FREE eBook

LEARNING aframe

Free unaffiliated eBook created from
Stack Overflow contributors.

#aframe

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: aframe

It is an unofficial and free aframe ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official aframe.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with aframe

Remarks

This section provides an overview of what aframe is, and why a developer might want to use it.

It should also mention any large subjects within aframe, and link out to the related topics. Since the Documentation for aframe is new, you may need to create initial versions of those related topics.

Versions

A-Frame 0.x

Version	Release Date
0.6	2017-05-25
0.5	2017-02-10
0.4	2016-12-17
0.3	2016-08-18

Legacy Versions

Version	Release Date
0.2	2016-03-26
0.1	2015-12-17

Examples

Getting started

A-Frame can be developed from a plain HTML file without having to install anything! A great way to try out A-Frame to remix the starter example on Glitch, an online code editor that instantly hosts and deploys for free. Or create an `.html` file and include A-Frame in the `head`:

```
<html>
  <head>
    <script src="https://aframe.io/releases/0.5.0/aframe.min.js"></script>
```

```
  </head>
  <body>
    <a-scene>
      <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9"></a-box>
      <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E"></a-sphere>
      <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color="#FFC65D"></a-cylinder>
      <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#7BC8A4"></a-
plane>
      <a-sky color="#ECECEC"></a-sky>
    </a-scene>
  </body>
</html>
```

Include the JS Build

To include A-Frame into an HTML file, we drop a `script` tag pointing to the CDN build:

```
<head>
  <script src="https://aframe.io/releases/0.5.0/aframe.min.js"></script>
</head>
```

Install from npm

We can also install A-Frame through npm:

```
$ npm install aframe
```

Then we can bundle A-Frame into our application. For example, with Browserify or Webpack:

```
require('aframe');
```

If you use npm, you can use angle, a command line interface for A-Frame. angle can initialize a scene template with a single command:

```
npm install -g angle && angle initscene
```

Features

VR Made Simple

Just drop in a `script` tag and `a-scene`. A-Frame will handle 3D boilerplate, VR setup, and default controls. Nothing to install, no build steps.

Declarative HTML

HTML is easy to read, understand, and copy-and-paste. Being based on top of HTML, A-Frame is accessible to everyone: web developers, VR enthusiasts, artists, designers, educators, makers, kids.

Cross-Platform VR

Build VR applications for Vive, Rift, Daydream, GearVR, and Cardboard with support for all respective controllers. Don’t have a headset or controllers? No problem! A-Frame still works on standard desktop and smartphones.

Entity-Component Architecture

A-Frame is a powerful three.js framework, providing a declarative, composable, reusable entity-component structure.js. HTML is just the tip of the iceberg; developers have unlimited access to JavaScript, DOM APIs, three.js, WebVR, and WebGL.

Performance

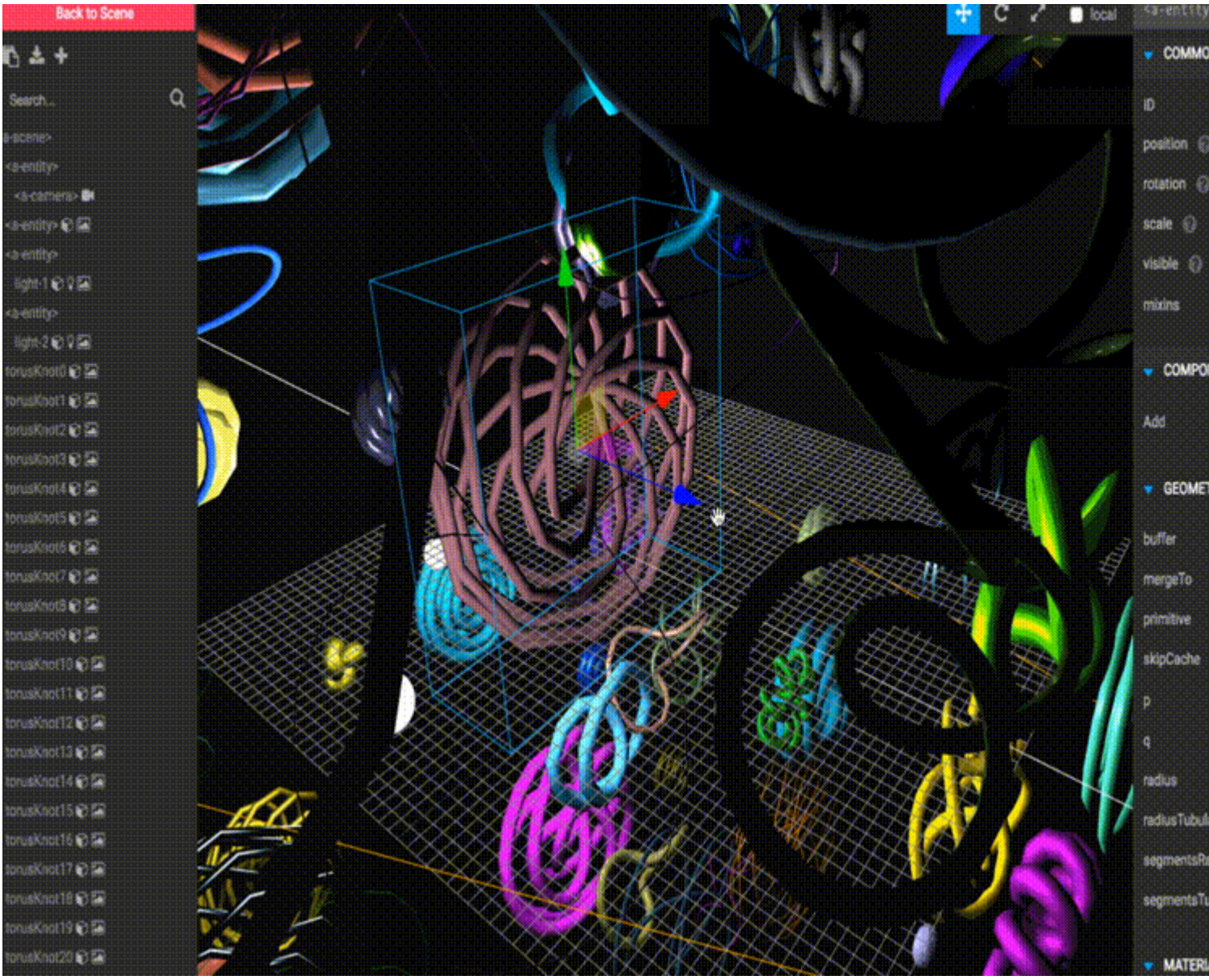
A-Frame is optimized from the ground up for WebVR. While A-Frame uses the DOM, its elements don’t touch the browser layout engine. 3D object updates are all done in memory with little overhead under a single requestAnimationFrame call. For reference, see A-Painter, a Tilt Brush clone built in A-Frame that runs like native (90+ FPS).

Tool Agnostic

Since the Web was built on the notion of the HTML, A-Frame is compatible with most libraries, frameworks, and tools including React, Preact, Vue.js, Angular, d3.js, Ember.js, jQuery.

Visual Inspector

A-Frame provides a handy built-in visual 3D inspector. Open up any A-Frame scene, hit `ctrl` + `alt` + `i`, and fly around to peek behind the hood!



Registry

Take powerful components that developers have published and plug them in straight from HTML. Similar to the Unity Asset Store, the A-Frame Registry collects and curates these components for easy discovery.

Components

Hit the ground running with A-Frame's core components such as geometries, materials, lights, animations, models, raycasters, shadows, positional audio, text, and Vive / Touch / Daydream / GearVR / Cardboard controls. Get even further with community components such as particle systems, physics, multiuser, oceans, mountains, speech recognition, motion capture, teleportation, super hands, and augmented reality.

Getting started for AR

To create AR applications on the web, you need to add a new library named AR.js. First you load A-frame followed by AR.js.

Newt you must setup you scene using the A-frames `a-scene`-tag with the `artoolkit`-attribute added. The `sourceType` must be your webcam. The font-camera of your smartphone is also supported using this.

the `a-marker-camera`-tag marks an image inside the recorded screen that represents an image. In this case it's `marker.png`. When the camera detects this marker the box will be displayed on the marker.

Below you could find the example code:

```
<script src="https://aframe.io/releases/0.5.0/aframe.min.js"></script>
<script src="https://rawgit.com/jeromeetienne/ar.js/master/aframe/build/aframe-
ar.js"></script>
<script>
    THREEx.ArToolkitContext.baseURL =
'https://rawgit.com/jeromeetienne/ar.js/master/three.js/'
</script>
<body>

    <a-scene artoolkit='sourceType: webcam;'>
        <a-box position='0 0 0.5' material='opacity: 0.5;'></a-box>
        <a-marker-camera preset='marker.png'></a-marker-camera>
    </a-scene>

</body>
```

Read Getting started with aframe online: https://riptutorial.com/aframe/topic/10017/getting-started-with-aframe

Chapter 2: Animation

Introduction

Animations and transitions in A-Frame are defined using the `<a-animation>` element as a child. The system is roughly based after the Web Animations specification. A-Frame uses tween.js internally.

Remarks

Attributes

Here is an overview of animation attributes. We'll go into more detail below.

Attribute	Description	Default Value
attribute	Attribute to animate. To specify a component attribute, use `componentName.property` syntax (e.g., `light.intensity`).	rotation
begin	Event name to wait on before beginning animation.	''
delay	Delay (in milliseconds) or event name to wait on before beginning animation.	0
direction	Direction of the animation (between `from` and `to`). One of `alternate`, `alternateReverse`, `normal`, `reverse`.	normal
dur	Duration in (milliseconds) of the animation.	1000
easing	Easing function of the animation. There are very many to choose from.	ease
end	Event name to wait on before stopping animation.	''
fill	Determines effect of animation when not actively in play. One of `backwards`, `both`, `forwards`, `none`.	forwards
from	Starting value.	Current value.
repeat	Repeat count or `indefinite`.	0
to	Ending value. Must be specified.	None

Begin

The `begin` attribute defines when the animation should start playing.

This can either be a *number*, representing milliseconds to wait, or an *event name* to wait for. For example, we can define an animation that waits 2 seconds before scaling an entity.

```
<a-entity>
  <a-animation attribute="scale" begin="2000" to="2 2 2"></a-animation>
</a-entity>
```

Or we can define an animation that waits for the parent element to trigger an event named `fade` before fading an entity.

```
<a-entity id="fading-cube" geometry="primitive: box" material="opacity: 1">
  <a-animation attribute="material.opacity" begin="fade" to="0"></a-animation>
</a-entity>
```

```
// Trigger an event to begin fading.
document.querySelector('#fading-cube').emit('fade');
```

Direction

The `direction` attribute defines which way to animate between the starting value and the final value.

When we define an alternating direction, the animation will go back and forth between the `from` and `to` values like a yo-yo. Alternating directions only take affect when we repeat the animation.

Value	Description
alternate	On even-numbered cycles, animate from `from` to `to`. On odd-numbered cycles, animation from `to` to `from`
alternate-reverse	On odd-numbered cycles, animate from `from` to `to`. On even-numbered cycles, animation from `to` to `from`
normal	Animate from `from` to `to`.
reverse	Animate from `to` to `from`.

Easing

The `easing` attribute defines the easing function of the animation, which defaults to `ease`. There are too many easing functions to list, but we can implicitly explain them.

