PHILIPPE LEEFSMA: All right, guys, let's get started. So welcome to this presentation. So here is the idea of this presentation. So viewing your 3D model on the browser or on your mobile application is great, but visualizing the CAD data is not just about nice meshes and pretty texture. So I wanted to show that there is more to it.

So in this class, I tried to put together a set of examples to show you how you can use JavaScript to leverage the data available in the Forge Viewer, and how you can connect your application to external database, custom database, third-party web services, leverage some JavaScript frameworks and libraries to really create a rich and powerful visualization experience to your users.

So a basic web development experience is preferable, but you don't have to be JavaScript developers in order to attend this class. So you may just look at it as a set of examples to give you idea, inspiration to create your own web application.

All right. So because we focusing on cloud, so we had-- the display is not so bright here. Maybe-- does it have-- no. Doesn't make any different.

So PowerPoint is more like a desktop product. That's more the past, if I can say. So today I used a new support to present this class, and you can see this is running in the browser. So this is Chrome here. I run local host, cause-- just for safety reason, like some of the demos run locally. It's just faster to load.

And I'm using a framework called Reveal.js, which is a JavaScript framework that lets you create presentation in an HTML page. So what you see here is HTML, JavaScript, and it gives you the flexibility to write some custom code. Like this one is just a small JavaScript and SVG application that I wrote that display those animations.

And as you will see in the rest of the presentation, you can integrate seamlessly demos based on webpages in the viewer with the slides.

AUDIENCE: [INAUDIBLE]

PHILIPPE LEEFSMA: It's Reveal.js. So my name is Philippe Leefsma. I'm working for the Forge partner development. This is the cloud branch of ADN, the Autodesk Developer Network. So we are a small team of engineers providing support for the Autodesk desktop APIs, and todays and

nowadays, we provide support for the web services and the cloud APIs of Autodesk.

So let's get started. Here is the agenda for this presentation. I was breaking it down into three main topic. The first one is-- whoops. Here we go. Sorry about that.

The first one is accessing the Forge design data. So in this part, I will show you how you can access the data, which is available directly in the viewer as part of the API.

In the second part, I will show you how you can customize the UI, so how flexible it is to customize what you have in the viewer, apply some overrides to your model, colors, information, and so on. And the third part, how to connect your application to the cloud to bring some more content, bring some more data, [INAUDIBLE] databases and web services.

So let's start with the first part, accessing the Forge design data. So here is an example of our viewer. So maybe some of the people-- probably most of the people have seen it. Maybe some of you haven't seen it already. So for those who have no idea, here is a quick recall about the technology stack, just to give you some idea about what we're dealing with.

So till last year, this was called the View and Data API, so we rename it to have a more modular approach, more consistent name, because this was something between a web component and an API. So now we have the viewer which is really a web component, and then we have a set of several APIs, so those are like file management, data management API, model derivative API, which are REST services, so authentication with OAuth2, and that lets you manipulate your files, so upload your CAD design, translate the CAD design into this web-viewable format.

So we support large amount of different file formats. And so I'm not going to focus on those APIs today, because that would be too much content. I'm going to focus on the fun part in this presentation, which is the client side JavaScript API.

So we have this viewer component, which is based on HTML5, WebGL, and Three.js. And basically it lets you control and customize the behavior of the viewer inside your own web application. You can access data, components, play with camera events, and so on.

So I just show you a quick example of what we have out of the box in the viewer. So basically, this model comes from Inventor, from an Inventor assembly. And what we get out of the box, once we converted this set of files into this web-viewable format, we get the model hierarchy,

so that's the structure of our model.

So we can see-- so based, obviously, of what we have in the original CAD file, we get all the knowledge of this hierarchy of components. So I can select one of the components. Each component can have potentially one of several level of children. And you can select all of them from this tree.

The other thing that comes with it, and that's why it was named View and Data API initially, it's because you don't only extract the geometry, but you also get the data which is associated with each component. So here, I can select the component here, and you can see there are the data which are associated with the specific component that comes from the original model.

All right. And the other thing that we can do, so this is the [INAUDIBLE] effects, usually you can explode the model based on the hierarchy. So you can see it's not just the external shell, but it has a lot of component inside it.

And then we've got basic measure. So this thing does not understand things like nerve surfaces, or cylindrical surfaces. But still, you know, you can recognize planar faces. You can snap to edges and vertices. And you can measure these things inside the viewer directly. So here I can snap from one face to a vertex, or I can measure the distance between those vertices, and so on.

