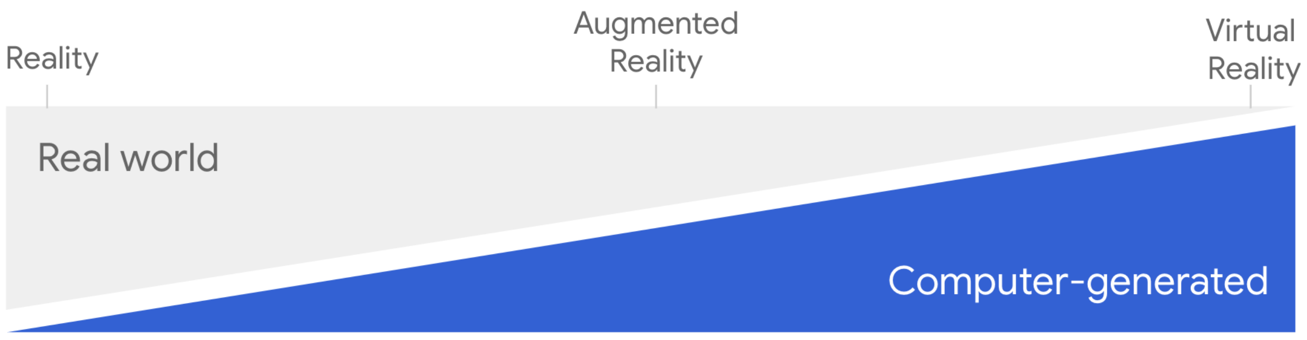
Welcome to the immersive web

By Joseph Medley
Technical Writer

The immersive web means virtual world experiences hosted through the browser. This covers entire virtual reality (VR) experiences surfaced in the browser or in VR enabled headsets like Google's Daydream, Oculus Rift, Samsung Gear VR, HTC Vive, and Windows Mixed Reality Headsets, as well as augmented reality experiences developed for AR-enabled mobile devices.

Welcome to the immersive web.

Though we use two terms to describe immersive experiences, they should be thought of as a spectrum from complete reality to a completely immersive VR environment, with various levels of AR in between.



The immersive web is a spectrum from complete reality to completely immersive, with various levels in between.

Examples of immersive experiences include:

- Immersive 360° videos
- Traditional 2D (or 3D) videos presented in immersive surroundings
- Data visualizations
- Home shopping
- Art
- Something cool nobody's thought of yet

How do I get there?

The immersive web has been available for nearly a year now in embryonic form. This was done through the WebVR 1.1 API, which has been available in an origin trial since Chrome 62. That API is also supported by Firefox and Edge as well as a polyfill for Safari.

But it's time to move on.

The origin trial is ending on July 24, 2018, and the spec has been superseded by the WebXR Device API and a new origin trial.

Note: If you're participating in the WebVR origin trial, you need a separate registration for the WebXR Origin Trial (explainer, sign-up form).

What happened to WebVR 1.1?

We learned a lot from WebVR 1.1, but over time, it became clear that some major changes were needed to support the types of applications developers want to build. The full list of lessons learned is too long to go into here, but includes issues like the API being explicitly tied to the main JavaScript thread, too many opportunities for developers to set up obviously wrong configurations, and common uses like magic window being a side effect rather than an intentional feature. (Magic window is a technique for viewing immersive content without a headset wherein the app renders a single view based on the device's orientation sensor.)

The new design facilitates simpler implementations and large performance improvements. At the same time, AR and other use cases were emerging and it became important that the API be extensible to support those in the future.

The WebXR Device API was designed and named with these expanded use cases in mind and provides a better path forward. Implementors of WebVR have committed to migrating to the WebXR Device API.

What is the WebXR Device API?

Like the WebVR spec before it, the WebXR Device API is a product of the Immersive Web Community Group which has contributors from Google, Microsoft, Mozilla, and others. The 'X in XR is intended as a sort of algebraic variable that stands for anything in the spectrum of immersive experiences. It's available in the previously mentioned origin trial as well as through a polyfill.

Note: As of Chrome 67 only VR capabilities are enabled. AR capabilities are expected in Chrome 69 so I hope to tell you about them soon.