One possible value is `linear`. And the basic easing functions are `ease`, `ease-in`, `ease-out`, and `ease-in-out`.

Then there are more groups of easing functions. The above basic easing functions prefix each

group of easing functions. The groups of easing functions are `cubic`, `quad`, `quart`, `quint`, `sine`, `expo`, `circ`, `elastic`, `back`, and `bounce`.

For example, the `cubic` group of easing functions would consist of `ease-cubic`, `ease-in-cubic`, `ease-out-cubic`, `ease-in-out-cubic`.

Fill

The `fill` attribute defines the effect of animation when not actively in play. Think of `fill` as what values the animation sets on the entity *before* and/or *after* each animation cycle. Below are the possible values for `fill` and their effects.

Value	Description
backwards	Before the animation starts, set the starting value to the `from` value.
both	Combine the effects of both backwards fill and forwards fill.
forwards	After the animation finishes, the final value will stay at the `to` value. The default fill.
none	Before the animation starts, set the starting value to the initial value. After the animation finishes, reset the value to the initial value.

Repeat

The `repeat` attribute defines how often the animation repeats. We call each repeat of the animation a cycle. Repeat can either be a number that counts down on each animation cycle until it reaches `0` at which point the animation will end, or we can set `repeat` to `indefinite` and the animation will loop continuously until the animation is manually removed or stopped.

EVENTS

The `<a-animation>` element emits a couple of events.

Event Name	Description
animationend	Emitted when the animation finishes. In case of repeats, emitted when the repeat count reaches `0`. Not emitted for indefinite repeats.
animationstart	Emitted immediately when the animation begins playing.

Examples

Example Animations

As an introductory example, to define a 5-meter orbit on an entity about the Y-axis that takes 10 seconds, we can offset the position and animate the rotation. This animation starts with the initial rotation about the Y-axis of 0 degrees, and goes around 360 degrees. It's defined with a duration of 10000 milliseconds, maintains the final value on each cycle of the animation, and loops infinitely.

```
<a-entity position="5 0 0" rotation="0 0 0">
  <a-animation attribute="rotation"
               to="0 360 0"
               dur="10000"
               fill="forwards"
               repeat="indefinite"></a-animation>
</a-entity>
```

Animating Different Types of Properties

A-Frame's animation system can animate different types of properties.

vec3 Properties

A-Frame has standard `vec3` components (i.e., `position`, `rotation`, and `scale`). These components consist of three factors: X, Y, and Z. We can pass three space-delimited numbers to the `from` and `to` attributes just as we would define them on an entity. In this case, the animation system will assume we are animating a `vec3` value.

For example, if we want to animate an entity going from one spot to another, we can animate the `position` component.

```
<a-entity>
  <a-animation attribute="position" from="1 1 1" to="2 4 -8"></a-animation>
</a-entity>
```

Boolean Properties

A-Frame has standard components that accept a single boolean value. Boolean values can be "animated" as well by flipping the boolean at the end of each animation cycle.

For example, we can define an animation that toggles off the visibility of an entity after 5 seconds.

```
<a-entity>
  <a-animation attribute="visible" dur="5000" to="false" repeat="indefinite"></a-animation>
</a-entity>
```

Numeric Properties

We can animate numeric attributes as well. For example, we can animate the intensity of the light primitive.

```
<a-light intensity="1">
  <a-animation attribute="intensity" to="3"></a-animation>
</a-light>
```

Color Properties

We can animate any component property that has a color type. For example, we can animate a box from white to red.

```
<a-entity id="blushing-cube" geometry="primitive: box">
  <a-animation attribute="material.color" from="white" to="red" dur="1000"></a-animation>
</a-entity>
```

Component Properties

We can animate a certain property of a multi-property component. To do so, we select the component property using the dot syntax: `componentName.propertyName`.

For example, to animate a cone's top radius, we can select the `radiusTop` value with `geometry.radiusTop`.

```
<a-entity geometry="primitive: cone; radiusTop: 1">
  <a-animation attribute="geometry.radiusTop" to="0.5"></a-animation>
</a-entity>
```

Read Animation online: https://riptutorial.com/aframe/topic/10071/animation---a-animation-

Chapter 3: Asset Management System

Introduction

A-Frame has an asset management system that allows us to place our assets in one place and to preload and cache assets for better performance.

Games and rich 3D experiences traditionally preload their assets, such as models or textures, before rendering their scenes. This makes sure that assets aren't missing visually, and this is beneficial for performance to ensure scenes don't try to fetch assets while rendering.

Remarks

Events

Since `<a-assets>` and `<a-asset-item>` are *nodes* in A-Frame, they will emit the `loaded` event when they say they have finished loading.

Event Name	Description
loaded	All assets were loaded, or assets timed out.
timeout	Assets timed out.

Load Progress on Individual Assets

`<a-asset-item>`

`<a-asset-item>` invokes the three.js FileLoader. We can use `<a-asset-item>` for any file type. When finished, it will set its `data` member with the text response.

Event Name	Description
error	Fetch error. Event detail contains `xhr` with `XMLHttpRequest` instance.
progress	Emitted on progress. Event detail contains `xhr` with `XMLHttpRequest` instance, `loadedBytes`, and `totalBytes`.
loaded	Asset pointed to by `src` was loaded.

`<img>`

Images are a standard DOM element so we can listen to the standard DOM events.

Event Name	Description
load	Image was loaded.

`HTMLMediaElement`

Audio and video assets are `HTMLMediaElement`s. The browser triggers particular events on these elements; noted here for convenience:

Event Name	Description
error	There was an error loading the asset.
loadeddata	Progress.
progress	Progress.

A-Frame uses these progress events, comparing how much time the browser buffered with the duration of the asset, to detect when the asset becomes loaded.

Examples

Example usage of assets

We place assets within `<a-assets>`, and we place `<a-assets>` within `<a-scene>`. Assets include:

- `<a-asset-item>` - Miscellaneous assets such as 3D models and materials
- `<audio>` - Sound files
- `<img>` - Image textures
- `<video>` - Video textures

The scene won't render or initialize until the browser fetches (or errors out) all the assets or the asset system reaches the timeout.

We can define our assets in `<a-assets>` and point to those assets from our entities using selectors:

```
<a-scene>
  <!-- Asset management system. -->
  <a-assets>
    <a-asset-item id="horse-obj" src="horse.obj"></a-asset-item>
    <a-asset-item id="horse-mtl" src="horse.mtl"></a-asset-item>
    <a-mixin id="giant" scale="5 5 5"></a-mixin>
    <audio id="neigh" src="neigh.mp3"></audio>
    <img id="advertisement" src="ad.png">
    <video id="kentucky-derby" src="derby.mp4"></video>
  </a-assets>

  <!-- Scene. -->
  <a-plane src="advertisement"></a-plane>
  <a-sound src="#neigh"></a-sound>
  <a-entity geometry="primitive: plane" material="src: #kentucky-derby"></a-entity>
```

```
  <a-entity mixin="giant" obj-model="obj: #horse-obj; mtl: #horse-mtl"></a-entity>
</a-scene>
```

The scene and its entities will wait for every asset (up until the timeout) before initializing and rendering.

Cross-Origin Resource Sharing (CORS)

Since A-Frame fetches assets using XHRs, browser security requires the browser to serve assets with cross-origin resource sharing (CORS) headers if the asset is on a different domain. Otherwise, we'd have to host assets on the same origin as the scene.

For some options, GitHub Pages serves everything with CORS headers. We recommend GitHub Pages as a simple deployment platform. Or you could also upload assets using the A-Frame + Uploadcare Uploader, a service that serves files with CORS headers set.

Given that CORS headers *are* set, `<a-assets>` will automatically set `crossorigin` attributes on media elements (e.g., `<audio>`, `<img>`, `<video>`) if it detects the resource is on a different domain.

Preloading Audio and Video

Audio and video assets will only block the scene if we set `autoplay` or if we set `preload="auto"`:

```
<a-scene>
  <a-assets>
    <!-- These will not block. -->
    <audio src="blockus.mp3"></audio>
    <video src="loadofblocks.mp4"></video>

    <!-- These will block. -->
    <audio src="blocky.mp3" autoplay></audio>
    <video src="blockiscooking.mp4" preload="auto"></video>
  </a-assets>
</a-scene>
```

Setting a Timeout

We can set a timeout that when reached, the scene will begin rendering and entities will begin initializing regardless of whether all the assets have loaded. The default timeout is 3 seconds. To set a different timeout, we just pass in the number of milliseconds to the `timeout` attribute:

If some assets are taking a long time to load, we may want to set an appropriate timeout such that the user isn't waiting all day in case their network is slow.

```
<a-scene>
  <a-assets timeout="10000">
    <!-- You got until the count of 10 to load else the show will go on without you. -->
    <img src="bigimage.png">
  </a-asset>
</a-scene>
```

Specifying Response Type

Content fetched by `<a-asset-item>` will be returned as plain text. If we want to use a different response type such as `arraybuffer`, use `<a-asset-item>`'s `response-type` attribute:

```
<a-asset-item response-type="arraybuffer" src="model.gltf"></a-asset-item>
```

How It Works Internally

Every element in A-Frame inherits from `<a-node>`, the `AFRAME.ANode` prototype. `ANode` controls load and initialization order. For an element to initialize (whether it be `<a-assets>`, `<a-asset-item>`, `<a-scene>`, or `<a-entity>`), its children must have already initialized. Nodes initialize bottom up.

`<a-assets>` is an `ANode`, and it waits for its children to load before it loads. And since `<a-assets>` is a child of `<a-scene>`, the scene effectively must wait for all assets to load. We also added extra load logic to `<a-entity>` such that they explicitly wait for `<a-assets>` to load if we have defined `<a-assets>`.

`<a-asset-item>` uses `THREE.FileLoader` to fetch files. three.js stores the returned data in `THREE.Cache`. Every three.js loader inherits from `THREE.FileLoader`, whether they are a `ColladaLoader`, `OBJLoader`, `ImageLoader`, etc. And they all have access and are aware of the central `THREE.Cache`. If A-Frame already fetched a file, A-Frame won't try to fetch it again.

Thus, since we block entity initialization on assets, by the time entities load, all assets will have been already fetched. As long as we define `<a-asset-item>`s, and the entity is fetching files using some form `THREE.FileLoader`, then caching will automatically work.

Accessing the `FileLoader` and Cache

To access the three.js `FileLoader` if we want to listen more closely:

```
console.log(document.querySelector('a-assets').fileLoader);
```

To access the cache that stores XHR responses:

```
console.log(THREE.Cache);
```

Read Asset Management System online: https://riptutorial.com/aframe/topic/10070/asset-management-system

Chapter 4: blend-model (component)

Introduction

blend-model component Loads a three.js format JSON model containing skeletal animation blending using **THREE.BlendCharacter**. This is mainly used to represent the hand and Vive controllers.

Syntax

- `<a-entity blend-model="#a-asset-item-selector"></a-entity>`

Remarks

VALUES

Type	Description
selector	Selector to an `<a-asset-item>`
string	`url()`-enclosed path to a JSON file

EVENTS

Event Name	Description
model-loaded	JSON model was loaded into the scene.