And another cool feature is the sectioning tool. So you can create a section plane on the model, and you can dynamically move the section plane in 3D, and to see the interior of your design or the interior of your building and things like that.

So yeah, that's the basic feature that come out of the box with the viewer. So here is a quick snippet on how to set it up. It's pretty straightforward. The only thing that you need on the HTML side is a div.

So you create a div inside your HTML. So in that case, I create a div with the viewer ID. And then here is just some pseudo JavaScript code that shows you how to initialize the viewer. So it's pretty straightforward.

So here, I get a URN, which is a Unique Resource Identifier for a specific CAD design that I uploaded through the REST APIs. So basically, I upload my files to the Autodesk cloud, and then I convert that to the viewable format, and I get a URN. And I will use this ID to load the model inside the viewer. So here I pass to the initializer method the URN, and a couple of

other option that I just remove for simplicity.

And then once it gets initialized, it will call back this function. And the only thing you have to do is to grab this div by ID here. So you will access this dom element, and create a viewer, and your viewer object, using this container, and load your model. And then the API will take care about the rest, so initializing the WebGL context, and create the 3D scene, so you don't have some complex set-up to do. This is pretty straightforward.

And when your model is loaded, or even, like, in that case before the model is fully loaded, you are able to access the API of the viewer. So in that case, for example, you want to change the background color of the viewer, or you want to load some customization, you can load your custom extension, which is just a small script that contains some more code to customize the behavior. So that's basically how it works.

So let me get back to our viewer. So here is the first thing we might want to do when we want to create some application on top of the viewer using the API is to create a hierarchy of component in order to access the properties of those components and to manipulate those components inside a viewing application. So that's the basic viewer component model that we can create.

So here is a basic JavaScript object with the following properties. So we have the name, which is the display name of the component. We have a DB ID, so it's a unique ID that identifies the component inside this hierarchy, and this doesn't change. So we can use this ID to map to your own database or to attach some custom property from somewhere else.

Then we have an array of fragment IDs. So this, I will show you later how you can use that. So basically, one component in the viewer can be represented by several meshes, and those are the IDs which lets you access programmatically those meshes to apply some overrides.

You can keep track of the parent ID, if you want to go above in the hierarchy. And then each component can have one or can have zero or several children, so an array of children nodes.

So here is a very simple method that shows you how you can build a tree that gives you access to this structure, and that create a tree of those component. So we start with the model, so we access the model from the viewer.model. Then we access the instance tree.

From the instance tree, we will get the root ID. So get the root ID of our model. We can build

the root node as follows, so root ID, the name of the node using the instance tree get node name. And then we will call this recursive method, passing the root node, and that will build the whole structure of the model.

So here is the source code of that build model tree. We pass a node, and then from the instance tree, we invoke the [INAUDIBLE] children. So we pass the ID of the node, and this method takes a callback function that give us the ID of the children. And from there, we can initialize this children collection. So if it contains nothing, then it will be initialized to an empty array. Otherwise, we just don't change it.

We create the child node. We push the child node inside the array of children. And we recursively build that method. So it will go down the tree and build the complete structure down to the children, and makes it available to your application.

The other thing that you can add to this method is the array of frag ID. So you will see in the later demo what it's useful for. But basically, we create an empty array here, and we call this method [INAUDIBLE] fragment, passing the ID of our node, and we will push so it gets a callback function that contains the fragment ID. And we push that to the array.

So here is how look the function with both, you know, first, that's the recursive function. So first we create the frag IDs. Then we create all the children nodes. And we call the function recursively.

So this is how it looks concretely in the viewer. So here, let me open up the debugger. So here I already have this extension, this code, loaded in that demo.

So here, if I simply move the camera here, you can see that it calls this exact same code that I showed you. You can see here it's loaded from a custom extension. And what it builds, it will build an object that contains this whole hierarchy.

So we can look here. The first node is actually the V8 engine. So that's what we got here. And if I look, for example, here, the fourth children in the list, here, one, two, three, here I can see this is the piston ring AC2, so that's what I have here, and it has five children. And here are the children, and so on.

So basically, give me a programmatic access to this structure of components.

The other thing that you may want to grab from the API are the component properties. So

we've got this model.getproperties. We pass a node ID, and then it returns a callback function that contains all the properties that are associated with the specific component.

And because those property lies on a SQL Lite database and they are on the cloud and they are fetched as needed by the viewer as we request those properties. So if you need a bunch of properties, let's say, for 100 components, you don't really want to call this method 100 times. It's going to take more time that you need. So there is a model.getbulkproperties where you can pass an array of IDs, and here can filter.