There's more to this new API than I can go to in an article like this. I want to give you enough to start making sense of the WebXR samples. You can find more information in both the original explainer and our Immersive Web Early Adopters Guide. I'll be expanding the latter as the origin trial progresses. Feel free to open issues or submit pull requests. For this article, I'm going to discuss starting, stopping and running an XR session, plus a few basics about processing input.

What I'm not going to cover is how to draw AR/VR content to the screen. The WebXR Device API does not provide image rendering features. That's up to you. Drawing is done using WebGL APIs. You can do that if you're really ambitious. Though, we recommend using a framework. The immersive web samples use one created just for the demos called Cottontail. Three.js has supported WebXR since May. I've heard nothing about A-Frame.

Starting and running an app

The basic process is this:

1. Request an XR device.
2. If it's available, request an XR session. If you want the user to put their phone in a headset, it's called an immersive session and requires a user gesture to enter.
3. Use the session to run a render loop which provides 60 image frames per second. Draw appropriate content to the screen in each frame.
4. Run the render loop until the user decides to exit.
5. End the XR session.

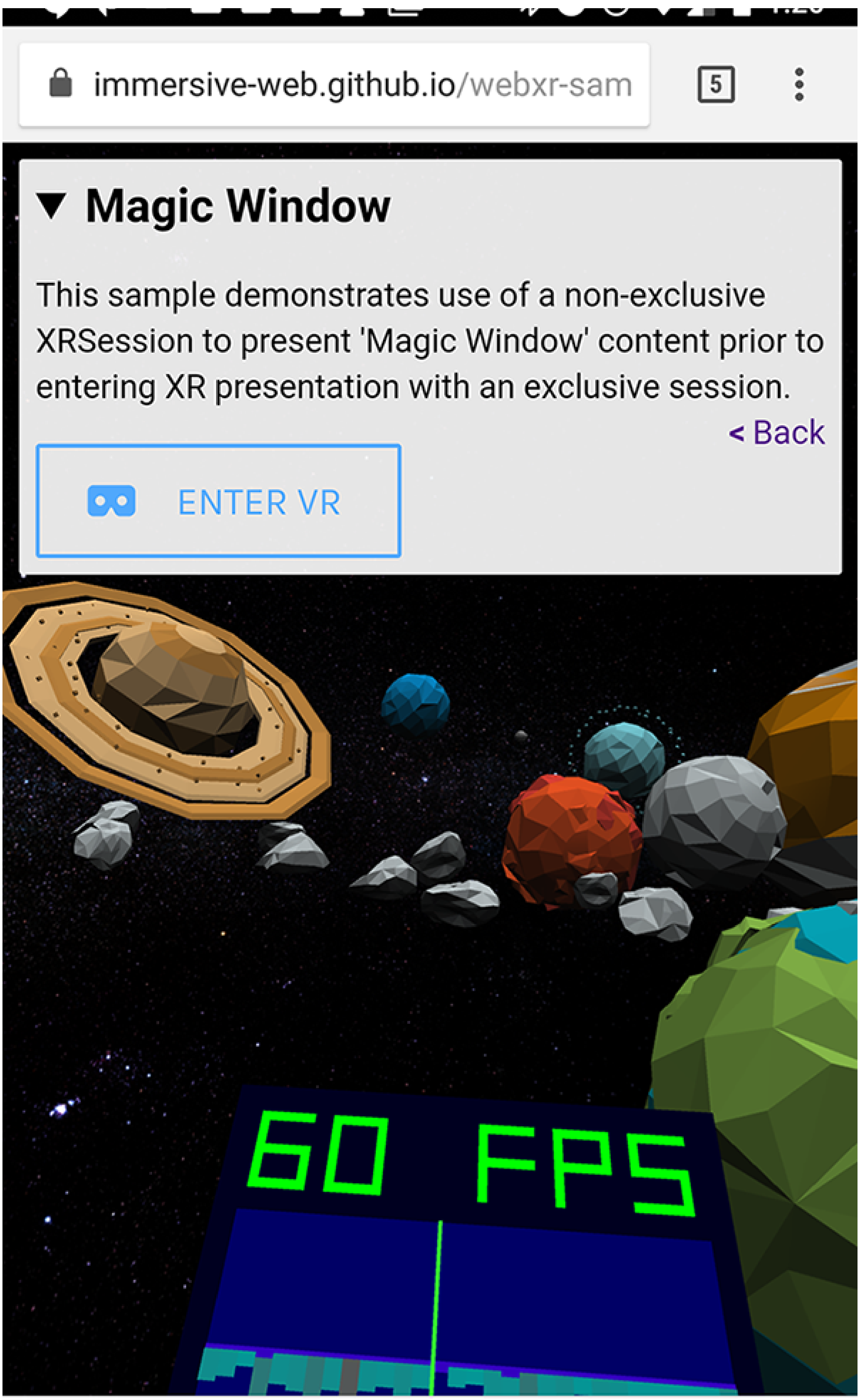
Let's look at this in a little more detail and include some code. You won't be able to run an app from what I'm about to show you. But again, this is just to give a sense of it.