Examples

Example usage of `blend-model`

We can load the model by pointing using the ID to an that specifies the src to a file:

```
<a-scene>
  <a-assets>
    <!-- At first we load skeletal animation blending JSON as asset -->
    <a-asset-item id="hand" src="/path/to/hand.json"></a-asset-item>
  </a-assets>
  <!-- Now we can use that asset with blend-model-->
  <a-entity blend-model="#hand"></a-entity>
</a-scene>
```

Read blend-model (component) online: https://riptutorial.com/aframe/topic/10073/blend-model--component-

Chapter 5: Camera

Introduction

The camera component defines from which perspective the user views the scene. The camera is commonly paired with controls components that allow input devices to move and rotate the camera.

Syntax

- <a-entity camera></a-entity>
- <a-camera></a-camera>

Parameters

Property	Description
active	Whether the camera is the active camera in a scene with more than one camera.
far	Camera frustum far clipping plane.
fov	Field of view (in degrees).
near	Camera frustum near clipping plane.
userHeight	How much height to add to the camera when not in VR mode. The default camera has this set to 1.6 (meters, to represent average eye level.).
zoom	Zoom factor of the camera.

Remarks

When not in VR mode, `userHeight` translates the camera up to approximate average height of human eye level. The injected camera has this set to 1.6 (meters). When entering VR, this height offset is *removed* such that we used absolute position returned from the VR headset. The offset is convenient for experiences that work both in and out of VR, as well as making experiences look decent from a desktop screen as opposed to clipping the ground if the headset was resting on the ground.

When exiting VR, the camera will restore its rotation to its rotation before it entered VR. This is so when we exit VR, the rotation of the camera is back to normal for a desktop screen.

Examples

Default camera

A camera situated at the average height of human eye level (1.6 meters or 1.75 yard or 5.25 feet).

```
<a-entity camera="userHeight: 1.6" look-controls></a-entity>
```

Changing the Active Camera

When the active property gets toggled, the component will notify the camera system to change the current camera used by the renderer:

```
var secondCameraEl = document.querySelector('#second-camera');
secondCameraEl.setAttribute('camera', 'active', true);
```

Fixing Entities to the Camera

To fix entities onto the camera such that they stay within view no matter where the user looks, you can attach those entities as a child of the camera. Use cases might be a heads-up display (HUD).

```
<a-entity camera look-controls>
  <a-entity geometry="primitive: plane; height: 0.2; width: 0.2" position="0 0 -1"
            material="color: gray; opacity: 0.5"></a-entity>
</a-entity>
```

Note that you should use HUDs sparingly as they cause irritation and eye strain in VR. Consider integrating menus into the fabric of the world itself. If you do create a HUD, make sure that the HUD is more in the center of the field of view such that the user does not have to strain their eyes to read it.

a-camera primitive

The camera primitive determines what the user sees. We can change the viewport by modifying the camera entity’s position and rotation.

Note that by default, the camera origin will be at 0 1.6 0 in desktop mode and 0 0 0 in VR mode. Read about the camera.userHeight property.

```
<a-scene>
  <a-box></a-box>
  <a-camera></a-camera>
</a-scene>
```

Manually Positioning the Camera

To position the camera, set the position on a wrapper . Don’t set the position directly on the

camera primitive because controls will quickly override the set position:

```
<a-entity position="0 0 5">
  <a-camera></a-camera>
</a-entity>
```

Read Camera online: https://riptutorial.com/aframe/topic/10181/camera

Chapter 6: Components

Introduction

In the entity-component-system pattern, a component is a reusable and modular chunk of data that we plug into an entity to add appearance, behavior, and/or functionality.

In A-Frame, components modify entities which are 3D objects in the scene. We mix and compose components together to build complex objects. They let us encapsulate three.js and JavaScript code into modules that we can use declaratively from HTML. Components are roughly analogous to CSS.

Remarks

Definition Lifecycle Handler Methods

With the schema being the anatomy, the lifecycle methods are the physiology; the schema defines the shape of the data, the lifecycle handler methods *use* the data to modify the entity. The handlers will usually interact with the **Entity API**.

Overview of Methods

Method	Description
init	Called once when the component is initialized. Used to set up initial state and instantiate variables.
update	Called both when the component is initialized and whenever any of the component's properties is updated (e.g, via *setAttribute*). Used to modify the entity.
remove	Called when the component is removed from the entity (e.g., via *removeAttribute*) or when the entity is detached from the scene. Used to undo all previous modifications to the entity.
tick	Called on each render loop or tick of the scene. Used for continuous changes or checks.
play	Called whenever the scene or entity plays to add any background or dynamic behavior. Also called once when the component is initialized. Used to start or resume behavior.
pause	Called whenever the scene or entity pauses to remove any background or dynamic behavior. Also called when the component is removed from the entity or when the entity is detached from the scene. Used to pause behavior.

Method	Description
updateSchema	Called whenever any of the component's properties is updated. Can be used to dynamically modify the schema.

Component Prototype Properties

Within the methods, we have access to the component prototype via `this`:

Property	Description
this.data	Parsed component properties computed from the schema default values, mixins, and the entity's attributes.
this.el	Reference to the [entity][entity] as an HTML element.
this.el.sceneEl	Reference to the [scene][scene] as an HTML element.
this.id	If the component can have [multiple instances][multiple], the ID of the individual instance of the component (e.g., `foo` from `sound__foo`).

METHODS

.init ()

`.init ()` is called once at the beginning of the component's lifecycle. An entity can call the component's `init` handler:

- When the component is statically set on the entity in the HTML file and the page is loaded.
- When the component is set on an attached entity via `setAttribute`.
- When the component is set on an unattached entity, and the entity is then attached to the scene via `appendChild`.

The `init` handler is often used to:

- Set up initial state and variables
- Bind methods
- Attach event listeners

For example, a cursor component's `init` would set state variables, bind methods, and add event listeners:

```
AFRAME.registerComponent('cursor', {
  // ...
  init: function () {
    // Set up initial state and variables.
    this.intersection = null;
    // Bind methods.
```

```
    this.onIntersection = AFRAME.utils.bind(this.onIntersection, this);
    // Attach event listener.
    this.el.addEventListener('raycaster-intersection', this.onIntersection);
  }
  // ...
});
```

.update (oldData)

`.update (oldData)` is called whenever the component's properties change, including at the beginning of the component's lifecycle. An entity can call a component's `update` handler:

- After `init ()` is called, at the beginning of component's lifecycle.
- When the component's properties are updated with `.setAttribute`.

The `update` handler is often used to:

- Do most of the work in making modifications to the entity, using `this.data`.
- Modify the entity whenever one or more component properties change.

Granular modifications to the entity can be done by [diffing][diff] the current dataset (`this.data`) with the previous dataset before the update (`oldData`).

A-Frame calls `.update()` both at the beginning of a component's lifecycle and every time a component's data changes (e.g., as a result of `setAttribute`). The update handler often uses `this.data` to modify the entity. The update handler has access to the previous state of a component's data via its first argument. We can use the previous data of a component to tell exactly which properties changed to do granular updates.

For example, the **visible** component's `update` sets the visibility of the entity.

```
AFRAME.registerComponent('visible', {
  /**
   * this.el is the entity element.
   * this.el.object3D is the three.js object of the entity.
   * this.data is the component's property or properties.
   */
  update: function (oldData) {
    this.el.object3D.visible = this.data;
  }
  // ...
});
```

.remove ()

`.remove ()` is called whenever the component is detached from the entity. An entity can call a component's `remove` handler:

- When the component is removed from the entity via `removeAttribute`.
- When the entity is detached from the scene (e.g., `removeChild`).

The `remove` handler is often used to:

- Remove, undo, or clean up all of the component's modifications to the entity.
- Detach event listeners.

For example, when the [light component][light] is removed, the light component will remove the light object that it had previously set on the entity, thus removing it from the scene.

```
AFRAME.registerComponent('light', {
  // ...
  remove: function () {
    this.el.removeObject3D('light');
  }
  // ...
});
```

.tick (time, timeDelta)

`.tick ()` is called on each tick or frame of the scene's render loop. The scene will call a component's `tick` handler:

- On each frame of the render loop.
- On the order of 60 to 120 times per second.
- If the entity or scene is not paused (e.g., the Inspector is open).
- If the entity is still attached to the scene.

The `tick` handler is often used to:

- Continuously modify the entity on each frame or on an interval.
- Poll for conditions.

The `tick` handler is provided the global uptime of the scene in milliseconds (`time`) and the time difference in milliseconds since the last frame (`timeDelta`). These can be used for interpolation or to only run parts of the `tick` handler on a set interval.

For example, the **tracked controls component** will progress the controller's animations, update the controller's position and rotation, and check for button presses.

```
AFRAME.registerComponent('tracked-controls', {
  // ...
  tick: function (time, timeDelta) {
    this.updateMeshAnimation();
    this.updatePose();
    this.updateButtons();
  }
  // ...
});
```

.pause ()

`.pause ()` is called when the entity or scene pauses. The entity can call a component's `pause`

handler:

- Before the component is removed, before the `remove` handler is called.
- When the entity is paused with `Entity.pause ()`.
- When the scene is paused with `Scene.pause ()` (e.g., the Inspector is opened).

The `pause` handler is often used to:

- Remove event listeners.
- Remove any chances of dynamic behavior.

For example, the **sound component** will pause the sound and remove an event listener that would have played a sound on an event:

```
AFRAME.registerComponent('sound', {
  // ...
  pause: function () {
    this.pauseSound();
    this.removeEventListener();
  }
  // ...
});
```

.play ()

`.play ()` is called when the entity or scene resumes. The entity can call a component's `play` handler:

- When the component is first attached, after the `update` handler is called.
- When the entity was paused but then resumed with `Entity.play ()`.
- When the scene was paused but then resumed with `Scene.play ()`.

The `play` handler is often use to:

- Add event listeners.

For example, the **sound component** will play the sound and update the event listener that would play a sound on an event:

```
AFRAME.registerComponent('sound', {
  // ...
  play: function () {
    if (this.data.autoplay) { this.playSound(); }
    this.updateEventListener();
  }
  // ...
});
```

.updateSchema (data)

`.updateSchema ()`, if defined, is called on every update in order to check if the schema needs to be

dynamically modified.

The `updateSchema` handler is often used to:

- Dynamically update or extend the schema, usually depending on the value of a property.

For example, the **geometry component** checks if the `primitive` property changed to determine whether to update the schema for a different type of geometry:

```
AFRAME.registerComponent('geometry', {
  // ...
  updateSchema: (newData) {
    if (newData.primitive !== this.data.primitive) {
      this.extendSchema(GEOMETRIES[newData.primitive].schema);
    }
  }
  // ...
});
```

COMPONENT PROTOTYPE METHODS

.flushToDOM ()

To save on CPU time on stringification, A-Frame will only update in debug mode the component's serialized representation in the actual DOM. Calling flushToDOM () will manually serialize the component's data and update the DOM:

```
document.querySelector('[geometry]').components.geometry.flushToDOM();
```

Examples

Register a custom A-Frame component

AFRAME.registerComponent (name, definition)

Register an A-Frame component. We must register components before we use them anywhere in . Meaning from an HTML file, components should come in order before .

- **{string} name** - Component name. The component's public API as represented through an HTML attribute name.
- **{Object} definition** - Component definition. Contains schema and lifecycle handler methods.