Let's say you're only interested in the material and the designer fields for that component. This will return all these array of properties containing only those specific properties that you requested for all the components. So it just makes your application more fluid and faster than requesting individually each component.

So again, if I switch back here, I've got another customization loaded inside this viewer. And you can see, if I select one of the components, it dumps all the properties. So it's calling that getproperty code, and basically it dumps to the brother console the same property that you can see in this panel.

So for example, let's take the material. So this is what you get from the API. You know, display category, the display name, material, [INAUDIBLE] plastic display value, the type of the property, and so on.

So just to show you basic access to the model structure, to the properties. And so basically, so just based on those two things, you should be able to create an application that looks like that pretty much. It doesn't really use much more of the API that when I just show you. It already gives you a good idea about the kind of visualization experience you can provide through the viewer.

So what we have here is a Revit, is a model exported from Revit. So that's a building. And you can see here we created this mapping between the properties of each component and this pie chart. And here, so for example, here we checked all the components which have these specific properties. And we just keep track of which are the IDs of those components.

So whenever I click here, a specific segment in the pie chart, then I can very easily interact with the viewer. It already knows the list of node IDs, and this is just a single function call, so I isolate those components based on those ideas. So it's pretty easy to have interaction

between 2D graphics and the viewer and create those kind of visual feedback, visual reports, in your web applications.

The other cool thing that this application is doing, well, it's doing a bunch of things. I'm not going to demo everything. It's using a Revit model that contains at the same time 3D views and 2D sheets. And you can see that it's syncing-- basically, whenever I select an object in the 3D view, it highlights the same object, the corresponding object entity, in the 2D drawing. And also if I select something in the 2D drawing, then it highlights and isolates these components in the 3D model.

And it's pretty straightforward to achieve that. It's just, basically you just listen to the selection event. You get an ID either in the 2D or the 3D, and you just select the same ID, you know, because they come from the same project, that the IDs are synchronized. So it's the same ID in 2D and 3D, and pretty straightforward to achieve this kind of application.

All right. So I wanted to show you that as an introduction. So let's continue with the part two. So in this part, I want to show you how flexible is the technology, and how different is the situation comparing to desktop. Create a desktop [INAUDIBLE] when you're really limited to what the API can do. In the case of this technology, it runs in the browser. It's WebGL and Three.js, so it's really open to any kind of customization on top of the graphic interface.

So as I mentioned previously, the viewer is just a 2D div where the Three.js, the 3D [? Canva, ?] the 3D graphics are rendered. So you can embed and you can overlay on top of that div any dom elements, any kind of HTML elements you want.

And you can style the viewer container as you want. You just have to place the div on your webpage. You can make it look like a rectangle, like a circle, or like fullscreen. And you just style it as you want using simple HTML and CSS.

So one of the feature that you can leverage in the API to add some custom elements on top of the viewer is this docking panel. So here, if you are familiar with JavaScript and ES6, I'm using the ES6 syntax. That's the new, the latest features of the language.

So instead of integrating the prototype, I'm just extending that base class, so this docking panel. And so they're just a small snippet, but you get the idea. You just have to call the base constructor of that docking panel, and here you can append to the container any dom element you want. So you can achieve result that look like that.

So here is another example. And this is a custom panel. So in a second, I'm going to show you what show up here. But I just want to focus on the container at the moment. So you can see it provides more than just a webpage experience. It's really like a desktop application or web application experience where you can create custom control here.

That's a dropdown where you have your various elements. This one, or so this top manager is a custom component I wrote in JavaScript and CSS.

So basically, you can inject, you can place inside this dialog pretty much anything you want. And I added another feature. When you double click here, you simply minimize the dialog, so you can interact with the model very quickly. You don't have to close the dialog.

So that's one of the tool that also is provided by the API, you know, as a class.

And then another thing that you may use also pretty much out of the box is deriving from the viewer property panel. So this is deriving from the previous panel, except that this is the panel that contains the properties in the viewer. And in that case, you can override the set property methods to inject your custom properties inside this original panel.

So in that case, just to demo here, I just put some dummy properties. But you could very well grab those properties from a web service or from a database, and you create some kind of seamless experience for the user which is used to work with the viewer. So in that case, you can see here I've got another extension loaded. When I display that here, I have the native properties of each component. And here I'm adding some hard-coded properties on top of the existing properties.

So here there is a text property. Here there is a link property. So if you click there, that will takes you to the Forge website. The file property, so I can click there, it will download the file on my disk, and also an image property. So I can click there. It's also download the image.