Request an XR device

Here, you'll recognize the standard feature detection code. You could wrap this in a function called something like `checkForXR()`.

If you're not using an immersive session you can skip advertising the functionality and getting a user gesture and go straight to requesting a session. An immersive session is one that requires a headset. A non-immersive session simply shows content on the device screen. The former is what most people think of when you refer to virtual reality or augmented reality. The latter is sometimes called a 'magic window'.

```
if (navigator.xr) {
  navigator.xr.requestDevice()
  .then(xrDevice => {
    // Advertise the AR/VR functionality to get a user gesture.
  })
  .catch(err => {
    if (err.name === 'NotFoundError') {
      // No XRDevices available.
      console.error('No XR devices available:', err);
    } else {
      // An error occurred while requesting an XRDevice.
      console.error('Requesting XR device failed:', err);
    }
  })
} else{
  console.log("This browser does not support the WebXR API.");
}
```

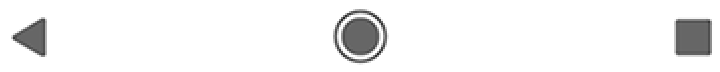

immersive-web.github.io/webxr-sam

▼ Magic Window

This sample demonstrates use of a non-exclusive XRSession to present 'Magic Window' content prior to entering XR presentation with an exclusive session.

< Back

ENTER VR



A user gesture in a magic window.