Registering component in foo in your js file e.g foo-

component.js

```
AFRAME.registerComponent('foo', {
  schema: {},
  init: function () {},
  update: function () {},
  tick: function () {},
  remove: function () {},
  pause: function () {},
  play: function () {}
});
```

Usage of foo component in your scene

```
<html>
  <head>
    <script src="aframe.min.js"></script>
    <script src="foo-component.js"></script>
  </head>
  <body>
    <a-scene>
      <a-entity foo></a-entity>
    </a-scene>
  </body>
</html>
```

Component HTML Form

A component holds a bucket of data in the form of one or more component properties. Components use this data to modify entities. Consider an engine component, we might define properties such as horsepower or cylinders.

HTML attributes represent component names and the value of those attributes represent component data.

Single-Property Component

If a component is a single-property component, meaning its data consists of a single value, then in HTML, the component value looks like a normal HTML attribute:

```
<!-- `position` is the name of the position component. -->
<!-- `1 2 3` is the data of the position component. -->
<a-entity position="1 2 3"></a-entity>
```

Multi-Property Component

If a component is a multi-property component, meaning the data is consists of multiple properties and values, then in HTML, the component value resembles inline CSS styles:

```
<!-- `light` is the name of the light component. -->
<!-- The `type` property of the light is set to `point`. -->
<!-- The `color` property of the light is set to `crimson`. -->
<a-entity light="type: point; color: crimson"></a-entity>
```

Defining compnent schema object

The schema is an object that defines and describes the property or properties of the component. The schema's keys are the names of the property, and the schema's values define the types and values of the property (in case of a multi-property component):

Defining schema in your component

```
AFRAME.registerComponent('bar', {
  schema: {
    color: {default: '#FFF'},
    size: {type: 'int', default: 5}
  }
}
```

Override defined schema defaults

```
<a-scene>
  <a-entity bar="color: red; size: 20"></a-entity>
</a-scene>
```

Single-Property Schema

A component can either be a single-property component (consisting of one anonymous value) or a multi-property component (consisting of multiple named values). A-Frame will infer whether a component is single-property vs. multi-property based on the structure of the schema.

A single-property component's schema contains `type` and/or `default` keys, and the schema's values are plain values rather than objects:

```
AFRAME.registerComponent('foo', {
  schema: {type: 'int', default: 5}
});
```

```
<a-scene>
  <a-entity foo="20"></a-entity>
</a-scene>
```

A-Frame's component schema property types

Property types primarily define how the schema parses incoming data from the DOM for each property. The parsed data will then be available via the `data` property on the component's prototype. Below are A-Frame's built-in property types:

Property Type	Description	Default Value
array	Parses comma-separated values to array (i.e., `"1, 2, 3" to ['1', '2', '3']`).	[]
asset	For URLs pointing to general assets. Can parse URL out of a string in the form of `url(<url>)`. If the value is an element ID selector (e.g., `#texture`), this property type will call `getElementById` and `getAttribute('src')` to return a URL. The `asset` property type may or may not change to handle XHRs or return MediaElements directly (e.g., `<img>` elements).	''
audio	Same parsing as the `asset` property type. Will possibly be used by the A-Frame Inspector to present audio assets.	''
boolean	Parses string to boolean (i.e., `"false"` to false, everything else truthy).	false
color	Currently doesn't do any parsing. Primarily used by the A-Frame Inspector to present a color picker. Also, it is required to use color type for color animations to work.	#FFF
int	Calls `parseInt` (e.g., `"124.5"` to `124`).	0
map	Same parsing as the `asset` property type. Will possibly be used bt the A-Frame Inspector to present texture assets.	''
model	Same parsing as the `asset` property type. Will possibly be used bt the A-Frame Inspector to present model assets.	''
number	Calls `parseFloat` (e.g., `'124.5'` to `'124.5'`).	0
selector	Calls `querySelector` (e.g., `"#box"` to `<a-entity id="box">`).	null
selectorAll	Calls `querySelectorAll` and converts `NodeList` to `Array` (e.g., `".boxes"` to [<a-entity class="boxes", ...]),	null
string	Doesn't do any parsing.	''
vec2	Parses two numbers into an `{x, y}` object (e.g., `1 -2` to `{x: 1, y: -2}`.	{x: 0, y: 0}
vec3	Parses three numbers into an `{x, y, z}` object (e.g., `1 -2 3` to `{x: 1, y: -2, z: 3}`.	{x: 0, y: 0, z: 0}
vec4	Parses four numbers into an `{x, y, z, w}` object (e.g., `1 -2 3 -4.5` to `{x: 1, y: -2, z: 3, w: -4.5}`.	{x: 0, y: 0, z: 0, w: 0}

Property Type Inference

The schema will try to infer a property type given only a default value:

```
schema: {default: 10}  // type: "number"
schema: {default: "foo"}  // type: "string"
schema: {default: [1, 2, 3]}  // type: "array"
```

The schema will set a default value if not provided, given the property type:

```
schema: {type: 'number'}  // default: 0
schema: {type: 'string'}  // default: ''
schema: {type: 'vec3'}  // default: {x: 0, y: 0, z: 0}
```

Custom Property Type

We can also define our own property type or parser by providing a `parse` function in place of a `type`:

```
schema: {
  // Parse slash-delimited string to an array
  // (e.g., `foo="myProperty: a/b"` to `['a', 'b']`).
  myProperty: {
    default: [],
    parse: function (value) {
      return value.split('/');
    }
  }
}
```

Accessing a Component's Members and Methods

A component's members and methods can be accessed through the entity from the **.components** object. Look up the component from the entity's map of components, and we'll have access to the component's internals. Consider this example component:

```
AFRAME.registerComponent('foo', {
  init: function () {
    this.bar = 'baz';
  },
  qux: function () {
    // ...
  }
});
```

Let's access the **bar** member and **qux** method:

```
var fooComponent = document.querySelector('[foo]').components.foo;
console.log(fooComponent.bar);
fooComponent.qux();
```

Read Components online: https://riptutorial.com/aframe/topic/10068/components

Chapter 7: Controls (component)

Introduction

Controllers are vital for immersing people into a VR application. The potential of VR is not met without them, namely controllers that provide six degrees of freedom (6DoF). With controllers, people can reach out and around the scene and interact with objects with their hands.

A-Frame provides components for controllers across the spectrum as supported by their respective WebVR browsers through the Gamepad Web API. There are components for Vive, Oculus Touch, Daydream, and GearVR controllers.

Remarks

It's possible that you must enable gamepadextentions. You could do that using this steps:

- **On Chrome:** browse to `chrome://flags`
- **On Firefox:** browse to `about:config`
- **On IE:** Go to Group Policy Editor on your desktop
- **On Opera:** browse to `opera:config`
- **On Edge:** browse to `about:flags`

Examples

Wasd controls

The wasd-controls component controls an entity with the `W`, `A`, `S` and `D` or arrow keyboard keys. The `wasd-controls` component is commonly attached to an entity with the camera component.

```
<a-entity camera look-controls wasd-controls></a-entity>
```

For azerty keyboards, you could use `Z`, `Q`, `S` and `D` keys

Look controls

The look-controls component:

- Rotates the entity when we rotate a VR head-mounted display (HMD).
- Rotates the entity when we click-drag mouse.
- Rotates the entity when we touch-drag the touchscreen.

The look-controls component is usually used alongside the camera component.

```
<a-entity camera look-controls></a-entity>
```

Caveats

If you want to create your own component for look controls, you will have to copy and paste the HMD-tracking bits into your component. In the future, we may have a system for people to more easily create their controls.

Adding gaze to cursor

For this you need to add a cursor component to your camera

```
<a-scene>
  <a-camera>
    <a-cursor></a-cursor>
    <!-- Or <a-entity cursor></a-entity> -->
  </a-camera>
</a-scene>
```

More information you could find on the cursor (component) topic.

Hand controls

A-Frame 0.x0.3

A-Frame provides an implementation for supporting multiple types of 6DoF controllers (Vive, Oculus Touch) via the hand-controls component. The hand-controls component is primarily for 6DoF controllers since it's geared towards room scale interactions such as grabbing objects. The hand-controls component works on top of both Vive and Oculus Touch controllers by:

- Setting both the vive-controls and oculus-touch-controls component
- Overriding the controller models with a simple hand model
- Mapping Vive-specific and Oculus Touch-specific events to hand events and gestures (e.g., gripdown and triggerdown to thumbup)

To add the hand-controls component:

```
<a-entity hand-controls="left"></a-entity>
<a-entity hand-controls="right"></a-entity>
```

Unfortunately, there is not yet a 3DoF controller component that abstracts well all the types of 3DoF controllers (i.e., Daydream, GearVR). We could create a custom controller that works with both controllers. It would be fairly easy to cover since 3DoF controllers do not offer much potential for interaction (i.e., only rotational tracking with a touchpad).

The hand-controls gives tracked hands (using a prescribed model) with animated gestures. hand-controls wraps the vive-controls and oculus-touch-controls components, which in turn wrap the

tracked-controls component. The component gives extra events and handles hand animations and poses.

```
<a-entity hand-controls="left"></a-entity>
<a-entity hand-controls="right"></a-entity>
```

Tracked controls

A-Frame 0.x0.3

The tracked-controls component is A-Frame's base controller component that provides the foundation for all of A-Frame's controller components. The tracked-controls component:

- Grabs a Gamepad object from the Gamepad API given an ID or prefix.
- Applies pose (position and orientation) from the Gamepad API to read controller motion.
- Looks for changes in the Gamepad object's button values to provide events when buttons are pressed or touched and when axis and touchpads are changed (i.e. `axischanged`, `buttonchanged`, `buttondown`, `buttonup`, `touchstart`, `touchend`).

All of A-Frame's controller components build on top of the `tracked-controls` component by:

- Setting the tracked-controls component on the entity with the appropriate Gamepad ID (e.g., Oculus Touch (Right)). For example, the vive-controls component does

  ```
  el.setAttribute('tracked-controls', {idPrefix: 'OpenVR'})
  ```

  `tracked-controls` will then connect to the appropriate Gamepad object to provide pose and events for the entity.
- Abstracting the events provided by tracked-controls. tracked-controls events are low-level; it'd difficult for us to tell which buttons were pressed based off of those events alone because we'd have to know the button mappings beforehand. Controller components can know the mappings beforehand for their respective controllers and provide more semantic events such as `triggerdown` or `xbuttonup`.
- Providing a model. `tracked-controls` alone does not provide any appearance. Controller components can provide a model that shows visual feedback, gestures, and animations when buttons are pressed or touched. The controller components following are only activated if they detect the controller is found and seen as connected in the Gamepad API.

The `tracked-controls` component interfaces with tracked controllers. tracked-controls uses the Gamepad API to handle tracked controllers, and is abstracted by the hand-controls component as well as the vive-controls and oculus-touch-controls components. This component elects the appropriate controller, applies pose to the entity, observes buttons state and emits appropriate events.

Note that due to recent browser-specific changes, Vive controllers may be returned by the

Gamepad API with id values of either *"OpenVR Gamepad"* or *"OpenVR Controller"*, so using `idPrefix` for Vive / OpenVR controllers is recommended.

```
<a-entity tracked-controls="controller: 0; idPrefix: OpenVR"></a-entity>
```

3Dof and 6Dof controllers

Adding 3DoF Controllers

Controllers with 3 degrees of freedom (3DoF) are limited to rotational tracking. 3DoF controllers have no positional tracking meaning we can't reach out nor move our hand back-and-forth or up-and-down. Having a controller with only 3DoF is like having a hand and wrist without an arm. Read more about degrees of freedom for VR.