But you get the idea. You know, you can very easily put your website here, link to your website, or let the user download some attachment to the part, like let's say a PDF or anything. So pretty flexible to inject-- with HTML and JavaScript, inject anything you want, pretty much, in those dialogs.

All right. So talking about overrides, and this is when the fragments come into play. So I show you previously how you can grab-- for each component, you can grab the list of fragments for

this component. And this is how you can use them.

In that case, what I will do is I will apply some custom material on top of the-- affect some custom material to the various components. So here, because the viewer API is just based on Three.js, so you don't have to learn a whole new API. If you know Three.js, you can reuse most of the feature which are available in Three.js.

So in that case, I create a foam material. I set the colors. You can set a set of other properties of your material.

And then you will get the fragment lists. There is this method they expose called set material. You're passing the fragments, passing the material, and update the scene. And it will change the material of your component.

So that's basically what this dialogue is doing here. So here, I can select a property. Let's say I want to select the various materials. And you can see it will affect a specific color based on the material to each component. And then I can interact with my model. Those are plastic. You know, those are rubbers, and so on.

So this pie chart is created with a pretty nice graphic library called D3. It can do a lot of things, so it's pretty powerful. Here is another example, so doing the same thing, but with a bar chart. Also hooked up to the viewer and can interact with the various component.

So this one is just, I was just playing around. This is called a Forge graph. So you can see it can now keep the things together. It's just like another feature of the library. The only thing I have to do is pass this hierarchy, to this Forge graph, to this method, and then it will create that graph for me. And it enters the interaction between the node and so on.

So here I can use the property, for example, like the mass. And here, in case of a property which is discrete, like the mass, you can see that the size of each element of each node depends of the value of the mass. So here, at a quick glance, you can tell-- I can tell that this node is going to be much heavier than that node, for example.

This one is just another example, as well, called the circle graph, where I can click the various elements and I zoom in, zoom out. It's just for fun. I just wanted to play around with the various tools that this library can offer.

All right. So that's how you override materials. Another approach, if you want to change the

color of your components, is to use an overlay. So in this previous example, I'm really changing the material of the component.

In this example, what you can do is get a render proxy. Then you will clone the geometry, copy the position of the components, and add the component as a 3D overlay. So what it's doing is really like, create a clone of the component in 3D, and overlay that to the existing component.

So it's faster, but at the same time, one of the drawback is that if you move the components, so for example, in the case of the explode, then here I am just setting the material of the component. If I use an overlay, then obviously I will have to programmatically move the transform my overlay, if I want to keep them in sync with the original component in the viewer.

So those are like color overrides. You can also apply transforms override. So in that case, you're still using the fragment ID. But in the previous example here, we are getting the render proxy. In that example, we're getting the fragment proxy.

So this allows us to apply transformation like 2D to the component. So we initialize the fragment proxy transform, and then you can modify the position of your components, so setting the xyz. And also you can modify the rotation of the component by creating quaternions. So it's a bit like a transformation matrix, but it's a vector representation of a transformation matrix. It just has four components instead of 16. And so basically, if you want to rotate the component, you will tweak those values, update the proxy, and update the viewer, and then you can see things moving, the component moving in the viewer.

So one of the example here is also loaded here in that model. So I wrote those extension translate tool and rotate tool. So here I can select a component, and I get this manipulator here, where I can constraints around the specific axes or a specific plane. And I can move this component along these axes, along this one, or inside that plane.

That's the translation. The rotation is there. And here you can select one of the circle here, and it will rotate your component around the specific axis.

All right. So that gives you an idea about how you can apply overrides to what you get out of the box in the viewer. So here is a demo from a customer called Dotty. So it's a little startup. They are doing various things, and they're using our cloud viewer. So let me open the link here.

So basically, what they did is they created a web application where you can upload your files,

and then they provide some services on top of the viewer to add some value to what you can do with the viewer. So in that case, if you wonder what is this thing is, that's a robotic arm. And they're using that-- you can see here that's-- they're working with the oil company, and those guys may have pipes under the sea, deep under the sea, which have leaks. And in that case, they are sending down that structure, that robot, along the pipes. And then they are remotely controlling the robotic arm in order to seal some clamp around the joints of the pipe.

So what they have, actually they created-- all on top of the JavaScript API, they created a tool that lets you author animation. So there is no back-and-forth between the CAD model and the thing that we have in the browser here. Basically, you upload your model to their cloud. And then from using their tool, you are able to create animation sequences of your models.

So you can see here, in that case, whenever I click a specific state here, those states are stored in their own database. And they keep track of the transformation that are applied to each component. And whenever I click to a specific state, it not only restore the camera position, but it also restore transformation for all those components.