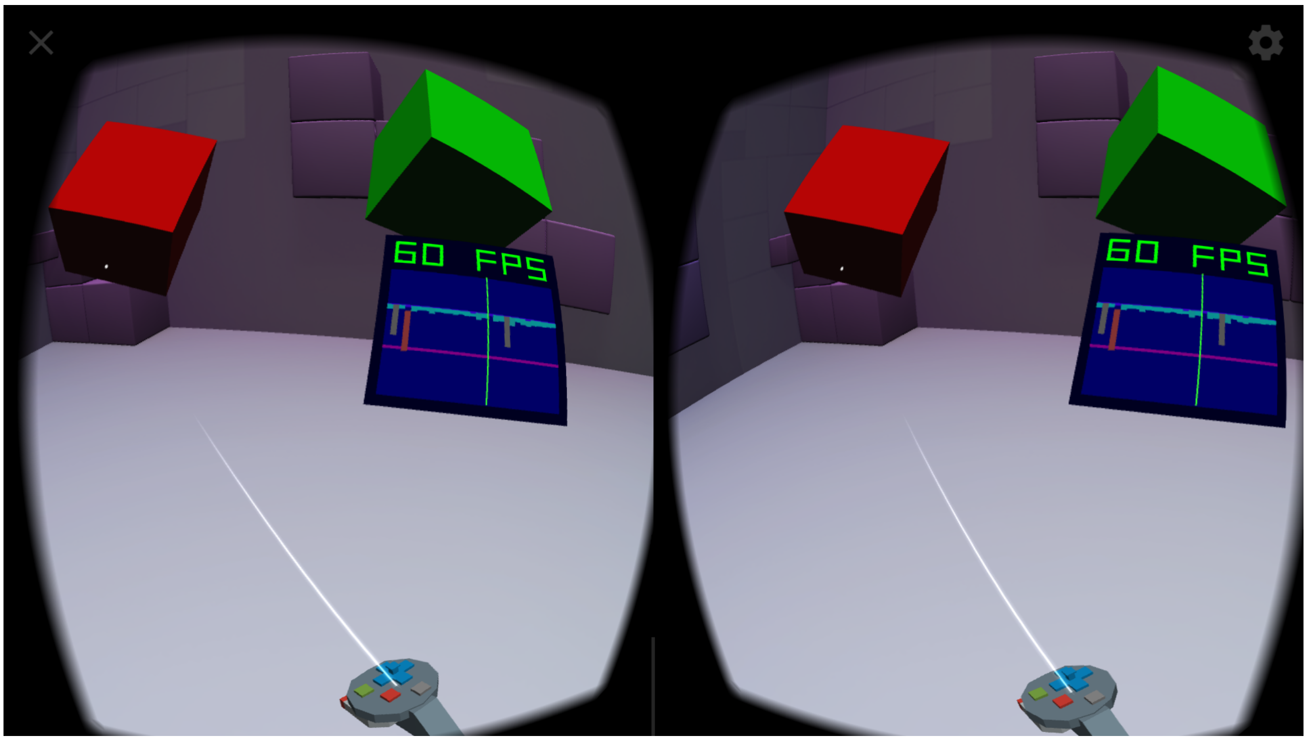
Request an XR session

Now that we have our device and our user gesture, it's time to get a session. To create a session, the browser needs a canvas on which to draw.

```
xrPresentationContext = htmlCanvasElement.getContext('xrpresent');
let sessionOptions = {
  // The immersive option is optional for non-immersive sessions; the value
  //   defaults to false.
  immersive: false,
  outputContext: xrPresentationContext
}
xrDevice.requestSession(sessionOptions)
.then(xrSession => {
  // Use a WebGL context as a base layer.
  xrSession.baseLayer = new XRWebGLLayer(session, gl);
  // Start the render loop
})
```

Run the render loop

The code for this step takes a bit of untangling. To untangle it, I'm about to throw a bunch of words at you. If you want a peek at the final code, jump ahead to have a quick look then come back for the full explanation. There's quite a bit that you may not be able to infer.



An immersive image as rendered to each eye.

The basic process for a render loop is this:

1. Request an animation frame.
2. Query for the position of the device.
3. Draw content to the position of the device based on it's position.
4. Do work needed for the input devices.
5. Repeat 60 times a second until the user decides to quit.

Request a presentation frame

The word 'frame' has several meanings in a Web XR context. The first is the *frame of reference* which defines where the origin of the coordinate system is calculated from, and what happens to that origin when the device moves. (Does the view stay the same when the user moves or does it shift as it would in real life?)

The second type of frame is the *presentation frame*, represented by an `XRFrame` object. This object contains the information needed to render a single frame of an AR/VR scene to the device. This is a bit confusing because a presentation frame is retrieved by calling `requestAnimationFrame()`. This makes it compatible with `window.requestAnimationFrame()`.

Before I give you any more to digest, I'll offer some code. The sample below shows how the render loop is started and maintained. Notice the dual use of the word frame. And notice the recursive call to **`requestAnimationFrame()`**. This function will be called 60 times a second.

```
xrSession.requestFrameOfReference('eye-level')
.then(xrFrameOfRef => {
  xrSession.requestAnimationFrame(onFrame(time, xrFrame) {
    // The time argument is for future use and not implemented at this time.
    // Process the frame.
    xrFrame.session.requestAnimationFrame(onFrame);
  }
});
```

Poses

Before drawing anything to the screen, you need to know where the display device is pointing and you need access to the screen. In general, the position and orientation of a thing in AR/VR is called a pose. Both viewers and input devices have a pose. (I cover input devices later.) Both viewer and input device poses are defined as a 4 by 4 matrix stored in a **`Float32Array`** in column major order. You get the viewer's pose by calling **`XRFrame.getDevicePose()`** on the current animation frame object. Always test to see if you got a pose back. If something went wrong you don't want to draw to the screen.

```
let pose = xrFrame.getDevicePose(xrFrameOfRef);
if (pose) {
  // Draw something to the screen.
}
```

Views

After checking the pose, it's time to draw something. The object you draw to is called a view (**`XRView`**). This is where the session type becomes important. Views are retrieved from the **`XRFrame`** object as an array. If you're in a non-immersive session the array has one view. If you're in an immersive session, the array has two, one for each eye.

```
for (let view of xrFrame.views) {
  // Draw something to the screen.
}
```

This is an important difference between WebXR and other immersive systems. Though it may seem pointless to iterate through one view, doing so allows you to have a single rendering path for a variety of devices.

