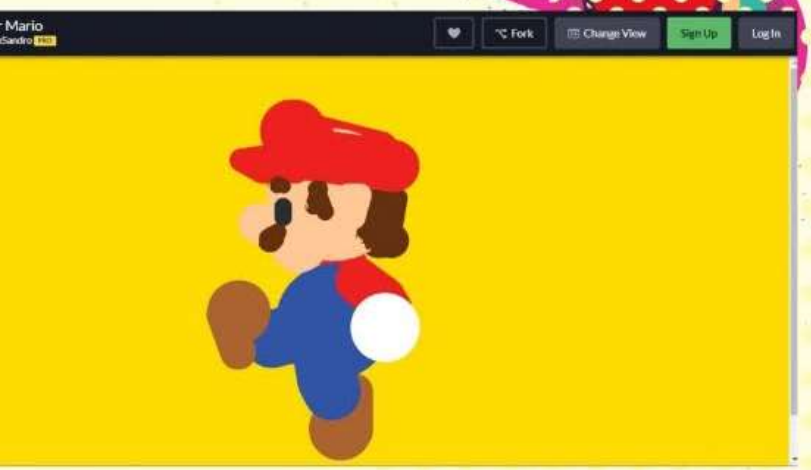
PIXIJS

https://www.pixiis.com/

PixiJS is another 2D drawing library. Its main purpose is to make it easier to display, animate and manage 2D graphics, so you can focus on building your experience or game without worrying about keeping up with all of the shapes and images you have to draw and animate. If you're building a game, assets (or sprites) can quickly balloon to a number that's hard to manage.

A compelling aspect of PixiJS is that it comes from an API that was built and used extensively in Adobe Flash. This is a huge benefit for developers coming from a Flash background, as the experience will feel familiar. It is also similar to Apple's SpriteKit.

PixiJS is not a game engine, so if you're using it for games, you won't find any tools or physics to handle things like collision detection. You'll need to wrap it in an actual game engine or one you build yourself, if you're feeling intrepid.



Left Zdog offers the look of 3D using 2D drawings

Below Snap.svg has an API that will select an SVG element for you to then draw elements to it

have the higher-end capabilities required for complex graphics and animations.

ZDOG

https://zzz.dog/

Most of the drawing engines that we have discussed so far are two-dimensional. This is because most of the interacts we have with our screen occur in two dimensions – along the X and Y axis. Three dimensional drawings and animations are usually much more complex.

Zdog is a library for building pseudo-3D experiences that are mostly flat in nature. It's called pseudo-3D because while it conceptualises its drawings in 3D space, it renders them as flat shapes. It uses visual tricks to make 2D objects appear 3D. The effect is really interesting. It looks completely three dimensional when the animation is viewed but when a screenshot is taken, it is clearly a flat image. Here is one example: the rotating Mario demo (*https://codepen.io/desandro/full/qxjmKM*).

Since the renderings are 2D, Zdog can render to either Canvas or SVG. Zdog is a fantastic option for 3D animations on simple objects – especially if those assets incorporate aspects of flat design. Developers who want 3D animation but don't want to get bogged down in the complex world of 3D graphics engines might find Zdog an adequate solution. Additionally, with its focus on flat images, Zdog can afford a much simpler interface and much higher performance than would be required for the 3D rendering of complex graphical images.

It does not have the elements of a game engine such as asset management and collision detection, so would need to be wrapped in a game engine or those considerations taken care of manually. This means Zdog is probably best for isolated 3D animations on landing pages.

SNAP.SVG

http://snapsvg.io/

Snap.svg says it makes "working with your SVG assets as easy as jQuery makes working with the DOM". You might be able to tell from the jQuery reference that Snap.svg is a bit older but its API does feel as easy as jQuery and that is quite a powerful thing.

Snap.svg has a clean and simple API for selecting your main SVG element and then drawing elements to it. It is most suited to developers looking for a quick solution for animating SVGs. It is a particularly good option if your animations are simple and you don't have a lot of knowledge about animation engines. While it is somewhat dated, it certainly shouldn't be overlooked, as it could be the easiest way to work with your SVGs.

Snap.svg is refreshing in its simplicity. It's scoped to the job of selecting and working with SVGs and doesn't try to be more than that. Snap could be combined with other graphics libraries here for drawing and rendering to SVGs. It is also good if you have existing SVGs and you want an easy way to work with them.

PIXIJS RENDERS TO WEBGL, SO IS USEFUL FOR ANIMATIONS THAT USE A LOT OF RESOURCES

PixiJS renders to WebGL. WebGL is an engine for doing GPU-accelerated graphics in the browser. This means it is useful for animations and graphics that use a lot of system resources and would perform best when rendered by a discrete Graphics Processing Unit (GPU). It is based on OpenGL, which is the desktop equivalent for running games and 3D graphics programs. Underneath the hood, WebGL uses the HTML `canvas` element.

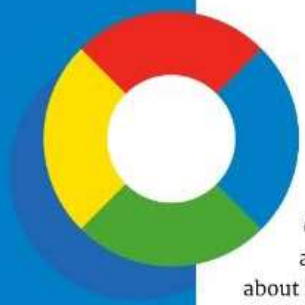
Serious graphics developers will appreciate the power of WebGL. However, these experiences may be degraded on lower-powered machines. Even as I was putting this article together, many of the PixiJS demos caused a noticeable slowdown in my system, which does not

Google design

Google is much more than its browser. Here we choose a selection of its best design and development tools to help build, create and improve your projects

AUTHOR
STEVEN JENKINS
Jenkins is the long-time editor of **Web Designer** magazine with a love of HTML, CSS and design. He is currently submerging himself in the world of infographics.

dev & tools



Working on the web usually means you will be working with Google in some shape or form. Which browser is streets ahead of the competition? Google Chrome of course. So designers and developers need to think about how their project will work with the browser. How will it look? What technologies does it support, how secure is it and how will it perform? All valuable considerations. Fortunately, Chrome provides tools to ensure any site or app will be at its best. DevTools enable designers and developers to gain insight into a web page: you can manipulate the DOM, check CSS, experiment on designs with live editing, debug JavaScript and check performance.

But Google offers more than just the browser. It has tools and resources to aid nearly every aspect of your design and development life. Want to know how to improve performance? Lighthouse is here to help. Want to build better performing mobile sites? Then say hello to AMP. Are you looking to build beautiful PWAs? Then Flutter, Material Design and Workbox are ready to step in. You'll find details of YouTube channels and extensions in this article to help you use these tools.

The beauty of using Google tools, resources, libraries and frameworks is that you know they will work well with the Chrome browser – the most popular browser on the planet.

CHROME DEVTOOLS

https://developers.google.com/web/tools/chrome-devtools

Every designer and developer knows (or at least should know) that Chrome comes with a set of tools built directly into the browser. These are ideal for inspecting the elements that make up a page, checking CSS, editing pages on the fly and much more.

So what's on offer? The Elements tab is the introduction to DevTools. It displays the HTML code that makes up the selected page. Get an insight into the properties of each div or tag from the selected page and start live editing. This is perfect for experimenting with designs. Check the Layout – whether you are using Flexbox or Grid – and take a look at related fonts with examples and examine animations.

What else can you do? View and change CSS. The Styles tab on the Elements panel lists the CSS rules being applied to the currently selected element in the DOM Tree. Switch properties on and off (or add new values) to experiment with designs. This is the perfect tool for ensuring that everything works as expected before applying any changes to the live design.