So you can go through each of the components, and you can play back the animation here. So it's pretty easy to create, to author those animation, animated sequences. And also it's pretty easy. I can go from one state to another. It will just interpolate the position of the component, the current position of the component to the next one. Bolts in, bolts out, and so on.

So here is a good example of really, like, value added on top of the viewer. All right. The next thing I wanted to show you, the next option, the next tool that you can use in order to create some more advanced customization on top of the viewer is SVG. So SVG stands for Scalable Vector Graphics. As Mozilla mentioned, is essentially to graphic what HTML is to text.

So one of the library I was using to create those demo, I'm just going to show you, is called Snap.svg. It's just a tiny wrapper around SVG that lets you manipulate the elements from the JavaScript.

So here is an example of SVG that I wrote using Snap. So you can see here I can tweak the percentage value. So I can change that to 95 and so on.

And what it's doing here, you can see the animation looks pretty smooth. Actually, I just have one function that create this pie chart for a specific value of the percent. And what it's doing,

every time I click Run, it will just draw this element for 1%, 2%, and so on, until 95, and remove the other one, but just gives you the illusion that it's animated. So you can achieve, you know, it's pretty easy to achieve animated graphics, animation with that technique.

Here is another one. I was just changing the color and creating some kind of color charts.

All right. So here is what you can concretely achieve with SVG inside the viewer. So the first thing is the mark-up 2D. So this one-- by the way, anybody recognize this model? That's the Oblivion copter from the movie, Tom Cruise movie. Just fun. That's a cool model. Kind of fancy. I could grab it.

So this is the mark-up 2D feature. So basically, that's a part of the API that lets you draw 2D mark-ups on top of the viewer. So it's an API-only feature, so it means that you have to create yourself the UI to control that feature. And it gives you the function to draw the elements, but doesn't come with any specific UI. So you have to do it yourself.

So that's what it can do. I switched to Edit mode. I've got a set of mark-ups, of predefined mark-ups, that I can use. Let's set an arrow here. Another arrow there. The rectangle, and so on.

And so I can undo and redo my mark-up. Also, once they are created, I can move them around, rotate. I can resize them. So all this is just, like, out of the box, provided from the API. I just created this little dialog to control those mark-ups.

I can also save the mark-ups as inside layers, so basically when I do that, what it's doing is that it returns a state of the mark-ups to my application as a string. And then I can very easily store that in my database or somewhere in my application and restore that later.

So let's say I clear the mark-ups here. I'm going to create some other mark-up. Let's put a text here, layer two. Whoops. Layer two, and maybe [INAUDIBLE] the one the cloud. All right. Here it is. I save that as layer two.

And now I can switch back to View mode. So there was an Edit mode. I switch back to View mode, and I can restore my mark-up as layers, so I can overlay layers, you know, layer one, two. All right.

AUDIENCE: Is the camera position also stored?

PHILIPPE LEEFSMA: Yeah, so in that case, it's a bit rigid, because once you create mark-up for a specific view, then you cannot rotate the model around. You know, it stays. You can zoom in, zoom out, with your mark-ups.

But basically it's like a snapshot of the model. It's a bit limited. It's mostly used for 2D. You can use it in 3D, but that would be most useful in 2D.

This is another sample. So this one I wrote from scratch, myself, with SVG and JavaScript. So basically, I pick a specific point on the model. And here you can select the type of label that you want to see here.

I want to show materials. I select this one. Here I want to show, for example, the appearance, and so on. So I create a set of mark-ups here, and so it's like 2D graphics overlay on top of the viewer.

But the cool thing is that I'm using camera events. So here I get the position of the point in 3D where I click. You know, I'll show you some function how you can do that. And whenever I move the model, then I will use the camera event to say, oh, this 3D point has moved, and then I can update accordingly the 2D point.

So you can see, when you move the model, then it's kind of sticky. It produce some kind of cool effects. I can explode the model, and the mark-ups are also following those specific element.

Then I was adding another feature that I call occlusion. So here, I can enable or disable that. And this one produces the effect of hiding the mark-ups, if the clicked point goes behind the-- is not visible directly to the user. So if the occlusion is enabled, then when those points go out of the view, then automatically they got removed.

All right. Yeah, that's pretty much it for that sample.

OK, so this is how you can achieve that. It's not so complicated. So you have basically two methods.