The 3DoF controller components provide rotational tracking, a default model matching the real-life hardware, and events to abstract the button mappings. The controllers for Google Daydream and Samsung GearVR have 3DoF, and both support only one controller for one hand.

A-Frame 0.x0.6

Daydream controllers

The daydream-controls component interfaces with the Google Daydream controllers. It wraps the tracked-controls component while adding button mappings, events, and a Daydream controller model that highlights the touched and/or pressed buttons (trackpad).

Match Daydream controller if present, regardless of hand.

```
<a-entity daydream-controls></a-entity>
```

Match Daydream controller if present and for specified hand.

```
<a-entity daydream-controls="hand: left"></a-entity>
<a-entity daydream-controls="hand: right"></a-entity>
```

GearVR-controllers

The gearvr-controls component interfaces with the Samsung/Oculus Gear VR controllers. It wraps the tracked-controls component while adding button mappings, events, and a Gear VR controller model that highlights the touched and/or pressed buttons (trackpad, trigger).

```
<!-- Match Gear VR controller if present, regardless of hand. -->
<a-entity gearvr-controls></a-entity>
<!-- Match Gear VR controller if present and for specified hand. -->
<a-entity gearvr-controls="hand: left"></a-entity>
<a-entity gearvr-controls="hand: right"></a-entity>
```

Adding 6DoF Controllers

Controllers with 6 degrees of freedom (6DoF) have both rotational and positional tracking. Unlike controllers with 3DoF which are constrained to orientation, controllers with 6DoF are able to move freely in 3D space. 6DoF allows us to reach forward, behind our backs, move our hands across our body or close to our face. Having 6DoF is like reality where we have both hands and arms. 6DoF also applies to the headset and additional trackers (e.g., feet, props). Having 6DoF is a minimum for providing a truly immersive VR experience.

The 6DoF controller components provide full tracking, a default model matching the real-life hardware, and events to abstract the button mappings. HTC Vive and Oculus Rift with Touch provide 6DoF and controllers for both hands. HTC Vive also provides trackers for tracking additional objects in the real world into VR.

A-Frame 0.x0.3

Vive controllers

The vive-controls component interfaces with the HTC Vive controllers/wands. It wraps the tracked-controls component while adding button mappings, events, and a Vive controller model that highlights the pressed buttons (trigger, grip, menu, system) and trackpad.

```
<a-entity vive-controls="hand: left"></a-entity>
<a-entity vive-controls="hand: right"></a-entity>
```

A-Frame 0.x0.5

Oculus touch controllers

The oculus-touch-controls component interfaces with the Oculus Touch controllers. It wraps the tracked-controls component while adding button mappings, events, and a Touch controller model.

```
<a-entity oculus-touch-controls="hand: left"></a-entity>
<a-entity oculus-touch-controls="hand: right"></a-entity>
```

Mouse control

Mouse controls are only supported outside the VR modus and could be use for games without a HMD. For more information about mouse controls, you could find in the mouse cursor example.

```
<a-scene>
  <a-entity camera look-controls mouse-cursor>
</a-scene>
```

Read Controls (component) online: https://riptutorial.com/aframe/topic/10112/controls--component-

Chapter 8: cursors

Introduction

The cursor component lets us interact with entities through clicking and gazing.

Syntax

- <a-entity cursor=""></a-cursor>
- <a-cursor></a-cursor>

Parameters

Property	Description
fuse	Whether cursor is fuse-based. Default value: `false` on desktop, `true` on mobile
fuseTimeout	How long to wait (in milliseconds) before triggering a fuse-based click event. Default value: 1500

Remarks

The cursor is a specific application of the raycaster component in that it

- Listens for mouse clicks and gaze-based fuses
- Captures only the first intersected entity
- Emits special mouse and hover events (e.g., relating to mouse down/up/enter/leave)
- Has more states for hovering.

When the mouse clicks, the closest visible entity intersecting the cursor, if any, will emit a click event. Note the cursor component only applies the raycasting behavior. To provide a shape or appearance to the cursor, you could apply the geometry and material components.

Events

Event	Description
click	Emitted on both cursor and intersected entity if a currently intersected entity is clicked (whether by mouse or by fuse).
fusing	Emitted on both cursor and intersected entity when fuse-based cursor starts counting down.

Event	Description
mousedown	Emitted on both cursor and intersected entity (if any) on mousedown on the canvas element.
mouseenter	Emitted on both cursor and intersected entity (if any) when cursor intersects with an entity.
mouseleave	Emitted on both cursor and intersected entity (if any) when cursor no longer intersects with previously intersected entity.
mouseup	Emitted on both cursor and intersected entity (if any) on mouseup on the canvas element.

Examples

Default cursor

For example, we can create a ring-shaped cursor fixed to the center of the screen. To fix the cursor to the screen so the cursor is always present no matter where we look, we place it as a child of the active camera entity. We pull it in front of the camera by placing it on the negative Z axis. When the cursor clicks on the box, we can listen to the click event.

```
<a-entity camera>
  <a-entity cursor="fuse: true; fuseTimeout: 500"
            position="0 0 -1"
            geometry="primitive: ring; radiusInner: 0.02; radiusOuter: 0.03"
            material="color: black; shader: flat">
  </a-entity>
</a-entity>

<a-entity id="box" cursor-listener geometry="primitive: box" material="color: blue">
</a-entity>
```

```
// Component to change to random color on click.
AFRAME.registerComponent('cursor-listener', {
  init: function () {
    var COLORS = ['red', 'green', 'blue'];
    this.el.addEventListener('click', function (evt) {
      var randomIndex = Math.floor(Math.random() * COLORS.length);
      this.setAttribute('material', 'color', COLORS[randomIndex]);
      console.log('I was clicked at: ', evt.detail.intersection.point);
    });
  }
});
```

Gaze-Based Interactions with cursor Component

We'll first go over gaze-based interactions. Gaze-based interactions rely on rotating our heads and looking at objects to interact with them. This type of interaction is for headsets without a controller. Even with a rotation-only controller (Daydream, GearVR), the interaction is still similar. Since A-

Frame provides mouse-drag controls by default, gaze-based can sort of be used on desktop to preview the interaction by dragging the camera rotation.

To add gaze-based interaction, we need to add or implement a component. A-Frame comes with a cursor component that provides gaze-based interaction if attached to the camera:

1. Explicitly define entity. Previously, A-Frame was providing the default camera.
2. Add `a-cursor` entity as a child element underneath the camera entity.
3. Optionally, configure the raycaster used by the cursor.

```
<a-scene>
  <a-camera>
    <a-cursor></a-cursor>
    <!-- Or <a-entity cursor></a-entity> -->
  </a-camera>
</a-scene>
```

a-cursor primitive

The cursor primitive is a reticle that allows for clicking and basic interactivity with a scene on devices that do not have a hand controller. The default appearance is a ring geometry. The cursor is usually placed as a child of the camera.

```
<a-scene>
  <a-camera>
    <a-cursor></a-cursor>
  </a-camera>
  <a-box></a-box>
</a-scene>
```

Fuse-Based Cursor

Also known as gaze-based cursor. If we set the cursor to be fuse-based, the cursor will trigger a click if the user gazes at an entity for a set amount of time. Imagine a laser strapped to the user's head, and the laser extends out into the scene. If the user stares at an entity long enough (i.e., the fuseTimeout), then the cursor will trigger a click.

The advantage of fuse-based interactions for VR is that it does not require extra input devices other than the headset. The fuse-based cursor is primarily intended for Google Cardboard applications. The disadvantage of fuse-based interactions is that it requires the user to turn their head a lot.

Configuring the Cursor through the Raycaster Component

The cursor builds on top of and depends on the raycaster component. If we want to customize the raycasting pieces of the cursor, we can do by changing the raycaster component properties. Say we want set a max distance, check for intersections less frequently, and set which objects are clickable:

```
<a-entity cursor raycaster="far: 20; interval: 1000; objects: .clickable"></a-entity>
```

Adding Visual Feedback

To add visual feedback to the cursor to show when the cursor is clicking or fusing, we can use the animation system. When the cursor intersects the entity, it will emit an event, and the animation system will pick up event with the begin attribute:

```
<a-entity cursor="fuse: true; fuseTimeout: 500"
          position="0 0 -1"
          geometry="primitive: ring"
          material="color: black; shader: flat">
  <a-animation begin="click" easing="ease-in" attribute="scale"
               fill="backwards" from="0.1 0.1 0.1" to="1 1 1"></a-animation>
  <a-animation begin="cursor-fusing" easing="ease-in" attribute="scale"
               fill="forwards" from="1 1 1" to="0.1 0.1 0.1"></a-animation>
</a-entity>
```

Mouse cursor

Note: For this example you need to add an external npm package.

If you want to use a mouse cursor of your computer, you need to add `aframe-mouse-cursor-component`. After if you must include the script using this code:

```
import 'aframe';
import 'aframe-mouse-cursor-component';

// or this

require('aframe');
require('aframe-mouse-cursor-component');
```

And on your camera you need to add the `mouse-cursor` component.

```
<a-scene>
  <a-entity camera look-controls mouse-cursor>
</a-scene>
```

Read cursors online: https://riptutorial.com/aframe/topic/10180/cursors

Chapter 9: Entities

Introduction

A-Frame represents an entity via the `<a-entity>` element. As defined in the entity-component-system pattern, entities are placeholder objects to which we plug in components to provide them appearance, behavior, and functionality.

Syntax

- `<a-entity> // Consider the entity below. By itself, it has no appearance, behavior, or functionality. It does nothing:`
- `<a-entity geometry="primitive: box" material="color: red"> // We can attach components to it to make it render something or do something. To give it shape and appearance, we can attach the geometry and material components:`
- `<a-entity geometry="primitive: box" material="color: red" light="type: point; intensity: 2.0"> // Or to make it emit light, we can further attach the light component:`

Parameters

Parameter	Details
components	`<a-entity>.components` is an object of components attached to the entity. This gives us access to the entity's components including each component's data, state, and methods.
isPlaying	Whether the entity is active and playing. If we pause the entity, then isPlaying becomes false.
object3D	`<a-entity>.object3D` is a reference to the entity's three.js Object3D representation. More specifically, object3D will be a `THREE.Group` object that may contain different types of `THREE.Object3D`s such as cameras, meshes, lights, or sounds:
object3DMap	An entity's object3DMap is an object that gives access to the different types of THREE.Object3Ds (e.g., camera, meshes, lights, sounds) that components have set.
sceneEl	An entity has a reference to its scene element.

Remarks

METHODS

addState (stateName)

addState will push a state onto the entity. This will emit the **stateadded** event, and we can check the state can for existence using **.is**:

```
entity.addEventListener('stateadded', function (evt) {
  if (evt.detail.state === 'selected') {
    console.log('Entity now selected!');
  }
});
entity.addState('selected');
entity.is('selected');  // >> true
```

---

emit (name, detail, bubbles)

emit emits a custom DOM event on the entity. For example, we can emit an event to trigger an animation:

```
<a-entity>
  <a-animation attribute="rotation" begin="rotate" to="0 360 0"></a-animation>
</a-entity>
```

```
entity.emit('rotate');
```

We can also pass event detail or data as the second argument:

```
entity.emit('collide', { target: collidingEntity });
```

The event will bubble by default. we can tell it not to bubble by passing false for bubble:

```
entity.emit('sink', null, false);
```

---

flushToDOM (recursive)

flushToDOM will manually serialize an entity's components' data and update the DOM.