You can also debug JavaScript, optimise website speed and inspect network speed. Here's a quick tip you can use to immediately speed up your workflow. Head to the Sources tab, click New Snippet and add frequently used code. Name the code snippet and save. Repeat as needed. Now you can grab this code snippet instead of writing it again.

Like every good browser, Chrome is constantly evolving and each new release brings new features. So how do you find out what's happening? For Chrome releases, try *https://chromereleases.googleblog.com/* or *https://www.chromestatus.com/features/schedule* to get details on dates and what the new features are.

> **"The Styles tab on the Elements panel lists the CSS rules being applied to the selected item in the DOM tree"**

Lighthouse

https://developers.google.com/web/tools/lighthouse

● Performance is a key factor in the success of a site and Lighthouse is Google's tool for improving the quality of web pages. So how do you use it and what can it do? In its simplest form, you can run Lighthouse from the Audits tab and choose from a selection of options including desktop or mobile, in addition to tick boxes for performance, accessibility and SEO, to generate a final report and suggested improvements.

Polymer

https://www.polymer-project.org/

● Polymer is well-known for its work with web components but the project has now expanded its repertoire to embrace a collection of libraries, tools and standards. What's included? LitElement is an editor that makes it easy to define web components, while lit-html is an HTML templating library that enables users to write next-gen HTML templates in JS. Plus, you will also find a PWA Starter kit, the original Polymer library and sets of web components.

APIs Explorer

https://developers.google.com/apis-explorer

● Google has a vast library of APIs available to developers but finding what you need is no easy task. This is where Google's APIs Explorer steps in to offer a helping

5 must-see Google YouTube channels

Google Chrome Developers
https://netm.ag/32gJWPj
If you want the latest news about what's happening in the Chrome development universe, this channel is a must. There are hundreds of videos covering PWAs, what's new, JavaScript, the Chrome Developer Summit and more.

Google Design
https://netm.ag/33whSHX
This channel is here to give designers (and anyone else that happens to be interested) an insight into Google's design guidelines, assets, resources, events and job opportunities (*design.google*). The videos are focused around all aspects of design and are speaker led.

Google Webmasters
https://netm.ag/31fWepZ
Getting a site seen is crucial to success and this channel provides the information and tools to help you understand and improve your site in Google Search. Get Google Search news, learn about SEO and JavaScript, get short SEO snippets and see what users are asking in #AskGoogleWebmasters sessions.

Dialogflow
https://netm.ag/2VG6re5
This channel is dedicated to Google's natural language processing tool for building conversational experiences, which typically means voice apps and chatbots powered by AI. Learn how to use Dialogflow and view presentations from events. Also make sure you check out the blog (*https://blog.dialogflow.com/*).

Flutter
https://netm.ag/2MdAxSW
Stay up-to-date with best practice for the Flutter SDK, see real-world code examples, find out what's new, discover the latest features, watch speakers from Google I/O and enjoy The Boring Flutter Development Show (apparently devs do possess a sense of humour).

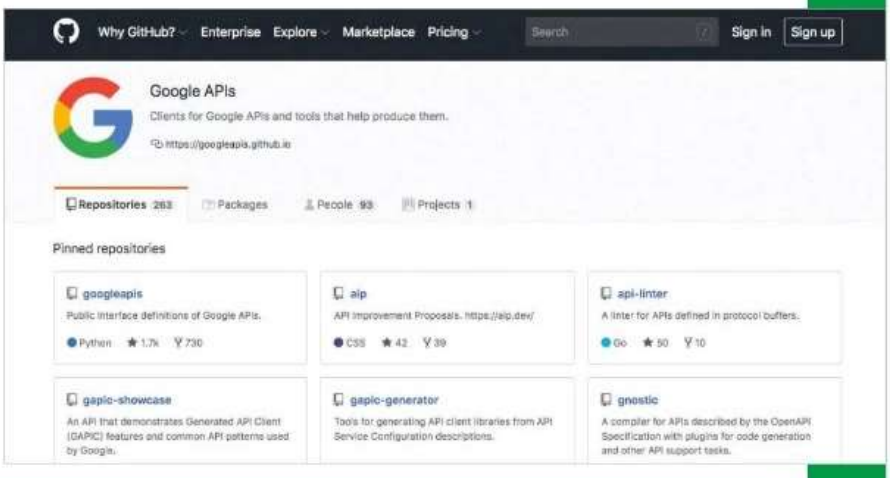
hand. There is a long list that can be scrolled through but, for quicker access, there is a search box to filter the API list. Each entry links to a reference page with more details on how to use the API.

Flutter
https://flutter.dev/
● If you are looking to build good-looking applications for mobile, web and desktop from a single codebase then Flutter could be for you. The site is a complete reference to working with and building with Flutter. Haven't got a clue what to do? The docs take a user from installation to creation, assisted by plenty of samples and tutorials.

The Google GitHub
https://github.com/googleapis/
● As most will know, GitHub is the hosting platform/repository to store and share code and files. And happily Google has its own spot on the platform with over 260 repositories to sift through. Use the filter to cut down on your search time and get closer to the repository you want to play with or contribute to.

Puppeteer
https://developers.google.com/web/tools/puppeteer
● Built in Node, Puppeteer offers a high-level API that enables you to access headless Chrome – effectively Chrome without the UI, which developers can then control through the command line. So what can you do with Puppeteer? A few options are available for generating screenshots and PDFs of pages, automating form submission and creating an automated testing environment. Check this video for more in-depth info: *https://youtu.be/lhZOFUY1weo* or take a look at *https://puppetron.now.sh/* for some more examples.

Workbox
https://developers.google.com/web/tools/workbox/
● If you are looking to build a PWA then this is a great starting point. Workbox provides a collection of JavaScript libraries for adding offline support to web apps. A selection of in-depth guides demonstrate how to create and register a service worker file, route requests, use plugins and use bundlers with Workbox. And there is also a set of example caching strategies to check out.



Codelabs

https://codelabs.developers.google.com

In need of practical guidance for a Google product? Codelabs provides "a guided, tutorial, hands-on coding experience". The site is neatly broken down into categories and events, making it quick and easy to find what you want. It includes Analytics, Android, Assistant, augmented reality, Flutter, G Suite, Search, TensorFlow and virtual reality. Select an option and get the code and directions you need to build small applications.

Design Sprints

https://designsprintkit.withgoogle.com

The Design Sprint Kit is for those who are learning how to participate in or run design sprints. It looks to cover all knowledge bases, from first-timers to experienced sprint facilitators. Learn about the methodology or jump straight into the planning stage, including writing briefs, gathering data and research, as well as what to do post-sprint. Also includes a host of resources such as tools, templates, recipes and the option to submit your own method.

Color Tool

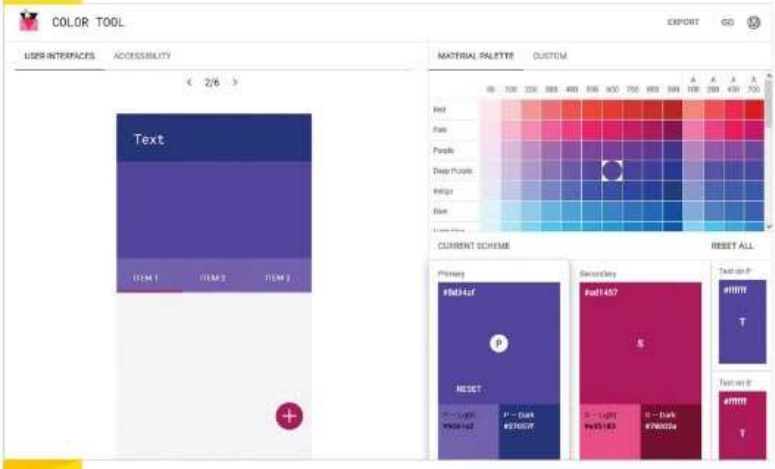
https://material.io/resources/color/

A straightforward tool that enables you to create, share and apply a palette in addition to checking accessibility. Users can choose a predefined palette from the Material palette. Simply pick a colour and then apply it to the primary colour scheme, switch to the secondary option and pick again. Finally, pick text colours for both schemes. Alternatively, switch to Custom to pick your colours. Then switch to Accessibility to check all is good before, finally, exporting the palette.