The whole render loop

If I put all this together, I get the code below. I've left a placeholder for the input devices, which I'll cover in a later section.

```
xrSession.requestFrameOfReference('eye-level')
.then(xrFrameOfRef => {
  xrSession.requestAnimationFrame(onFrame(time, xrFrame) {
    // The time argument is for future use and not implemented at this time.
    let pose = xrFrame.getDevicePose(xrFrameOfRef);
    if (pose) {
      for (let view of xrFrame.views) {
        // Draw something to the screen.
      }
    }
    // Input device code will go here.
    frame.session.requestAnimationFrame(onFrame);
  }
}
```

End the XR session

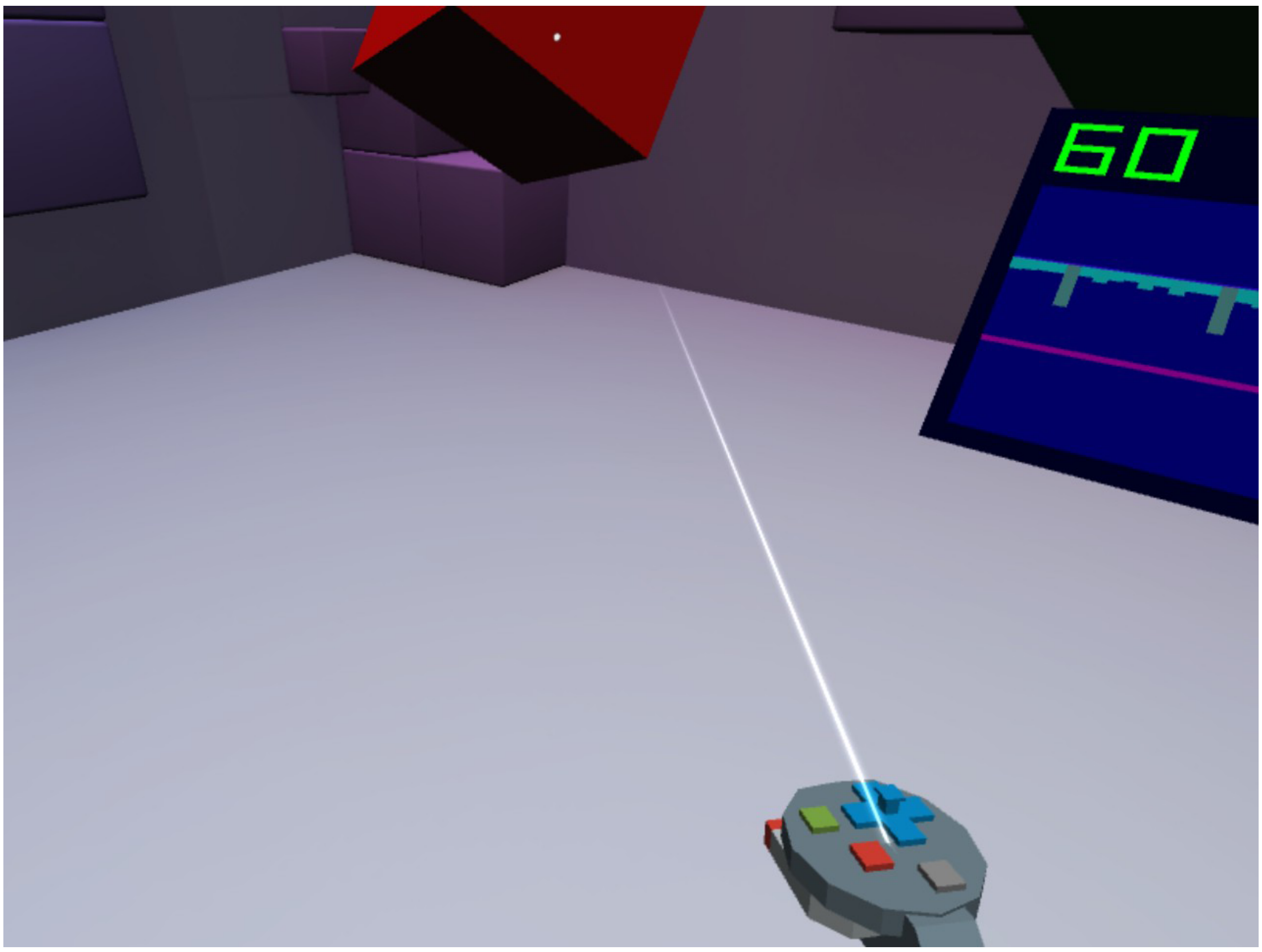
An XR session may end for several reasons, including ending by your own code through a call to **`XRSession.end()`**. Other causes include the headset being disconnected or another application taking control of it. This is why a well-behaved application should monitor the end event and when it occurs, discard the session and renderer objects. An XR session once ended cannot be resumed.

```
xrDevice.requestSession(sessionOptions)
.then(xrSession => {
  // Create a WebGL layer and initialize the render loop.
  xrSession.addEventListener('end', onSessionEnd);
});

// Restore the page to normal after immersive access has been released.
function onSessionEnd() {
  xrSession = null;

  // Ending the session stops executing callbacks passed to the XRSession's
  // requestAnimationFrame(). To continue rendering, use the window's
  // requestAnimationFrame() function.
  window.requestAnimationFrame(onDrawFrame);
}
```