The first one is to convert from 2D to 3D, so from a screen point to a world point. So basically, when I click with my mouse on the model, I want to find out what is the 3D point in my model. So I can access the viewer navigation, get screen point. I normalize that point, and then I'm using this method from the viewer API, you know, to get the hit point, so if I hit something on

the model, then this will return me a 3D point mapped on the surface of the model.

And then whenever I move the camera, then I just go the other way around. I go from the 3D point, which is basically it doesn't change on the model, but then now my screen point is going to change. So I will convert back the 3D point to know where is the new position of the mark-up on the screen.

So this is a sheet that's following. I pass the 3D point, you're getting the current camera, then you apply the metric's inverse transform, basically, of the model. Then you also apply the projects and metrics. And then you call this method view port to client that gives you a screen point. And you snap to just the center of that screen point.

So now you get the 2D point, and you can upload your 2D mark-ups on the screen and give the illusion that the point, that the mark-up is sticking to the model.

All right. So those were many 2D overlays. Now we can take a look at 3D overlays.

So a good example of 3D overlays-- so what I mean by 3D overlays, you will understand in a second, is embedding 3D custom shapes directly inside your model. So it's not like just a 2D which is overlay on the model. It's really like-- it's a custom shape inside the model that doesn't come initially from the geometry of the design, but that can enhance the visualization experience of your model.

So a pretty good example of that is the simulation hub. And then, by the way, they have a booth at AU, so you can go and check them out. They pulled out those examples.

So what they doing here is a fluid simulation. So basically, you upload your design. They will run some analyses, some simulation, on their backend. And they're able to display the result inside the viewer.

And so what is cool here is that you can really leverage the feature of the viewer, for example, the section planes here. And while you can see here the simulation of the fluid inside your component. And this is custom meshes embedded, directly added, to the scene. You can also animate those things. So a pretty nice result.

It's pretty cool. I like this app.

So here is another demo that I created to show you how flexible is this thing. Basically, that a

small particle simulation, so you can see, this is based completely on the viewer. So here, what it's doing, I just have a piece of JavaScript that updates my particles based on those magnetic fields.

So I can increase the number of particles. I can select that component. And so basically, what I'm doing here is I just create custom meshes. I'm adding custom meshes to the scene, and then the position of each mesh is computed by the simulation.

So I can change the force of those emitters. And so basically, it's another way to somehow add information to the model. So you can see-- so one of the drawback of this approach is that the frame rate gets pretty low as I increase the number of particle into the scene. You can see the frame rate drops. So let me shut it down. So it's pretty straightforward. You know, you just create Three.js geometry. Again, we are based on Three.js, so you can really leverage all the existing shapes and feature that you have in Three.js.

So I create a simple sphere. I create a mesh based on the geometry and I set the position. And I'm adding the mesh to the scene.

So this is what-- in that case, it's getting pretty slow, because when I start to have many meshes, the simulation takes-- the rendering of all those particles takes a lot of time. So there is another feature in Three.js that is called the point cloud. In that case, you can represent a large amount of points by using a single mesh, a single object. So the rendering is much faster.

So unfortunately, we don't have it yet in the viewer, because we don't support the latest version of Three.js. But we're going to have it pretty soon.

So for this reason, I just wrote another example which is based here-- that's not the viewer. It's completely based on Three.js. But pretty much, I could grab this code. As soon as we support point cloud in the viewer, I'll be able to grab this code and use the same thing in the viewer.

So in that case, I'm using the same particle engine. So the position of my particles are computed using the exact same code, but just the rendering is much faster, and I can handle a much larger amount of particles. I can see here-- you can see the frame rate. It remains pretty good. I have way more particles than before.

Just to give you an idea about what you can achieve. So it's pretty flexible. You're not limited by only loading the initial CAD data. You can really embed custom data, embed point cloud,

and really bend that things to pretty much whatever you need.

All right. Here's another example of overlaying information inside the viewer. So so far, I show you overlaying 2D graphics on top of the viewer, embedding 3D forms directly inside the view. Now it's a bit of a mix between the two. So it's called the CSS renderer and CSS object.

So this is what this is doing. So basically, this is embedding 2D graphics inside the 3D scene. So I show you what this is about. So here we've got a 3D model, but then what we did, that's a model of Pier 9, you know, at Autodesk. And what we have is that we have a set of sensor which are hooked up to those CNC machine in Pier 9.

And then they are sending back the data to our server. We are grabbing those data for each machine, and then we are displaying the result here inside the viewer. So this is directly hooked up to the cloud and to the result live of these sensors.