People + AI Guidebook

https://pair.withgoogle.com

This guide is the work of the People + AI Research initiative at Google and looks to offer help to those wanting to build human-centred AI products. The comprehensive guidebook is split into six chapters covering user needs, data collection and evaluation, mental models, trust, feedback and graceful failure. Each chapter is accompanied by exercises, worksheets and the tools and resources that are needed to make it happen.



5 must-try Chrome extensions

Marmoset

https://netm.ag/2BeLTzK

Code on screen is not the most attractive or exciting visual. Marmoset is here to make that code a little bit more exciting. It's simple to use; add your code, select an angle, zoom in/out, choose a filter, add effects and hit save.

LambdaTest Screenshots

https://netm.ag/2q5YvHc

Web pages need to work across all major browsers, browser versions, operating systems, mobile devices and resolutions. LambdaTest helps with this by taking screenshots across more than 2,000 real browsers running on real operating systems on cloud-connected machines.

Checkbot

https://netm.ag/2B8UrrV

This extension is a website auditor that tests hundreds of pages at once for SEO, speed, security issues, broken links, HTML/JavaScript/CSS errors and much more. The free option checks 250 URLs per site on as many sites as you want.

META SEO inspector

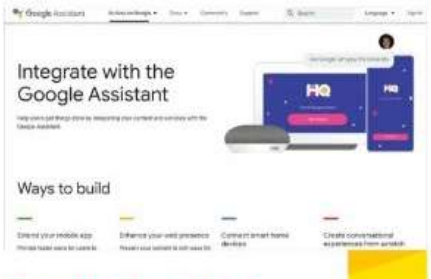
https://netm.ag/31f7Dq7

This extension inspects meta data and offers fixes. It is mainly aimed at web developers that want their HTML to follow the Webmasters' Google Guidelines. As well as the usual meta tags, the extension also looks at XFN tags, microformats, the canonical attribute and no-follow links.

45to75

https://bit.ly/2MI9ePF

This is a tool that helps designers and developers ensure that their text line lengths measure as they expect. It is a very simple tool to use – it's simply a case of highlighting the text to count then right-clicking to make the character count then appear in a tooltip.



Google Assistant

https://developers.google.com/assistant

● This is the Google Assistant's developer platform, offering a guide on how to integrate your content and services with the Google Assistant. So what does it give you help with? It shows you how to extend your mobile app, present content in rich ways for Search and Assistant, control lights, coffee machines and other devices around the home and build voice and visual experiences for smart speakers, displays and phones.

PageSpeed Insights

https://developers.google.com/speed/pagespeed/insights/

● PageSpeed Insights analyses web content and then offers suggestions on how to make it load faster. Simply add a URL, hit the Analyze button and wait for the magic to happen. Check the Docs to get a better insight into how the PageSpeed API works and how to start using it.

AMP on Google

https://developers.google.com/amp

● AMP is Google's tool for creating fast-loading mobile pages that will (hopefully) get to the top of search rankings. Learn how to create fast, user-first sites, integrate AMP across Google products, use Google AMP Cache to make AMP pages faster and monetise AMP pages with other Google products.

MATERIAL DESIGN

https://material.io

Development may be seen as Google's favoured child but, whatever you are making, creating or building, it needs to look good and give the user an experience that makes them want to use it. Material is a more recent addition to the Google stable but is a design system that has matured into a vital piece of design kit.

Like any good design system, it has its own set of guidelines, which you need to look at before stepping into the more exciting stuff. Get an overview of how to use different elements, what Material theming is, how to implement a theme and usability guides including accessibility. Elsewhere, there is an insight into Material Foundation, which includes the key areas of design such as layout, navigation, colour, typography, sound, iconography, motion and interaction. Each category reveals its dos and don'ts and where you should consider caution. To give an idea of what to expect, the Layout category offers sections on understanding layout, pixel density, how to work with a responsive layout including columns, gutters and margins, breakpoints, UI regions and spacing methods to name but a few.

Beyond the Design section is Components, which provides the physical building blocks needed to create a design. What's included here? Buttons, banners, cards, dialogs, dividers, lists, menus, progress indicators, sliders, snackbars (these are brief messages about app processes at the bottom of the screen), tabs, text fields and tooltips. Undoubtedly a comprehensive collection of components.

And developers haven't been forgotten, with details and tutorials on how to build for different platforms – Android, iOS, Web and Flutter. And, finally, there is a page dedicated to a host of resources to help make your chosen design happen.

"Material is a more recent addition to the Google stable but is a design system that has matured into a vital piece of design kit"

View source files here!

All the files you need for this tutorial can be found at *http://richardmattka.com/public/support/utils.js.zip*

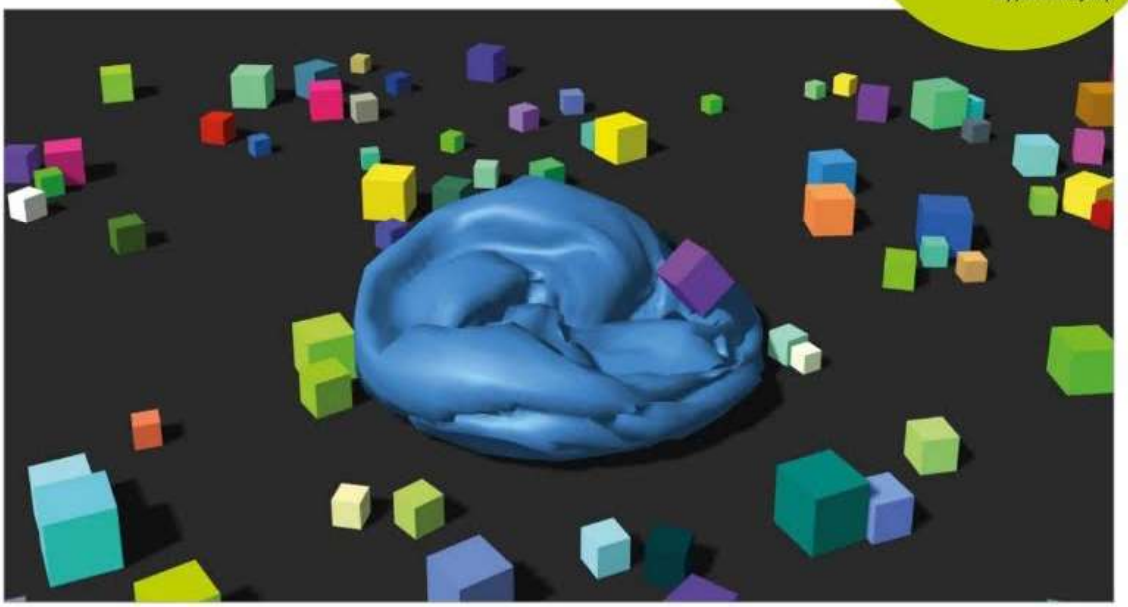
ABOUT THE AUTHOR
RICHARD MATTKA

w: richardmattka.com

t: @synergyseeker

job: Creative director, designer, developer

areas of expertise: Shaders, VFX, WebGL



* THREE.JS

UPGRADE YOUR 3D PHYSICS WITH THREE.JS

Richard Mattka continues exploring real-time web physics, this time focusing on soft-body dynamics using the three.js and ammo.js libraries

Following our piece in **net** 325, in this article we'll continue to explore physics engines and look at how you can create a 3D physics scene using three.js and ammo.js. Over the steps you'll learn how to render a physics simulation with both soft body and rigid body physics.