How does interaction work?

As with the application lifetime, I'm just going to give you a taste for how to interact with objects in AR or VR.

The WebXR Device API adopts a "point and click" approach to user input. With this approach every input source has a defined *pointer ray* to indicate where an input device is pointing and events to indicate when something was selected. Your app draws the pointer ray and shows where it's pointed. When the user clicks the input device, events are fired—`select`, `selectStart`, and `selectEnd`, specifically. Your app determines what was clicked and responds appropriately.



Selecting in VR.

The input device and the pointer ray

To users, the pointer ray is just a faint line between the controller and whatever they're pointing at. But your app has to draw it. That means getting the pose of the input device and drawing a line from its location to an object in AR/VR space. That process looks roughly like this:

```
let inputSources = xrSession.getInputSources();
for (let xrInputSource of inputSources) {
  let inputPose = frame.getInputPose(inputSource, xrFrameOfRef);
  if (!inputPose) {
    continue;
  }
  if (inputPose.gripMatrix) {
    // Render a virtual version of the input device
    //   at the correct position and orientation.
  }
  if (inputPose.pointerMatrix) {
    // Draw a ray from the gripMatrix to the pointerMatrix.
  }
}
```

This is a stripped down version of the Input Tracking sample from the Immersive Web Community Group. As with frame rendering, drawing the pointer ray and the device is up to you. As alluded to earlier, this code must be run as part of the render loop.

Selecting items in virtual space

Merely pointing at things in AR/VR is pretty useless. To do anything useful, users need the ability to select things. The WebXR Device API provides three events for responding to user interactions: `select`, `selectStart`, and `selectEnd`. They have a quirk I didn't expect: they only tell you that an input device was clicked. They don't tell you what item in the environment was clicked. Event handlers are added to the `XRSession` object and should be added as soon as its available.

```
xrDevice.requestSession(sessionOptions)
.then(xrSession => {
  // Create a WebGL layer and initialize the render loop.
  xrSession.addEventListener('selectstart', onSelectStart);
  xrSession.addEventListener('selectend', onSelectEnd);
  xrSession.addEventListener('select', onSelect);
});
```

This code is based on an Input Selection example , in case you want more context.

To figure out what was clicked you use a pose. (Are you surprised? I didn't think so.) The details of that are specific to your app or whatever framework you're using, and hence beyond the scope of this article. Cottontail's approach is in the Input Selection example.

```
function onSelect(ev) {
  let inputPose = ev.frame.getInputPose(ev.inputSource, xrFrameOfRef);
  if (!inputPose) {
    return;
  }
  if (inputPose.pointerMatrix) {
    // Figure out what was clicked and respond.
  }
}
```

Conclusion: looking ahead

As I said earlier, augmented reality is expected in Chrome 69 (Canary some time in June 2018). Nevertheless, I encourage you try what we've got so far. We need feedback to make it better. Follow it's progress by watching ChromeStatus.com for WebXR Hit Test. You can also follow WebXR Anchors which will improve pose tracking.