---

getAttribute (componentName)

getAttribute retrieves parsed component data (including mixins and defaults).

```
// <a-entity geometry="primitive: box; width: 3">
entity.getAttribute('geometry');
// >> {primitive: "box", depth: 2, height: 2, translate: "0 0 0", width: 3, ...}
entity.getAttribute('geometry').primitive;
// >> "box"
entity.getAttribute('geometry').height;
// >> 2
entity.getAttribute('position');
// >> {x: 0, y: 0, z: 0}
```

If componentName is not the name of a registered component, getAttribute will behave as it normally would:

```
// <a-entity data-position="0 1 1">
entity.getAttribute('data-position');
// >> "0 1 1"
```

---

getDOMAttribute (componentName)

getDOMAttribute retrieves only parsed component data that is explicitly defined in the DOM or via **setAttribute**. If **componentName** is the name of a registered component, **getDOMAttribute** will return only the component data defined in the HTML as a parsed object. **getDOMAttribute** for components is the partial form of **getAttribute** since the returned component data does not include applied mixins or default values:

Compare the output of the above example of **getAttribute**:

```
// <a-entity geometry="primitive: box; width: 3">
entity.getDOMAttribute('geometry');
// >> { primitive: "box", width: 3 }
entity.getDOMAttribute('geometry').primitive;
// >> "box"
entity.getDOMAttribute('geometry').height;
// >> undefined
entity.getDOMAttribute('position');
// >> undefined
```

---

getObject3D (type)

getObject3D looks up a child **THREE.Object3D** referenced by **type** on **object3DMap**.

```
AFRAME.registerComponent('example-mesh', {
  init: function () {
     var el = this.el;
     el.getOrCreateObject3D('mesh', THREE.Mesh);
     el.getObject3D('mesh');  // Returns THREE.Mesh that was just created.
  }
});
```

---

getOrCreateObject3D (type, Constructor)

If the entity does not have a **THREE.Object3D** registered under **type**, **getOrCreateObject3D** will register an instantiated **THREE.Object3D** using the passed **Constructor**. If the entity does have an **THREE.Object3D** registered under **type**, **getOrCreateObject3D** will act as **getObject3D**:

```
AFRAME.registerComponent('example-geometry', {
  update: function () {
    var mesh = this.el.getOrCreateObject3D('mesh', THREE.Mesh);
    mesh.geometry = new THREE.Geometry();
  }
```

```
});
```

---

pause ()

pause() will stop any dynamic behavior as defined by animations and components. When we pause an entity, it will stop its animations and call **Component.pause()** on each of its components. The components decide to implement what happens on pause, which is often removing event listeners. An entity will call pause() on its child entities when we pause an entity.

```
// <a-entity id="spinning-jumping-ball">
entity.pause();
```

For example, the look-controls component on pause will remove event handlers that listen for input.

---

play ()

play() will start any dynamic behavior as defined by animations and components. This is automatically called when the DOM attaches an entity. When an entity **play()**, the entity calls **play()** on its child entities.

```
entity.pause();
entity.play();
```

For example, the sound component on play will begin playing the sound.

---

setAttribute (componentName, value, [propertyValue | clobber])

If **componentName** is not the name of a registered component or the component is a single-property component, **setAttribute** behaves as it normally would:

```
entity.setAttribute('visible', false);
```

Though if **componentName** is the name of a registered component, it may handle special parsing for the value. For example, the **position component** is a single-property component, but its property type parser allows it to take an object:

```
entity.setAttribute('position', { x: 1, y: 2, z: 3 });
```

---

setObject3D (type, obj)

setObject3D will register the passed **obj**, a **THREE.Object3D**, as **type** under the entity's **object3DMap**. A-Frame adds **obj** as a child of the entity's root **object3D**.

```
AFRAME.registerComponent('example-orthogonal-camera', {
```

```
  update: function () {
    this.el.setObject3D('camera', new THREE.OrthogonalCamera());
  }
});
```

---

removeAttribute (componentName, propertyName)

If **componentName** is the name of a registered component, along with removing the attribute from the DOM, **removeAttribute** will also detach the component from the entity, invoking the component's **remove** lifecycle method.

```
entity.removeAttribute('goemetry');  // Detach the geometry component.
entity.removeAttribute('sound');  // Detach the sound component.
```

If **propertyName** is given, **removeAttribute** will reset the property value of that property specified by **propertyName** to the property's default value:

```
entity.setAttribute('material', 'color', 'blue');  // The color is blue.
entity.removeAttribute('material', 'color');  // Reset the color to the default value, white.
```

---

removeObject3D (type)

removeObject3D removes the object specified by **type** from the entity's **THREE.Group** and thus from the scene. This will update the entity's **object3DMap**, setting the value of the **type** key to **null** . This is generally called from a component, often within the remove handler:

```
AFRAME.registerComponent('example-light', {
  update: function () {
    this.el.setObject3D('light', new THREE.Light());
    // Light is now part of the scene.
    // object3DMap.light is now a THREE.Light() object.
  },
  remove: function () {
    this.el.removeObject3D('light');
    // Light is now removed from the scene.
    // object3DMap.light is now null.
  }
});
```

---

removeState (stateName)

removeState will pop a state from the entity. This will emit the **stateremoved** event, and we can check the state its removal using **.is**:

```
entity.addEventListener('stateremoved', function (evt) {
  if (evt.detail.state === 'selected') {
    console.log('Entity no longer selected.');
  }
});
entity.addState('selected');
```

```
entity.is('selected');  // >> true
entity.removeState('selected');
entity.is('selected');  // >> false
```

EVENTS

Event Name	Description
child-attached	A child entity was attached to the entity.
child-detached	A child entity was detached from the entity.
componentchanged	One of the entity's components was modified.
componentinit	One of the entity's components was initialized.
componentremoved	One of the entity's components was removed.
loaded	The entity has attached and initialized its components.
object3dset	THREE.Object3D was set on entity using setObject3D(name). Event detail will contain name used to set on the object3DMap.
pause	The entity is now inactive and paused in terms of dynamic behavior.
play	The entity is now active and playing in terms of dynamic behavior.
stateadded	The entity received a new state.
stateremoved	The entity no longer has a certain state.
schemachanged	The schema of a component was changed.

EVENT DETAILS

Below is what the event detail contains for each event:

Event Name	Property	Description
child-attached	el	Reference to the attached child element.
componentchanged	name	Name of component that had its data modified.
	id	ID of component that had its data modified.
	newData	Component's new data, after it was modified.

Event Name	Property	Description
	oldData	Component’s previous data, before it was modified.
componentinitialized	name	Name of component that was initialized.
	id	ID of component that had its data modified.
	data	Component data.
componentremoved	name	Name of component that was removed.
	id	ID of component that was removed.
stateadded	state	The state that was attached (string).
stateremoved	state	The state that was detached (string).
schemachanged	component	Name of component that had it’s schema changed.

Examples

Listening for Component Changes

We can use the **componentchanged** event to listen for changes to the entity:

```
entity.addEventListener('componentchanged', function (evt) {
  if (evt.detail.name === 'position') {
    console.log('Entity has moved from',
      evt.detail.oldData, 'to', evt.detail.newData, '!');
  }
});
```

Listening for Child Elements Being Attached and Detached

We can use the **child-attached** and **child-detached** events to listen for when the scene attaches or detaches an entity:

```
entity.addEventListener('child-attached', function (evt) {
  if (evt.detail.el.tagName.toLowerCase() === 'a-box') {
    console.log('a box element has been attached');
  };
});
```

Entity Multi-Property Component Data (setAttribute)

Updating Multi-Property Component Data

To update component data for a multi-property component, we can pass the name of a registered

component as the **componentName**, and pass an object of properties as the **value**. A string is also acceptable (e.g., **type: spot; distance: 30**), but objects will save A-Frame some work in parsing:

```
// Only the properties passed in the object will be overwritten.
entity.setAttribute('light', {
  type: 'spot',
  distance: 30,
  intensity: 2.0
});
```

Or to update individual properties for a multi-property component, we can pass the name of registered component as the **componentName**, a property name as the second argument, and the property value to set as the third argument:

```
// All previous properties for the material component (besides the color)  will be unaffected.
entity.setAttribute('material', 'color', 'crimson');
```

Note that array property types behave uniquely:

- Arrays are mutable. They are assigned by reference so changes to arrays will be visible by the component.
- Updates to array type properties will not trigger the component's update method nor emit events.

Updating Multi-Property Component Data

If **true** is passed as the third argument to **.setAttribute**, then non-specified properties will be reset and clobbered:

```
// All previous properties for the light component will be removed and overwritten.
entity.setAttribute('light', {
  type: 'spot',
  distance: 30,
  intensity: 2.0
}, true);
```

Retrieving an Entity

We can simply retrieve an entity using DOM APIs.

```
<a-entity id="mario"></a-entity>
```

```
var el = document.querySelector('#mario');
```

Retrieving an Entity components

For example, if we wanted to grab an entity's three.js camera object or material object, we could

reach into its components

```
var camera = document.querySelector('a-entity[camera]').components.camera.camera;
var material = document.querySelector('a-entity[material]').components.material.material;
```

Or if a component exposes an API, we can call its methods:

```
document.querySelector('a-entity[sound]').components.sound.pause();
```

Read Entities online: https://riptutorial.com/aframe/topic/10066/entities--a-entity-

Chapter 10: gltf-model (component)

Introduction

The gltf-model component allows to use 3D models in the glTF format with A-Frame. glTF is a Khronos standard for efficient, full-scene 3D models and is optimised for usage on the web.

Syntax

- `<a-entity gltf-model="url(https://cdn.rawgit.com/KhronosGroup/glTF-Sample-Models/9176d098/1.0/Duck/glTF/Duck.gltf)"></a-entity>`
- `<a-entity gltf-model="#duck"></a-entity>`

Parameters

Parameter	Details
`url(...)`	will load the glTF model wrapped from the URL wrapped in `url()`
`#example`	will load the `<a-asset-item>` with the ID `example`

Examples

Loading a glTF model via URL

```
<a-scene>
  <a-entity gltf-model="url(https://cdn.rawgit.com/KhronosGroup/glTF-Sample-Models/9176d098/1.0/Duck/glTF/Duck.gltf)" position="0 0 -5"></a-entity>
</a-scene>
```

Loading a gltf-model via the asset system

```
<a-scene>
  <a-assets>
    <a-asset-item id="duck" src="https://cdn.rawgit.com/KhronosGroup/glTF-Sample-Models/9176d098/1.0/Duck/glTF/Duck.gltf"></a-asset-item>
  </a-assets>

  <a-entity gltf-model="#duck" position="0 0 -5"></a-entity>
</a-scene>
```

Read gltf-model (component) online: https://riptutorial.com/aframe/topic/10758/gltf-model--component-

Chapter 11: light (component)

Introduction

The light component defines the entity as a source of light. Light affects all materials that have not specified a flat shading model with shader: flat. Note that lights are computationally expensive we should limit number of lights in a scene.

Syntax

- <a-entity light="color: #AFA; intensity: 1.5" position="-1 1 0"></a-entity>
- <a-light type="point" color="blue" position="0 5 0"></a-light>

Parameters

Parameters	Details
type	One of ambient, directional, hemisphere, point, spot.
color	Light color.
intensity	Light strength.

Examples

Ambient

Ambient lights globally affect all entities in the scene. The color and intensity properties define ambient lights. Additionally, position, rotation, and scale have no effect on ambient lights.