THREE.JS 3D FOR THE WEB

Three.js is an open-source 3D library created by Ricardo Cabello. It is feature-rich, flexible and has excellent performance, making it a great choice for WebGL development.

WHAT ARE PHYSICS ENGINES?

Physics engines are simulations that emulate the physical world. They compute how particles collide, water flows and buildings crumble. You've seen physics engines at work in virtually any video game you've played and in many visual effects for film.

For the web, we are interested in real-time physics simulations. These can include fluid simulations, soft-body dynamics, particles simulations and much more. In fact, physics simulations can attempt to emulate any physical world phenomena we want.

In the last article, we created a rigid body physics simulation. Rigid bodies do not break down or change shape when they interact. This is useful for many game interactions. But what if you wanted to simulate cloth or fluid? What if you wanted to have objects that appear soft when they collide? In this tutorial, we'll look at how to use soft-body dynamics to emulate this.

CREATE ANIMATIONS WITH THREE.JS AND PHYSICS

We'll be using ammo.js to run the physics. It's a port of the popular Bullet physics engine, which means it's powerful, accurate and very fast. We'll use

READ PART ONE OF THIS TUTORIAL IN netm.ag/issue325

three.js to render our 3D graphics but enable ammo to direct the action. Let's jump in and get started!

SET UP A WEB SERVER

It's best to run web-based code on a web server, either locally or remotely. If not, you may run into cross-domain or origin restrictions loading assets.

You can use simple tools like MAMP (*https://www.mamp.info/en/*) for both Windows and Mac that install with a few clicks. You can also check out the three.js How To Run Things Locally docs (*https://threejs.org/docs/#manual/en/introduction/How-to-run-things-locally*).

STEP 1: GET THREE.JS AND AMMO.JS

Download the latest version of three.js from the repository here: *https://github.com/mrdoob/Three.js/*.

Also, download the ammo.js library (*https://github.com/kripken/ammo.js/tree/master/builds*). You'll only need the ammo.js file.

Note: **This code has been tested on the latest release of three.js v109. You can find source files and code on ammo examples like this one here as well: *https://github.com/mrdoob/three.js/blob/master/examples/webgl_physics_volume.html***

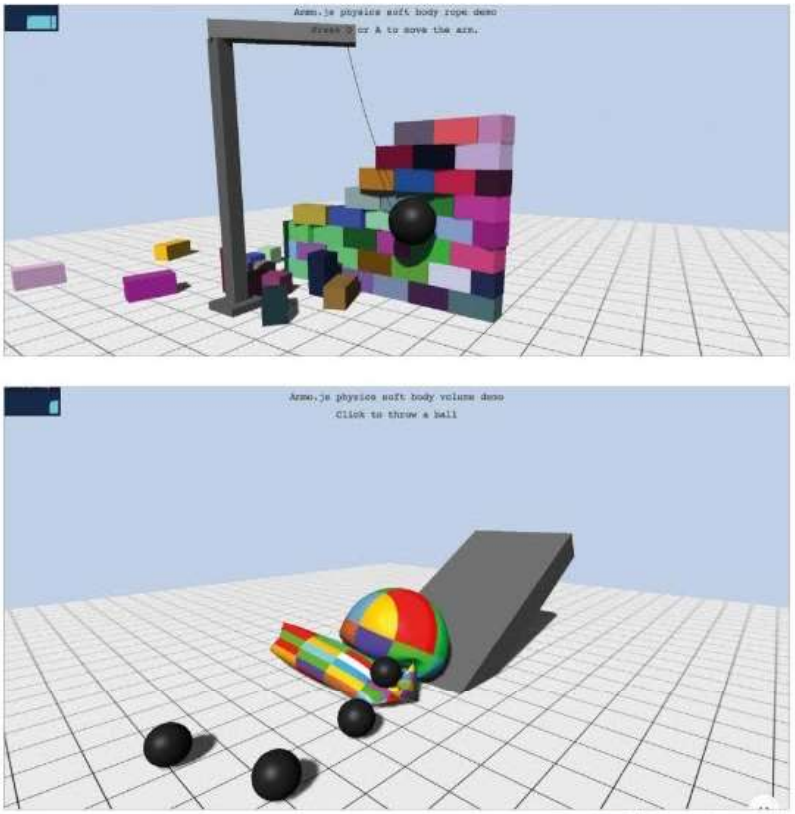
STEP 2: CREATE AN HTML FILE

Set up a simple HTML file with some CSS to ensure it fills your screen. Start by adding the following code to a new HTML file.

```
<!DOCTYPE html>
<html>
    <head>
        <title>NET - three.js + ammo.js</title>
        <style>
            html, body { margin: 0; padding:0; overflow: hidden; }
        </style>
    </head>
    <body>
        <script>
            // JAVASCRIPT CODE GOES HERE
        </script>
    </body>
</html>
```

STEP 3: ADD THE LIBRARIES

Next you need to include the libraries for three.js and ammo.js. In addition, I have created a utility file that includes a few functions from the three.js examples folder for you (*http://richardmattka.com/public/support/utils.js*). Add this code between the `head` tags after your `style` tags.



Top Physics engines enable complex simulations of real-world phenomena

Above Ammo.js handles soft-body and rigid-body physics, as seen in this demonstration

```
<script src="libs/three.min.js"></script>
<script src="libs/ammo.js"></script>
<script src="libs/utils.js"></script>
```

STEP 4: CREATE GLOBAL VARIABLES

Start adding your JavaScript code between the script tags in your HTML. You need variables for the physics engine world and the initial transformation (both position and rotation). Additionally, you need a camera, a scene and renderer for the 3D scene. On top of this, you will need the `clock` class to know how much time has passed between each frame, along with arrays to hold all of the soft and rigid body objects.

```
let world, initialTransform, scene, camera, renderer;
let objects = [], softBodies = [], softBodyHelpers;
let clock = new THREE.Clock();
```

STEP 5: CALL INITIALISATION FUNCTIONS

To initialise ammo.js, you call its root function. Do that, along with calling the initialisation functions and the animation function.

```
Ammo();
initEngine();
```

```
initScene();
animate();
```

STEP 6: CREATE CODE FRAMEWORK

Add stub code to lay out the framework of what you'll be coding. Add the following empty functions, which you will fill in afterwards.

```
function initEngine(){
}
function initScene(){
}
function animate(){
}
function createGround(){
}
function createSoftBodies(){
}
function createBoxes(){
}
function updatePhysics( deltaTime ){
}
```

STEP 7: SET UP THE PHYSICS ENGINE

Add the following code inside your `initEngine` function to set up and initialise the physics engine.