So basically, here, that would be pretty hard to create in 3D, some kind of graphics like that to embed inside the scene. But here, this is just a 2D webpage, so 2D graphics overlaid inside in 3D, you know, inside the 3D scenes. So it just gives you some of the options to overlay custom data, overlay custom information.

So here, if I select one of the machines, again, this is bringing a 2D page that contains documentation for this machine. So you could even imagine you are in VR inside this model. And then you can simply select the machine, and it brings up the product description, the webpage of the machine. And you can even send some orders to your manufacturer or to the provider of that machine directly inside the VR or here, directly inside the model without leaving even the model.

All right. So when I saw that, what I had previously, that's more just to play around. So I had this 3D model of a TV. So I said, OK, that would be funny to overlay some of these things, some 2D over that TV screen, just to show the result.

So this is what I did here. I hooked up some code to the selection of this button on the remote. So whenever I click one of the button here, I can display the same thing on my TV screen. So in that case, it's not hooked up to a server or to some IOT sensor. This is just displaying random data.

But you can see it's pretty flexible. So I can switch the channel here. I'm loading a webpage.

So this one is-- that's my own site. And I could even log in and start using the website. This is live.

So I saw that, OK, you can embed a 2D website. How about you try to embed a 3D web application inside this, just for fun? So this is what I did here. When I click on the third button, what it's doing is it's just going to load the viewer inside the viewer.

You know, it looks pretty good. It's completely like-- has completely the full set of feature, not even [INAUDIBLE]. I can access the properties here, select my components. And it just doesn't make any sense, but I had to try that. Just really fun.

So I'm going to-- I don't know performance-wise how good it works, but just-- so it's pretty straightforward. You just use those CSS renderer, CSS 3D object. So here, you create a CSS renderer. You append this dom element, or the CSS renderer to the viewer container.

So it's like as simple as appending a dom element to the viewer div. You create a parallel scene, and then here you can add any dom elements you want. So in that case, an i-frame. You create a CSS 3D object using that i-frame, and then you can start setting the position and the scale of your CSS object in 3D. It would just display these 2D graphics inside the 3D, and then the rest is computed automatically by the CSS object.

Now, when you rotate the scene and zoom in, zoom out, then you don't have to do any programming to enter that. So it just gives you just some ideas what you can achieve, how you can overlay more information in the viewer.

All right. So connecting your application to the cloud. Here, I don't want to show exhaustive things that you can do with the cloud. I just wanted to show you a set of samples and show it's pretty straightforward to expose some more data from your server to the client application.

So in most of my sample nowadays, I use node.js. So I know that ASP.NET might be more popular through the programmers, but I just wanted to show you this quick code sample that shows it's pretty easy to expose some feature from your server to the client application using node.js and Express.

So here, I'm simply using the Express router. I will create a route. So item and ID, so basically I want to be able to grab some items from my client application based on specific ID, pull that from the database.

So here, I've got this get item from database. Basically, you have to implement that based on which database you use. Is it MongoDB? Is it SAP? SQL database? That's highly dependent on what you use on the database sides. But basically, once you wrote this function, then the rest is-- you could reuse that.

And so here we are exposing this router as an API. So from my client application, then it's pretty easy to call that function and grab the data from my server, pull up off my database. So the standard way to do that is using the XML HTTP request. But a new feature that comes with the latest version of the browsers is called fetch. So not all the browsers have support for it, but you can find policies or libraries that can help you use the same code.

So basically, you will fetch this specific URL. So let's say you want to get an item by ID. So you create that wrapper method around. You pass the ID of the item you want to access. Then you fetch the specific URL, and it returns a callback function that contains the response.

And the response basically is a stream. So you can stream that to a JSON object, and in that case, call the on-success method to return the response of your items to the client.

So here, I'm going to show you how you can leverage that using Promise and async and await features. So again, those are like latest features of JavaScript. So if you wrote an async function to get the item, so instead of having what we have here based on callbacks, if you're writing an async function, you can simply-- because this fetch method also return a promise. It doesn't return directly the value. It returns an object that will return the result later on.

So you can very easily use that syntax. You're awaiting the result of the fetch, and basically what is doing this line is that it will lets you write the code, asynchronous code, in a synchronous way. So basically, when your code hit this line, you basically send a web service, web request, that is waiting for the response from your server or from a third-party web service.

And the code is just going to pause here until the response come back. But the rest of the code in your website is still going to continue to execute. It's a non-blocking call that lets you write the code in a synchronous way.

So basically, here we are waiting the result of the fetch, getting the response, and here we are awaiting the result of the-- the conversion of the response to JSON, and here we are returning the item. And we can write a function like that that will run a specific task on all the items.