We recommend to have some form of ambient light such that shadowed areas are not fully black and to mimic indirect lighting.

```
<a-entity light="type: ambient; color: #CCC"></a-entity>
```

Directional

Directional lights are like a light source that is infinitely far away, but shining from a specific direction, like the sun. Thus, absolute position do not have an effect on the intensity of the light on an entity. We can specify the direction using the position component.

The example below creates a light source shining from the upper-left at a 45-degree angle. Note that because only the vector matters, position="-100 100 0" and position="-1 1 0" are the same.

```
<a-entity light="type: directional; color: #EEE; intensity: 0.5" position="-1 1 0"></a-entity>
```

We can specify the direction of the directional light with its orientation by creating a child entity it targets. For example, pointing down its -Z axis:

```
<a-light type="directional" position="0 0 0" rotation="-90 0 0" target="#directionaltarget">
    <a-entity id="directionaltarget" position="0 0 -1"></a-entity>
</a-light>
```

Hemisphere

Hemisphere lights are like an ambient light, but with two different colors, one from above (color) and one from below (groundColor). This can be useful for scenes with two distinct lighting colors (e.g., a grassy field under a gray sky).

```
<a-entity light="type: hemisphere; color: #33C; groundColor: #3C3; intensity: 2"></a-entity>
```

Property	Description	Default Value
groundColor	Light color from below.	#fff

Point

Point lights, unlike directional lights, are omni-directional and affect materials depending on their position and distance. Point likes are like light bulb. The closer the light bulb gets to an object, the greater the object is lit.

```
<a-entity light="type: point; intensity: 0.75; distance: 50; decay: 2"
      position="0 10 10"></a-entity>
```

Property	Description	Default Value
decay	Amount the light dims along the distance of the light.	1.0
distance	Distance where intensity becomes 0. If distance is 0, then the point light does not decay with distance.	0.0

Spot

Spot lights are like point lights in the sense that they affect materials depending on its position and distance, but spot lights are not omni-directional. They mainly cast light in one direction, like the Bat-Signal.

```
<a-entity light="type: spot; angle: 45"></a-entity>
```

Property	Description	Default Value
angle	Maximum extent of spot light from its direction (in degrees).	60
decay	Amount the light dims along the distance of the light.	1.0
distance	Distance where intensity becomes 0. If distance is 0, then the point light does not decay with distance.	0.0
penumbra	Percent of the spotlight cone that is attenuated due to penumbra.	0.0
target	element the spot should point to. set to null to transform spotlight by orientation, pointing to it’s -Z axis.	null

Default lighting

By default, A-Frame scenes inject default lighting, an ambient light and a directional light. These default lights are visible in the DOM with the data-aframe-default-light attribute. Whenever we add any lights, A-Frame removes the default lights from the scene.

```
<!-- Default lighting injected by A-Frame. -->
<a-entity light="type: ambient; color: #BBB"></a-entity>
<a-entity light="type: directional; color: #FFF; intensity: 0.6" position="-0.5 1 1"></a-
entity>
```

Read light (component) online: https://riptutorial.com/aframe/topic/10078/light--component-

Chapter 12: Mixins

Introduction

Mixins provide a way to compose and reuse commonly-used sets of component properties. They are defined using the `<a-mixin>` element and are placed in `<a-assets>`. Mixins should be set with an id, and when an entity sets that id as its mixin attribute, the entity will absorb all of the mixin's attributes.

Examples

Example usage of mixins

```
<a-scene>
  <a-assets>
    <a-mixin id="red" material="color: red"></a-mixin>
    <a-mixin id="blue" material="color: blue"></a-mixin>
    <a-mixin id="cube" geometry="primitive: box"></a-mixin>
  </a-assets>
  <a-entity mixin="red cube"></a-entity>
  <a-entity mixin="blue cube"></a-entity>
</a-scene>
```

The entity with red cube will take the properties from the red mixin and the cube mixin in that order. Likewise with the blue cube. Conceptually, the entities above expand to:

```
<a-entity material="color: red" geometry="primitive: box"></a-entity>
<a-entity material="color: blue" geometry="primitive: box"></a-entity>
```

Merging Component Properties

Properties of a multi-property component will merge if defined by multiple mixins and/or the entity. For example:

```
<a-scene>
  <a-assets>
    <a-mixin id="box" geometry="primitive: box"></a-mixin>
    <a-mixin id="tall" geometry="height: 10"></a-mixin>
    <a-mixin id="wide" geometry="width: 10"></a-mixin>
  </a-assets>
  <a-entity mixin="wide tall box" geometry="depth: 2"></a-entity>
</a-scene>
```

All of the geometry component properties will merge since they are included as mixins and defined on the entity. The entity would then be equivalent to:

```
<a-entity geometry="primitive: box; height: 10; depth: 2; width: 10"></a-entity>
```

Order and Precedence

When an entity includes multiple mixins that define the same component properties, the right-most mixin takes precedence. In the example below, the entity includes both `red` and `blue` mixins, and since the `blue` mixin is included last, the final color of the cube will be blue.

```
<a-scene>
  <a-assets>
    <a-mixin id="red" material="color: red"></a-mixin>
    <a-mixin id="blue" material="color: blue"></a-mixin>
    <a-mixin id="cube" geometry="primitive: box"></a-mixin>
  </a-assets>

  <a-entity mixin="red blue cube"></a-entity>
</a-scene>
```

If an entity itself defines a property that is already defined by a mixin, the entity's definition takes precedence. In the example below, the entity includes both `red` and `blue` mixins and also defines a green color. Since the entity directly defines its own color, the final color of the cube will be green.

```
<a-scene>
  <a-assets>
    <a-mixin id="red" material="color: red"></a-mixin>
    <a-mixin id="blue" material="color: blue"></a-mixin>
    <a-mixin id="cube" geometry="primitive: box"></a-mixin>
  </a-assets>

  <a-entity mixin="red blue cube" material="color: green"></a-entity>
</a-scene>
```

Read Mixins online: https://riptutorial.com/aframe/topic/10072/mixins--a-mixin-

Chapter 13: Primitives

Introduction

Primitives are just `<a-entity>`s under the hood. This means primitives have the same API as entities such as positioning, rotating, scaling, and attaching components. A-Frame provides a handful of elements such as `<a-box>` or `<a-sky>` called primitives that wrap the entity-component pattern to make it appealing for beginners. . Developers can create their own primitives as well.

Remarks

Under the Hood

Primitives act as a convenience layer (i.e., syntactic sugar) primarily for newcomers. Keep in mind for now that primitives are `<a-entity>`s under the hood that:

- Have a semantic name (e.g., `<a-box>`)
- Have a preset bundle of components with default values
- Map or proxy HTML attributes to [component][component] data

Primitives are similar to prefabs in Unity. Some literature on the entity-component-system pattern refer to them as assemblages. They abstract the core entity-component API to:

- Pre-compose useful components together with prescribed defaults
- Act as a shorthand for complex-but-common types of entities (e.g., `<a-sky>`)
- Provide a familiar interface for beginners since A-Frame takes HTML in a new direction

Under the hood, this `<a-box>` primitive:

```
<a-box color="red" width="3"></a-box>
```

represents this entity-component form:

```
<a-entity geometry="primitive: box; width: 3" material="color: red"></a-entity>
```

`<a-box>` defaults the `geometry.primitive` property to `box`. And the primitive maps the HTML `width` attribute to the underlying `geometry.width` property as well as the HTML `color` attribute to the underlying `material.color` property.

Examples

Registering a Primitive

We can register our own primitives (i.e., register an element) using `AFRAME.registerPrimitive(name, definition)`. `definition` is a JavaScript object defining these properties:

Property	Description
defaultComponents	Object specifying default components of the primitive. The keys are the components' names and the values are the components' default data.
mappings	Object specifying mapping between HTML attribute name and component property names. Whenever the HTML attribute name is updated, the primitive will update the corresponding component property. The component property is defined using a dot syntax `${componentName}.${propertyName}`.

For example, below is A-Frame's registration for the `<a-box>` primitive:

```
var extendDeep = AFRAME.utils.extendDeep;

// The mesh mixin provides common material properties for creating mesh-based primitives.
// This makes the material component a default component and maps all the base material
properties.
var meshMixin = AFRAME.primitives.getMeshMixin();

AFRAME.registerPrimitive('a-box', extendDeep({}, meshMixin, {
  // Preset default components. These components and component properties will be attached to
the entity out-of-the-box.
  defaultComponents: {
    geometry: {primitive: 'box'}
  },

  // Defined mappings from HTML attributes to component properties (using dots as delimiters).
  // If we set `depth="5"` in HTML, then the primitive will automatically set
`geometry="depth: 5"`.
  mappings: {
    depth: 'geometry.depth',
    height: 'geometry.height',
    width: 'geometry.width'
  }
}));
```

Which we can use then

```
<a-box depth="1.5" height="1.5" width="1.5"></a-box>
```

represents this entity-component form:

```
<a-entity geometry="primitive: box; depth: 1.5; height: 1.5; width:1.5;"></a-entity>
```

Read Primitives online: https://riptutorial.com/aframe/topic/10074/primitives

Chapter 14: Raycasters (component)

Introduction

The raycaster component does general intersection testing with a raycaster. Raycasting is the method of extending a line from an origin towards a direction, and checking whether that line intersects with other entites. The raycaster component is a wrapper on top of the three.js raycaster. It checks for intersections at a certain interval against a list of objects, and will emit events on the entity when it detects intersections or clearing of intersections (i.e., when the raycaster is no longer

Parameters

Parameter	Details
far	Maximum distance under which resulting entities are returned. Cannot be lower then near.
interval	Number of milliseconds to wait in between each intersection test. Lower number is better for faster updates. Higher number is better for performance.
near	Minimum distance over which resuilting entities are returned. Cannot be lower than 0.
objects	Query selector to pick which objects to test for intersection. If not specified, all entities will be tested.
recursive	Checks all children of objects if set. Else only checks intersections with root objects.

Remarks

Events

Name	Details
`raycaster-intersected`	Emitted on the intersected entity. Entity is intersecting with a raycaster. Event detail will contain el, the raycasting entity, and intersection, an object containing detailed data about the intersection.
`raycaster-intersected-cleared`	Emitted on the intersected entity. Entity is no longer intersecting with a raycaster. Event detail will contain el, the raycasting entity.

Name	Details
`raycaster-intersection`	Emitted on the raycasting entity. Raycaster is intersecting with one or more entities. Event detail will contain els, an array with the intersected entities, and intersections, an array of objects containing detailed data about the intersections.
`raycaster-intersection-cleared`	Emitted on the raycasting entity. Raycaster is no longer intersecting with an entity. Event detail will contain el, the formerly intersected entity.

Member

Member	Description
`intersectedEls`	Entities currently intersecting the raycaster.
`objects`	three.js objects to test for intersections. Will be scene.children if not objects property is not specified.
`raycaster`	three.js raycaster object.

Methode

Method	Description
`refreshObjects`	Refreshes the list of objects based off of the objects property to test for intersection.

Examples

Setting the Origin and Direction of the Raycaster

The raycaster has an origin, where its ray starts, and a direction, where the ray goes.

The origin of the raycaster is at the raycaster entity's position. We can change the origin of the raycaster by setting the position component of the raycaster entity (or parent entities of the raycaster entity).