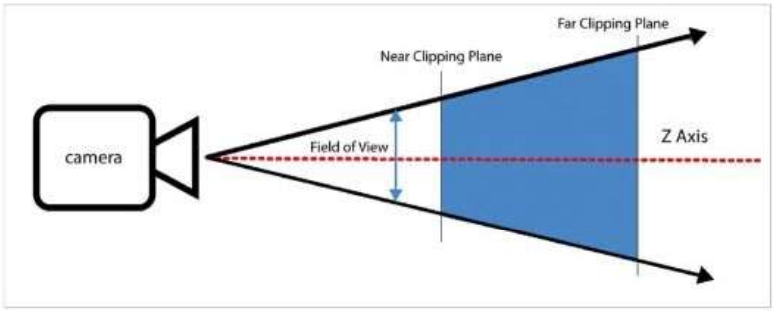
You store the initial transform for the physics world in `initialTransform`. Then configure the engine, setup a `dispatcher` to emit simulation updates, define a `cache` and the two `solvers`, one for solving constraints and one for soft bodies.

These values then plug into the creation of the `world`. Then set up the `gravity` as a 9.8 downward value and create the `softBodyHelpers`. If you want to try out other engine options, you can read more in the documentation (*https://github.com/kripken/ammo.js/*).

```
initialTransform = new Ammo.btTransform();
let collisionConfig = new Ammo.
btSoftBodyRigidBodyCollisionConfiguration();
let dispatcher = new Ammo.btCollisionDispatcher(collisio
nConfig);
let cache = new Ammo.btDbvtBroadphase();
let solver = new Ammo.
btSequentialImpulseConstraintSolver();
let softBodySolver = new Ammo.
btDefaultSoftBodySolver();
world = new Ammo.btSoftRigidDynamicsWorld(dispatcher,
cache, solver, collisionConfig, softBodySolver);
world.setGravity(new Ammo.btVector3(0, -9.8, 0));
softBodyHelpers = new Ammo.btSoftBodyHelpers();
```

STEP 8: CREATE A 3D SCENE

Next, add code in the `init` function. Start by setting up the three.js scene. This will be the container for all your objects. You can also set a default background colour for the scene.



Above The three.js virtual camera uses a field of view, near and far clipping planes.

```
scene = new THREE.Scene();
scene.background = new THREE.Color( 0xbfd1e5 );
```

STEP 9: SET UP A CAMERA

To see the scene, you need to add a camera. The `lookAt` function orients the camera to point at the centre of the scene (0,0,0).

```
camera = new THREE.PerspectiveCamera( 45, window.
innerWidth / window.innerHeight, 0.1, 500 );
camera.position.set( 0, 10, 40 );
camera.lookAt(new THREE.Vector3(0, 5, 10));
```

STEP 10: MAKE A RENDERER

Next, add the renderer, which handles drawing the scene to the HTML canvas element. You can also define the pixel ratio and size so it adapts to the screen space. Set the renderer to use shadows as well as using the `shadowMap` property.

```
renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight
);
renderer.shadowMap.enabled = true;
document.body.appendChild( renderer.domElement );
```

STEP 11: AMBIENT LIGHT

Ambient light evenly lights the scene with no shadow casting or direction. It's a way to bring up the overall visibility of the scene with low overhead.

```
let ambientLight = new THREE.AmbientLight( 0xffffff, 0.5
);
scene.add( ambientLight );
```

STEP 12: ADD DIRECTIONAL LIGHT

To make a scene look 3D, you need some type of directional light. This type of light is similar to the light of the sun, in that it has consistent intensity

at a constant angle regardless of distance. You also need to enable shadow casting for this light.

```
let light = new THREE.DirectionalLight( 0xffffff , 1);
light.position.set( -100, 150, 100 );
light.castShadow = true;
light.shadow.mapSize.width = 2048;
light.shadow.mapSize.height = 2048;
light.shadow.camera.left = -50;
light.shadow.camera.right = 50;
light.shadow.camera.top = 50;
light.shadow.camera.bottom = -50;
scene.add( dirLight );
```

STEP 13: GROUND AND BOXES

The last thing you do in the `init` function is call the three functions to set up the ground, boxes and soft body shapes for the scene.

```
createGround();
createBoxes();
createSoftBodies();
```

STEP 14: ANIMATION RENDER LOOP

Next, add code for the `animate` function. For each render frame you will update the physics engine and render the 3D scene via three.js. Use the `Clock` class to see how much time passed between frames to pass that to the `updatePhysics` function you'll create. Then render the `three.js` scene as usual. Add the following function to do this.

```
let d = clock.getDelta();
updatePhysics( d );
renderer.render( scene, camera );
requestAnimationFrame( animate );
```

STEP 15: CREATE GROUND 3D OBJECT

This next code goes in your `createGround` function. Start by defining the size and mass of the object. Then create the three.js version of the ground using a box shape, so you can see it in the scene.

```
let size = {x: 60, y: 1, z: 60}, mass = 0;
let ground = new THREE.Mesh(new THREE.
BoxBufferGeometry(size.x, size.y, size.z), new THREE.
MeshPhongMaterial({color: 0x999999}));
ground.position.set(0,0,0);
ground.receiveShadow = true;
scene.add(ground);
```

STEP 16: CREATE PHYSICS GROUND

Next create the physics engine version of the ground object and add it to the physics engine `world`. Use the following code to do this: ▶

RESOURCES

MORE ABOUT PHYSICS ENGINES

3D physics engines

I found these 3D physics engines work well with three.js and are relatively easy to use.

Ammo.js
Ammo is a direct port of the Bullet physics engine, arguably one of the most accurate simulators for games. It was not rewritten but ported directly, so it contains all the great features of Bullet including collision detection, rigid bodies, cloth simulations and complex systems.
https://github.com/kripken/ammo.js

Cannon.js
Super lightweight JavaScript 3D engine that includes cloth simulations, rigid bodies and constraints. This engine is fast, accurate and easy to use.
https://schteppe.github.io/cannon.js/

Oimo.js
A JavaScript port of OimoPhysics originally created for ActionScript 3.0. This engine is lightweight and very performant.
https://github.com/lo-th/Oimo.js/

2D physics engines

If your needs are only in two dimensions, here are a few great engines to check out.

Matter.js
A rock-solid 2D physics engine that covers rigid body simulations, elastic collisions, comped joints and much more.
http://brm.io/matter-js/

Planck.js
A JavaScript port of Box2D, a popular 2D physics engine.
http://piant.com/planck.js/

P2.js
An easy to use physics engine for 2D from the creator of the cannon.js 3D physics engine.
https://github.com/schteppe/p2.js

```
let transform = new Ammo.btTransform();
transform.setRotation( new Ammo.btQuaternion( 0,0,0,1 ) );
let state = new Ammo.btDefaultMotionState( transform );
let boxShape = new Ammo.btBoxShape( new Ammo.btVector3( size.x * 0.5, size.y * 0.5, size.z * 0.5 ) );
let localInertia = new Ammo.btVector3( 0, 0, 0 );
boxShape.calculateLocalInertia( mass, localInertia );
let config = new Ammo.btRigidBodyConstructionInfo( mass, state, boxShape, localInertia );
let rb = new Ammo.btRigidBody( config );
world.addRigidBody( rb );
```