So let's say we're passing an array of IDs, and then we want to run a specific task on all those items, but asynchronously. We don't want to call one item, process the result, call the second item. We want to do this in parallel. You know, send the request and not wait for each response to come back from the server, but still being able to run that in the non-blocking way.

So it's a pretty standard construct that you can use with the async and await. Basically, what it's doing is that you will map each ID of the item using the map function. That's pretty standard JavaScript. So basically, it takes each ID of your item, and it returns a get item. It returns-- whoops. Sorry.

It returns this function. So here, when you say return get item, it doesn't return the result of the get item. It returns a promise. So it really returns an object. And in the meantime, by the time you call the first item, second item, and so on, maybe the first item is processed, or not. You don't really have to worry about that.

And here, when you call this line, await, promise all item task. What it's doing is that it will asynchronously wait that all the item task have been processed. And because I'm awaiting that here, it will return an array of responses.

So with just the simple code, what I get here is an array of items that contain results for each of my request, sent asynchronously, but I'm writing that in a synchronous way. So if I had to write that with callbacks, then that would be pretty complex, because I would have to make sure that the first callback is done before the second, and so on. So that would be, like, really messy to enter that without using this async and await things.

So here are another set of samples that I created, so highly based on those asynchronous programming. So this one, I call that Forge RCDB, so that's an integration of a Mongo database and the viewer. So what you can see here are items which are stored in a database.

So it contains just a list of materials with the price, the suppliers, the currency of each item. And you can see it computes that pie chart based on the price of each item. So if I change the price of the plastic here, then you can see that the pie chart updates. Also, if I change the currency of each item, that will update the pie chart. Clicking an item is connected to the viewer. So pretty similar to what I was showing previously.

So basically, this sample is a summarize of all what I show you previously, but in a more realistic way of creating a web application. I've also got the mark-up 3D here, so I can create

my mark-ups here. I was hiding some fancy tool tip, again, like tiny animation entered by SVG.

And so one of the thing I added, one of the cool feature I added, is that those mark-ups are saved in the database in some custom states. So I've got a set of predefined mark-ups here that I can very easily restored. So all this information is stored in my custom database.

And here, that's the same feature than before. So yeah, mapping the colors to the components. The difference is that in that case, this dialogue is responsive. So I can resize this. I've got my bar chart. But that's pretty much the same sample I was showing previously.

So if you're looking for real-world application, applied concept of all those various code snippet, then you can check that code sample, Forge RCDB. All right. And just to show you a couple of other demos I found interesting, this one was put out by Dotty. I don't have it here. Let's go down.

So this is a demo that also connects the viewer and the model to custom data, to an external database. So basically, they are dealing with those oil companies, and you can see here they are using a similar feature of mapping the 2D mark-ups to the 3D.

But it's a bit more advanced, because you can toggle those various items on and off. You can click on specific items here, and it pulls out the item properties. And this is connected directly to a live database, so you may have an operator on the field, in the boat, who is checking what is the problem on your boat, and can go to a specific item, a specific location. And he has his tablet, he click there and can see what he has to do, how he has to fix the problem, or what has to be done there.

And you can hook it up, it's hooked up to states as well. So you can go through the various locations here and so on, so you get the idea. You know, it's really leveraging the various features that are there out of the box in the viewer.

And maybe the last demo I wanted to show you is that one. So it's called Project Dasher. So this is implemented by the R&D team at Autodesk.

So basically, what they have is a bunch of sensors which are hooked up to one of our office here in-- I think this is in Canada, in Toronto. And building navigation, so I can go fourth floor-- let me-- whoops. It's a bit heavy for the browser. I need to close some of the tabs.

That's the last demo. It's a bit too much to ask. Let me-- I'm not sure. Maybe I can show it up

from there. Don't ask me to continue. Nope. Let's check. Come on. It's going to load at some point.

All right. So surface shading. Adding the texture.

So what they have is they hooked up the data sent by some sensors, temperature sensors, and they are mapping based on those values of the temperature. They are mapping some-- they're using some custom shaders to map some colors on the ground and show the various temperature shifts. So here it's just replaying some set of data. But yeah, you can imagine that you have this live and change the material on your model as the sensors send back the data from a server.

All right. So here is the list of resources-- this presentation, a bunch of links. So yeah, here, think about filling up the survey about the class. Take a minute to fill up the survey. It's important for us.

And I hope you appreciate the class. So if you have more questions, if you have questions, then feel free to show up at the Forge booth, and also I'm happy to answer your question right now. All right. Thanks.

[APPLAUSE]