The direction of the raycaster is in "front" of the raycaster entity (i.e., 0 0 -1, on the negative Z-axis). We can change the direction of the raycaster by setting the rotation component of the raycaster entity (or parent entities of the raycaster entity).

For example, here is applying a raycaster along the length of a rotated bullet:

```
<!-- Bullet, rotated to be parallel with the ground. -->
<a-entity id="bullet" geometry="primitive: cylinder; height: 0.1" rotation="-90 0 0">
  <!-- Raycaster, targets enemies, made to be as long as the bullet, positioned to the start
of the bullet, rotated to align with the bullet. -->
  <a-entity raycaster="objects: .enemies; far: 0.1" position="0 -0.5 0" rotation="90 0 0"></a-
entity>
</a-entity>
```

Whitelisting Entities to Test for Intersection

We usually don't want to test everything in the scene for intersections (e.g., for collisions or for clicks). Selective intersections are good for performance to limit the number of entities to test for intersection since intersection testing is an operation that will run over 60 times per second.

To select or pick the entities we want to test for intersection, we can use the objects property. If this property is not defined, then the raycaster will test every object in the scene for intersection. objects takes a query selector value:

```
<a-entity raycaster="objects: .clickable" cursor></a-entity>
<a-entity class="clickable" geometry="primitive: box" position="1 0 0"></a-entity>
<a-entity class="not-clickable" geometry="primitive: sphere" position="-1 0 0"></a-entity>
```

Read Raycasters (component) online: https://riptutorial.com/aframe/topic/10036/raycasters--component-

Chapter 15: Scene

Introduction

A scene is represented by the `<a-scene>` element. The scene is the global root object, and all entities are contained within the scene.

The scene inherits from the Entity class so it inherits all of its properties, its methods, the ability to attach components, and the behavior to wait for all of its child nodes (e.g., `<a-assets>` and `<a-entity>`) to load before kicking off the render loop.

Parameters

Parameter	Details
behaviors	Array of components with tick methods that will be run on every frame.
camera	Active three.js camera.
canvas	Reference to the canvas element.
isMobile	Whether or not environment is detected to be mobile.
object3D	THREE.Scene object.
renderer	Active THREE.WebGLRenderer.
renderStarted	Whether scene is rendering.
effect	Renderer for VR created by passing active renderer into THREE.VREffect.
systems	Instantiated systems.
time	Global uptime of scene in seconds.

Remarks

METHODS

Name	Description
enterVR	Switch to stereo render and push content to the headset. Needs to be called within a user-generated event handler like `click`. the first time a page enters VR.

Name	Description
exitVR	Switch to mono renderer and stops presenting content on the headset.
reload	Revert the scene to its original state.

EVENTS

Name	Description
enter-vr	User has entered VR and headset started presenting content.
exit-vr	User has exited VR and headset stopped presenting content.
loaded	All nodes have loaded.
renderstart	Render loop has started.

Examples

Attaching Scene Components

Components can be attached to the scene as well as entities. A-Frame ships with a few components to configure the scene:

Component	Details
embedded	Remove fullscreen styles from the canvas.
fog	Add fog.
keyboard-shortcuts	Toggle keyboard shortcuts.
inspector	Inject the A-Frame Inspector.
stats	Toggle performance stats.
vr-mode-ui	Toggle UI for entering and exiting VR.
debug	Enables component-to-DOM serialization.

Using embedded scenes

If you want to use WebVR content mixed with HTML content, for example when you're making a extended showcase key content, you could use the `embedded` tag. When you're using this, it's possible to look around inside 360° content using the gyroscope of your smartphone or click and

drag on computer.

```
<script src="https://aframe.io/releases/0.5.0/aframe.min.js"></script>
<div class="vrcontent">
  <a-scene embedded>
    <a-assets>
      <img id="sky" src="https://c1.staticflickr.com/5/4248/34705881091_37b5cf2d28_o.jpg"
alt="" />
    </a-assets>

    <a-sky src="#sky"></a-sky>
  </a-scene>
</div>

<div class="overlay">
  <button class="calltoaction">Click me!</button>
</div>

<div class="content">
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Deleniti animi aliquid
architecto quibusdam ipsum, debitis dolor mollitia. Quidem, cumque quos porro doloribus iure
dolore illum, qui rem asperiores unde laboriosam.Dolorum tempora quam eveniet ea recusandae
deserunt, velit similique. Cum sunt rerum beatae officiis qui sed molestiae et ullam quasi,
harum maxime vel, aspernatur quidem molestias. Provident quae illo harum?Sunt expedita,
repellat saepe vel accusamus odio. Alias, obcaecati harum earum inventore asperiores quaerat,
sit autem nostrum. Sunt illo numquam, temporibus pariatur optio nam, expedita necessitatibus
aliquid nemo maxime nisi. Praesentium corporis, ea sunt asperiores, recusandae animi, rem
doloribus, possimus cum laudantium libero. Maiores a, iusto aspernatur reiciendis ratione sunt
nisi, rem, quasi temporibus ullam non. Neque repellat facilis illo.Quibusdam reiciendis sunt
tempora fuga deleniti, molestias temporibus doloremque. Nam sed consequatur consectetur ut
tempora a nesciunt, perspiciatis dolorem reprehenderit modi enim at veritatis, excepturi
voluptate quod, voluptatibus voluptas. Cum.Debitis, nesciunt, repellat voluptatem sapiente
incidunt quidem asperiores reprehenderit vero quisquam placeat sunt voluptatibus velit.
Consectetur atque voluptates, repellendus facere sequi ea totam quia quis non incidunt.
Soluta, aut, provident. Eos sequi itaque dolorem atque ex id maiores dolor eaque libero iste
deserunt ea voluptate minima cum laboriosam, qui animi, fuga suscipit necessitatibus vero,
autem blanditiis, totam nulla. Quo, et. Quisquam commodi voluptatum dolorem aspernatur,
distinctio et ullam laborum laboriosam quo nisi, praesentium quaerat ab excepturi. Illum harum
doloremque, accusantium, beatae culpa assumenda laboriosam, quos mollitia aperiam dolorem
praesentium minus!</p>
</div>
```

Debug

The debug component enables component-to-DOM serialization.

```
<a-scene debug></a-scene>
```

Component-to-DOM Serialization

By default, for performance reasons, A-Frame does not update the DOM with component data. If we open the browser's DOM inspector, we will see only the component names (and not the values) are visible.

```
<a-entity geometry material position rotation></a-entity>
```

A-Frame stores the component data in memory. Updating the DOM takes CPU time for converting internal component data to strings. If we want to see the DOM update for debugging purposes, we can attach the debug component to the scene. Components will check for an enabled debug component before trying to serialize to the DOM. Then we will be able to view component data in the DOM:

```
<a-entity geometry="primitive: box" material="color: red" position="1 2 3" rotation="0 180
0"></a-entity>
```

Make sure that this component is not active in production.

Manually Serializing to DOM

To manually serialize to DOM, use Entity.flushToDOM or Component.flushToDOM:

```
document.querySelector('a-entity').components.position.flushToDOM();  // Flush a component.
document.querySelector('a-entity').flushToDOM();  // Flush an entity.
document.querySelector('a-entity').flushToDOM(true);  // Flush an entity and its children.
document.querySelector('a-scene').flushToDOM(true);  // Flush every entity.
```

Running Content Scripts on the Scene

The recommended way is to write a component, and attach it to the scene element.
The scene and its children will be initialized before this component.

```
AFRAME.registerComponent('do-something', {
  init: function () {
    var sceneEl = this.el;
  }
});
```

```
<a-scene do-something></a-scene>
```

If for some particular reason you prefer not to write a dedicated component you need to wait for the scene to finish initializing and attaching:

```
var scene = document.querySelector('a-scene');

if (scene.hasLoaded) {
  run();
} else {
  scene.addEventListener('loaded', run);
}

function run () {
  var entity = scene.querySelector('a-entity');
  entity.setAttribute('material', 'color', 'red');
}
```

Read Scene online: https://riptutorial.com/aframe/topic/10069/scene--a-scene-

Chapter 16: System

Introduction

A system, of the entity-component-system pattern, provides global scope, services, and management to classes of components. It provides public APIs (methods and properties) for classes of components. A system can be accessed through the scene element, and can help components interface with the global scene.

For example, the camera system manages all entities with the camera component, controlling which camera is the active camera.

Parameters

Parameter	Details
data	Data provided by the schema available across handlers and methods
el	Reference to `<a-scene>`
schema	Behaves the same as component schemas. Parses to data.

Remarks

METHODS

A system, like a component, defines lifecycle handlers. It can also define methods intended to be public API.

Method	Description
init	Called once when the system is initialized. Used to initialize.
pause	Called when the scene pauses. Used to stop dynamic behavior.
play	Called when the scene starts or resumes. Used to start dynamic behavior.
tick	If defined, will be called on every tick of the scene's render loop.

Examples

Registering a System

A system is registered similarly to a A-Frame component.

If the system name matches a component name, then the component will have a reference to the system as this.system:

```
AFRAME.registerSystem('my-component', {
  schema: {},  // System schema. Parses into `this.data`.
  init: function () {
    // Called on scene initialization.
  },
  // Other handlers and methods.
});
AFRAME.registerComponent('my-component', {
  init: function () {
    console.log(this.system);
  }
});
```

Accessing a System

An instantiated system can be accessed through the scene:

```
document.querySelector('a-scene').systems[systemName];
```

Registered system prototypes can be accessed through `AFRAME.systems`.

Separation of Logic and Data

Systems can help separate logic and behavior from data if desired. We let systems handle the heavy lifting, and components only worry about managing its data through its lifecycle methods:

```
AFRAME.registerSystem('my-component', {
  createComplexObject: function (data) {
    // Do calculations and stuff with data.
    return new ComplexObject(data);
  }
});

AFRAME.registerComponent('my-component', {
  init: function () {
    this.myObject = null;
  },

  update: function () {
    // Do stuff with `this.data`.
    this.myObject = this.system.createComplexObject(this.data);
  }
});
```

Gathering All Components of a System

There is no strict API for defining how systems manage components. A common pattern is to have components subscribe themselves to the system. The system then has references to all of its

components:

```
AFRAME.registerSystem('my-component', {
  init: function () {
    this.entities = [];
  },

  registerMe: function (el) {
    this.entities.push(el);
  },

  unregisterMe: function (el) {
    var index = this.entities.indexOf(el);
    this.entities.splice(index, 1);
  }
});

AFRAME.registerComponent('my-component', {
  init: function () {
    this.system.registerMe(this.el);
  },

  remove: function () {
    this.system.unregisterMe(this.el);
  }
});
```

Read System online: https://riptutorial.com/aframe/topic/10067/system

Credits

S. No	Chapters	Contributors
1	Getting started with aframe	Community, H. Pauwelyn, M.Kungla
2	Animation	M.Kungla
3	Asset Management System	M.Kungla
4	blend-model (component)	M.Kungla
5	Camera	H. Pauwelyn
6	Components	M.Kungla
7	Controls (component)	H. Pauwelyn
8	cursors	H. Pauwelyn
9	Entities	M.Kungla
10	gltf-model (component)	geekonaut
11	light (component)	H. Pauwelyn
12	Mixins	M.Kungla
13	Primitives	M.Kungla
14	Raycasters (component)	H. Pauwelyn, M.Kungla
15	Scene	H. Pauwelyn, M.Kungla
16	System	M.Kungla