STEP 17: SOFT-BODY OBJECTS

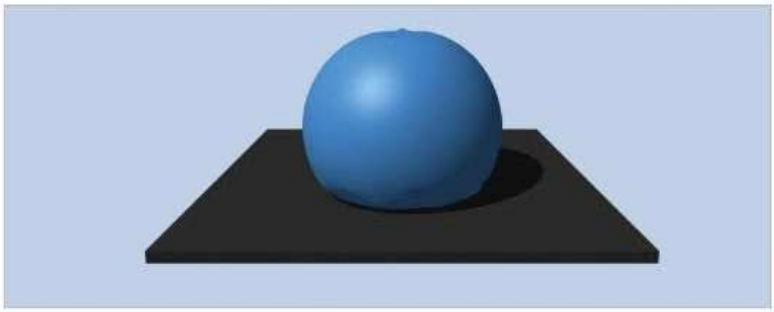
Creating the soft bodies is similar to the rigid body we just made for the ground but requires a few more steps. Start by adding this code to the `createSoftBodies` function to create the three.js version of the sphere.

```
let volumeMass = 15;
let pressure = 200;
let sphereGeometry = new THREE.SphereBufferGeometry( 10, 40, 25 );
let volume = new THREE.Mesh( sphereGeometry, new THREE.MeshPhongMaterial( { color: 0xFF0000 } ) );
volume.castShadow = true;
volume.receiveShadow = true;
volume.frustumCulled = false;
scene.add( volume );
sphereGeometry.translate( 0, 20, 0 );
```

STEP 18: MERGE SHAPE VERTICES

In order to enable the shape to deform and change, we need to get the vertices that describe our shape and pass them into the physics engine. Create a new `BufferGeometry` to hold all the position data from the sphere shape we just made. Then index the vertices and map them together using the utility function `mapIndices`, which will append the `ammoVertices` and `indices` to the sphere's geometry.

```
let posOnlyBufGeometry = new THREE.BufferGeometry();
posOnlyBufGeometry.addAttribute( 'position', sphereGeometry.getAttribute( 'position' ) );
posOnlyBufGeometry.setIndex( sphereGeometry.getIndex() );
// Merge the vertices to indexed triangles
let indexedBufferGeom = BufferGeometryUtils.mergeVertices( posOnlyBufGeometry );
// Create index arrays mapping the indexed vertices to bufGeometry vertices
mapIndices( sphereGeometry, indexedBufferGeom );
let volumeSoftBody = softBodyHelpers.CreateFromTriMesh(
    world.getWorldInfo(),
    sphereGeometry.ammoVertices,
    sphereGeometry.ammoIndices,
    sphereGeometry.ammoIndices.length / 3,
    true );
```



Top A simple ground object to stop our objects from falling forever

Above A soft, balloon-like sphere, deforming under its own weight and gravity

STEP 19: CONFIGURE THE SOFT BODY

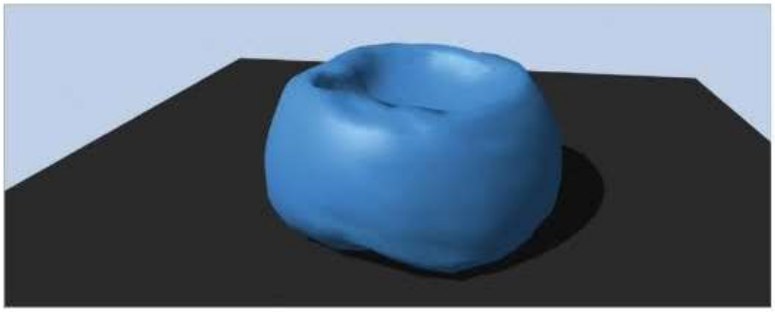
Next, configure the settings for the physics engine soft-body sphere. You can set the friction and pressure, for example. Use the following code in order to do this:

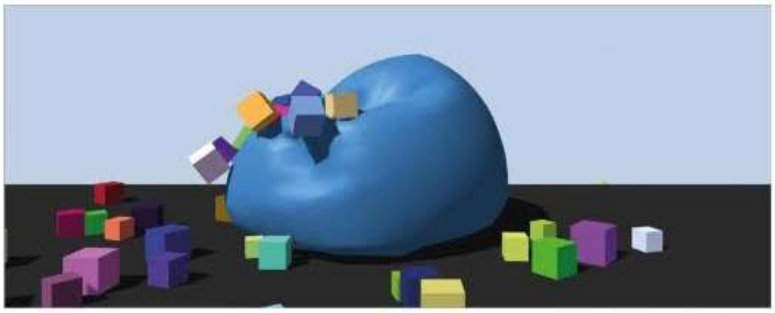
```
var sbConfig = volumeSoftBody.get_m_cfg();
sbConfig.set_viterations( 40 );
sbConfig.set_piterations( 40 );
sbConfig.set_collisions( 0x11 ); // Soft-soft and soft-rigid collisions
sbConfig.set_kDF( 0.1 ); // Friction
sbConfig.set_kDP( 0.01 );// Damping
sbConfig.set_kPR( 100 ); // Pressure
volumeSoftBody.get_m_materials().at( 0 ).set_m_kLST( 0.9 ); // Stiffness
volumeSoftBody.get_m_materials().at( 0 ).set_m_kAST( 0.9 ); // Stiffness
```

STEP 20: ADD THE SOFT BODY

Now you've created the soft body, you can add it to the physics engine world using the following:

```
world.addSoftBody( volumeSoftBody, 1, - 1 );
volume.userData.physicsBody = volumeSoftBody;
volumeSoftBody.setActivationState( 4 ); // Disable
```





Top A falling soft-body sphere changing shape with real-time physics!

Above Blocks from above crushing and destroying our soft-body sphere!

```
deactivation
softBodies.push( volume );
```

STEP 21: UPDATE THE PHYSICS

The last function you need to define is updatePhysics. The following code goes in this function. Each render frame you'll be updating the physics world in parallel with the normal 3D scene render.

Start by stepping the physics engine forward using the deltaTime you passed in from the animation function. Then loop through the array of softBodies. So far we made just one sphere soft body but this code will handle as many as you want. Later on you can go back and add some more soft bodies following the model in the createSoftBodies function.

```
world.stepSimulation( deltaTime, 10 );
// Update soft bodies
for ( var i = 0, il = softBodies.length; i < il; i ++ ) {
}
```

STEP 22: INDEX AND VERTICES ARRAYS

Inside the for loop you just made, get references to the positions, normals and indexes to map the physics object's vertices to the 3D object's vertices. Then create another loop that will iterate over the vertices you need to update. Essentially, you are updating the vertices of the object to reflect the physical change in the object's shape and position.

```
let volume = softBodies[ i ];
let geometry = volume.geometry;
let softBody = volume.userData.physicsBody;
let volumePositions = geometry.attributes.position.array;
let volumeNormals = geometry.attributes.normal.array;
let association = geometry.ammoIndexAssociation;
let numVerts = association.length;
let nodes = softBody.get_m_nodes();
geometry.attributes.position.needsUpdate = true;
geometry.attributes.normal.needsUpdate = true;
for ( var j = 0; j < numVerts; j ++ ) {
}
```

STEP 23: REFERENCES TO X,Y,Z

Inside the new for loop you will need to get the references to the positions in x,y,z and the normals. Then create a final loop to iterate the mapped indices and vertices to connect everything together.

```
let node = nodes.at( j );
let nodePos = node.get_m_x();
let x = nodePos.x();
let y = nodePos.y();
let z = nodePos.z();
let nodeNormal = node.get_m_n();
let nx = nodeNormal.x();
let ny = nodeNormal.y();
let nz = nodeNormal.z();
let assocVertex = association[ j ];
for ( var k = 0, kl = assocVertex.length; k < kl; k ++ ) {
}
```

STEP 24: UPDATE THE 3D GEOMETRY

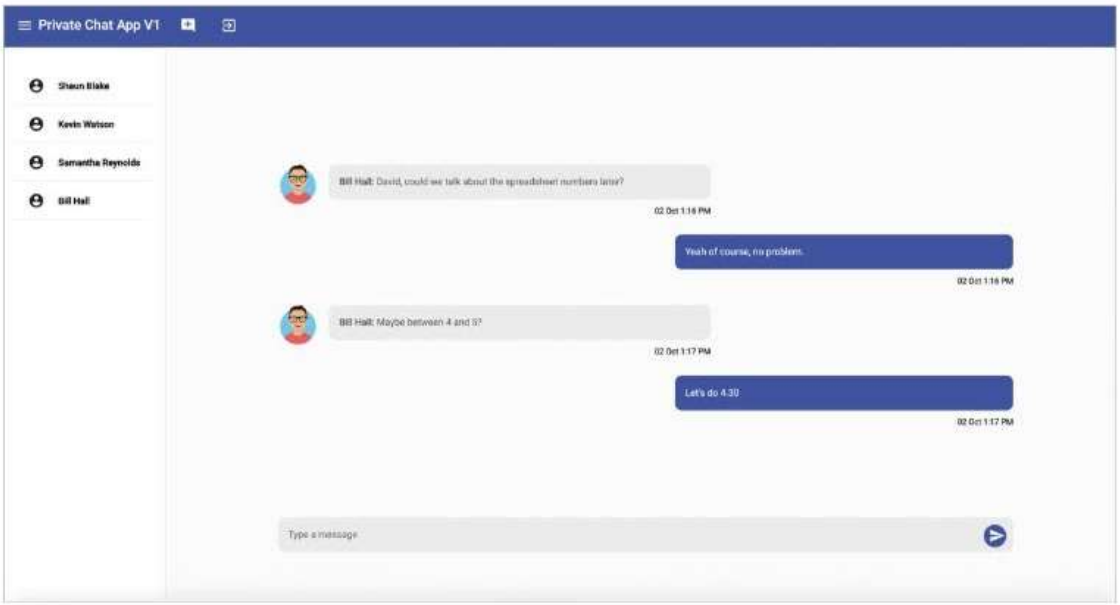
Now you can update the 3D sphere's individual vertices position and normal faces using the physics engine's values. Add this code inside that last for loop you created to wrap things up.

```
let indexVertex = assocVertex[ k ];
volumePositions[ indexVertex ] = x;
volumeNormals[ indexVertex ] = nx;
indexVertex ++;
volumePositions[ indexVertex ] = y;
volumeNormals[ indexVertex ] = ny;
indexVertex ++;
volumePositions[ indexVertex ] = z;
volumeNormals[ indexVertex ] = nz;
```

WRAPPING UP

Once you run the code, you'll see your soft-body sphere – looking much like a balloon – fall to the ground and deform. You can tweak the values to dial in the features of this shape. If you download the associated source code, you'll see I combined the last tutorial with this one to drop some rigid body blocks onto the scene as well. You can now add more soft bodies and rigid bodies to create complex scenes!

ABOUT THE AUTHOR
DARRYL BARTLETT
t: @darrylabartlett
job: Web and mobile developer
areas of expertise: JavaScript, PHP, mobile



ANGULAR AND FIREBASE

CREATE A PRIVATE CHAT APPLICATION

Darryl Bartlett shows you how to build a simple one-to-one chat app using Angular and Firebase

In this tutorial, we will be developing a simple chat application using Angular and Firebase, where users will be able to send each other private text messages. To get started, download the project files (*https://github.com/darryl-bartlett/Chat1to1Angular*).

FIREBASE SETUP AND CONFIGURATION

Head over to *firebase.google.com* and sign up for an account if you don't already have one. Create a new project inside the dashboard, go to 'Firebase > auth' and then enable email/password. You will also need to set up a Firestore database in test mode to write data.

You will have been given some configuration details when you set up a new project. You will need to add these inside the environment.ts file.

COMPONENTS AND SERVICES

There are three components, one for login, one for signup and one for the dashboard. We also have two services (api.service.ts and auth-service.ts). These are used for the Firestore and Firebase authentication logic.

SET UP REGISTER/SIGN IN

The user will need to register and/or sign in so that they can use the application. These functions are already set up and ready to go. One thing to note about sign in is that `currentUser` is the given instance of the logged in user. In order to maintain it, we keep it in the service (api.service.ts). We save the `uid` inside local storage upon sign in, which we subsequently use to fetch the user data and maintain it in `currentUser`.

HOW CHAT WORKS

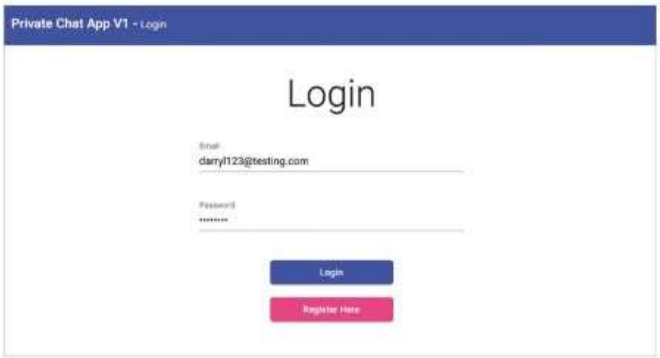
For a one-to-one chat, we're going to need to create a common reference point for both users. This will be called `chatId`. One of the users will

"For a one-to-one chat, we're going to need to create a common reference point for both users"

initiate the chat and the other will be the receiver but to achieve it we will have another collection/table in our database called `conversations`. Similarly, every user will also maintain an array of objects called `conversations`, which will basically contain the information of every other user they are in conversation with.

The add button in the top bar will bring up a dialog that enables the user to start a new conversation. The dialog contains a list, which will be populated through a function called `getAllUsers()`. When the user selects someone to chat with, the selected user is passed into the `selectUser(user)` function, which will first check if User A hasn't already talked to User B by searching against the latter's `uid`.

```
let convo = [...this.api.currentUser.conversations]; //
Spread operators for ensuring type Array.
let find = convo.find(item => item.uid == user.uid); // Check
if spoken before
if (find) { // Conversation Found
 this.api.getChat(find.chatId).subscribe(m => {
  this.temp = m;
  // Set the service values
  this.api.chat = this.temp[0];
  this.messages = this.api.chat.messages == undefined ?
[] : this.api.chat.messages
  this.showMessages = true;
  setTimeout(() => {
   this.triggerScrollTo() // Scroll to bottom
```



Above Here is the login screen where users will be able to log in to the app using their email and password

FEATURE LIST

CHAT FEATURES TO CONSIDER

Below is a list of features you might consider implementing in a full-scale chat application:

Authentication Your app will need to know the identity of a user. Firebase offers users several sign-in methods, including Facebook and Google.

Group chat Users should be able to have group conversations, not just one-on-one chats.

Encryption Security is really important inside any application, especially a chat application that will store personal messages and even private data. Make sure you're using encryption.

File sharing and media Users should be able to share images, videos, audio and documents. You can use something like Firebase Cloud Storage for this. You should also consider adding the option for stickers and GIFs.

Notifications Users will want to know when they have received messages, so implementing notifications is a must.

Voice / video calling Enabling users to send / receive calls has become quite a popular feature inside chat apps like WhatsApp and Facebook Messenger.

Read receipts Users like to know when their messages have been delivered and read.

Message expiry After a particular period of time, messages will no longer be viewable to the recipient.

Location sharing This could be used for friends to share their location with each other.

Ecommerce You could enable your users to be able to send money to each other through the application.

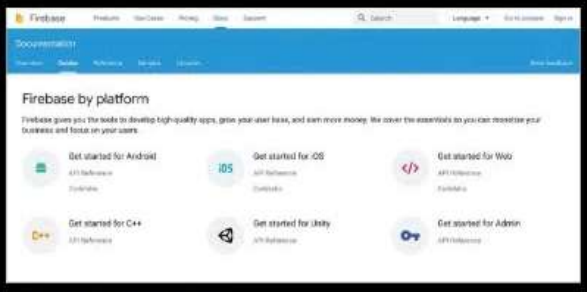
Chatbots Useful for providing support and information to the user.

Availability across all platforms Users will want to see their messages at home and also while they are out and about, so it's worth making sure that your application is available on as many devices as possible.

RESOURCES

MORE INFORMATION

Here are some resources to help you build your own chat applications for the web.



Firebase documentation
https://firebase.google.com/docs
You should take some time to learn Firebase through the documentation provided on its website.

Udemy: Build a Slack App
https://www.udemy.com/build-a-slack-chat-app-with-react-redux-and-firebase
If you are looking to build a group messaging app then you should try looking at this course by Reed Barger, which walks you through building a Slack-style app using React, Redux and Firebase.

OneSignal
https://onesignal.com
Take your chat application one step further by providing web notifications for your users when they receive a message. OneSignal provides a great notification service for all platforms and there is a free tier.

Angular Material
https://material.angular.io
A suite of pre-built UI components based on Google's Material Design specification. This includes things like buttons, progress bars and tabs.

How to Add End-to-End Encryption to Your Angular App
https://www.youtube.com/watch?v=tBpplt8WLxY
A great starting point to learn about end-to-end encryption for your app.

Bootsnipp
https://bootsnipp.com
A selection of design elements and code snippets for Bootstrap that you can use for your own web projects. Handy for prototyping or starting out.

```
  }, 1000);
  return
 })
} else {
 /* User is talking to someone for the very first time. */
 this.api.addNewChat().then(async () => { // This will create a chatId Instance.
  let b = await this.api.addConvo(user);
 })
}
```

When User A initialises a conversation with User B for the first time, it will cause the following two things to take place.

CREATE NEW CHAT

Inside the `addNewChat()` function, `this.afs.createId()` will provide us with a unique random Firebase identifier, which we will use to create a new document in the conversations collection. This includes a `chatId` and an empty array of messages. This will also be maintained in `this.chat`.

> "We will update User A's and B's data, checking first if a conversation already exists"

```
addNewChat(){
 const chatId =this.afs.createId();
  return this.afs.doc('conversations/'+ chatId).set({
   chatId: chatId,
   messages:[]
  }).then(()=> {
   this.chat = {
    chatId:chatId,
     messages:[]
   }
  })
}
```

UPDATE BOTH USER REFERENCES

Now we have the `chatId` reference as the common link for User A and User B, we need to update the conversations array for both User A and User B under `users > uid` with the `chatId` and details of whom they talked with.

We will create two references called `myReference` and `otherReference`, which are used to set up Firebase references for both users. We also create `userMsg` and `otherMsg`, which are used to store the conversation information (`name`, `uid` and `chatId`). We will then

update both User A's and User B's data, checking first if a conversation already exists between the two. If it turns out that there is already a conversation, then our next step will be to update the conversation list. Otherwise we will go on to create a new reference.

```
async addConvo(user){
 // Data to be added
 let userMsg ={name:user.name, uid: user.uid,chatId: this.chat.chatId}
 let otherMsg={name:this.currentUser.name, uid: this.currentUser.uid, chatId:this.chat.chatId}
 // First set both references.
 let myReference = this.afs.doc('users/'+ this.currentUser.uid);
 let otherReference = this.afs.doc('users/'+ user.uid);
 // Updating profile
myReference.get().subscribe(d=>{
     let c=d.data()
     if(!c.conversations){
      c.conversations = [];
     }
     c.conversations.push(userMsg);
     return myReference.update({conversations: c.conversations})
  })
  // Updating other user profile
  otherReference.get().subscribe(d=>{
   let c=d.data()
   if(!c.conversations){
    c.conversations = [];
   }
   c.conversations.push(otherMsg);
   return otherReference.update({conversations: c.conversations})
 })
}
```

SENDING A MESSAGE

When the user presses the send message button or hits enter on the keyboard, it calls sendMessage(). We first check to make sure that the text input contains a string, before setting the message object with the senderId (user's uid), senderName, timestamp and the content of the message.

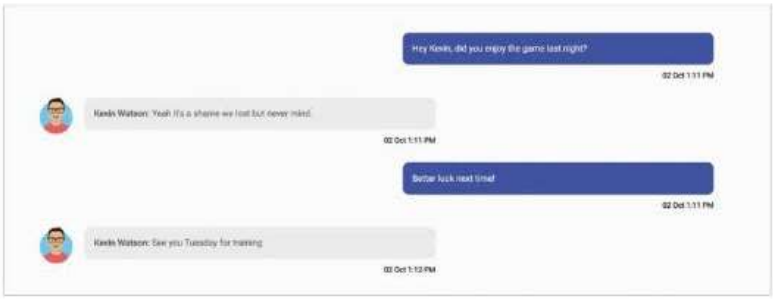
As mentioned earlier, we will be maintaining the current conversation information inside this.chat, which means that we will simply have to update the array of messages.

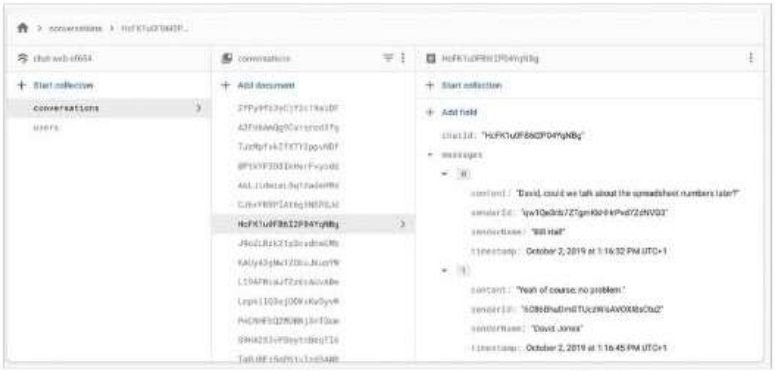
```
sendMessage() {
 // If message string is empty
 if (this.message == '') {
  alert('Enter message');
  return
 }
 // Set the message object
 let msg = {
  senderId: this.api.currentUser.uid,
  senderName: this.api.currentUser.name,
  timestamp: new Date(),
  content: this.message
 };
 this.message = '';
 this.messages.push(msg);
 this.api.pushNewMessage(this.messages).then(() => {
 })
}
```

Finally, our new message gets pushed to Firebase through pushNewMessage() inside api.service.ts. Conversations in Firebase are stored inside 'conversations > chatId > messages'.

```
pushNewMessage(list){
 return this.afs.doc('conversations/'+this.chat.chatId).update(
  {messages: list}
 )
}
```

Now you have all the information you need to set up your own private text message system for friends, clients or anyone else!





Top When you are receiving messages from another user (in this case Kevin), it will display their name and their avatar

Above This is how the messages are stored in Firebase. You will need to use encryption if building a full-scale app