GETTING STARTED WITH VR

VRNM

MAKERS OF THE FUTURE

Table of Contents

Chapter 1:
A Beginner’s Guide to WebVR & A-Frame

1 Getting Started

1.1 Frequently Asked Questions (FAQ)

Does this project involve any programming?

Yes it does! You will be using HTML to create your virtual reality environments, and some JavaScript if your project contains user interactions!

Oh no! What if I have zero programming background!

No worries! This guide is meant to be really beginner friendly and we will be guiding you through each step of the process, so there is no problem even if it is your first time dealing with code. Anyone can do this! :)

How do I use the resources available to me?

Before starting on this document, do check the printed manual on how your working space should be set up. Your computer should have already been installed with the necessary files required to host your own local server for working with the HTML files in this tutorial. See Page 13 of the printed 'Guide to VRNM' manual for more details.

With this document, begin with Sections 1 through 3 before diving into the later projects. The guided project templates in Sections 4 to 6 will require you to have already mastered the basics in order to keep up! While we will still guide you along the way with worked examples, they may seem confusing if you are not yet familiar with the basic concepts. :)

What can expect to learn from this? What can I make?

In Sections 1 to 3, you will be learning the HTML skills required to make your own virtual environments using A-Frame! You can then share this with others for them to explore using their web browser. There is also short introduction on JavaScript towards the end to add interactivity to your project! The subsequent guided project templates in Sections 4 to 6 will also teach you how to build more interesting stuff like a virtual garden, an interactive story world, and a virtual piano

Do note that this is an introductory guide to A-Frame meant to help beginners get started on making VR products of their own. It may be a little too slow for you if you already have a decent grasp of HTML and JavaScript.

How long will this tutorial take me?

Sections 1 to 3 will take approximately 120 minutes to complete in total, with each bite-sized subsection taking about 10 to 15 minutes. Sections 4, 5, and 6 are for users ready to create their own projects and will take a varying amount of time depending on the proficiency of the user.

Where can I find other tutorials if I want to learn more?

Sections 1 to 3 contains a simplified version of what can be found at A-Frame School and the various chapters under the A-Frame 9.0 documentation. You can also look at sample projects on the A-Frame homepage by clicking on the different examples under the side-panel. There are also various projects on Glitch that you can remix into your own project! Additional functionalities you might want to add to your projects can be found at A-Frame Registry (we will briefly cover this under **Section 2.6**).

Other than looking for tutorials online, the best way to learn is to experiment and keep making! You can also ask for help from others on their official Slack or Discord channels, or search for similar questions that others may have asked on StackOverflow!

1.2 Tools Required

"A-Frame can be developed from a plain HTML file without having to install anything"
from A-Frame Documentation - Introduction

First, you will need any basic **text editor** to edit your project. *Notepad.exe* comes with every Windows computer and can be used for this project, but I personally prefer using *Sublime Text 3* which can be downloaded for free here. Popular alternatives include *Notepad++* or *Atom*. One of the benefits of having such a text editor is the in-built automatic colour schemes which helps with the readability of your code for easier editing! *Sublime Text 3* should already be installed on all computers in the Makerspace.

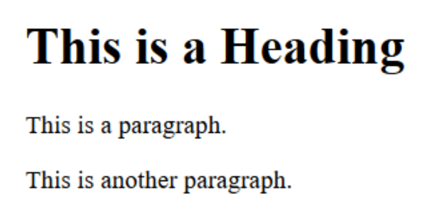
Next, you will need any **web browser** that supports A-Frame WebVR. I'm using Mozilla Firefox on Windows, but a list of compatible browsers can be found here.

That's all you need to start working on your very first VR project! In the next sections, we will guide you through each step of developing a simple VR scene.

1.3 What is HTML?

HTML, or Hyper Text Markup Language, is the language used for creating the webpages that you see on the Internet. Webpages are made up of **HTML elements** consisting of a *start* and *end tag* using the <> brackets, with its content in between. For example, a simple HTML webpage (see *00_sample.html*) consists of the following structure (left) and its resulting webpage would look like the image on the right:

```
<!DOCTYPE html>
<html>

<head>
    <title>Page Title</title>
</head>

<body>
    <h1>This is a Heading</h1>
    <p>This is a paragraph.</p>
    <p>This is another paragraph.</p>
</body>

</html>
```



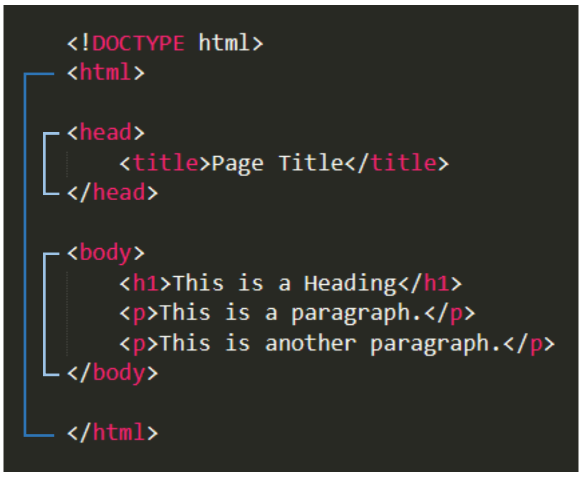
This is a Heading

This is a paragraph.

This is another paragraph.

Adapted from https://www.w3schools.com/whatis/whatis_html.asp

All HTML webpages have a *<head>* and a *<body>* section contained within a larger *<html>* tag, as show below. As you can see in the example above, only the information enclosed by the *<body>* tags will be visible on the webpage. That's all you need to know for now!



```
<!DOCTYPE html>
<html>

<head>
    <title>Page Title</title>
</head>

<body>
    <h1>This is a Heading</h1>
    <p>This is a paragraph.</p>
    <p>This is another paragraph.</p>
</body>

</html>
```

While you do not need to know what each the others tags mean for this project, you can read about them at the W3Schools page if you wish to find out more.

1.4 What is A-Frame?

"A-Frame is a web framework for building virtual reality (VR) experiences."
from [A-Frame Documentation - Introduction]

A-Frame is a great way to get started with developing Virtual Reality content. It provides various tools with which you can create your VR environment within a web browser just by editing your HTML file—all you need to do is include a reference to the A-Frame library under the *<head>* section of your HTML webpage to get started! For example, your *<head>* section may look like this:

```
<head>
    <title>My First VR Project!</title>
    <script src="https://aframe.io/releases/0.9.0/aframe.min.js"></script>
</head>
```

Don't worry if this is confusing to you! You can simply replace the *<head>* section of your HTML webpage with the above lines, or just use the ready-made pages we have provided for the below tutorials in **Section 2**.

2 Step-by-Step Introduction to A-Frame

Before we start, you should replace the *C:/A-Frame/Getting Started* folder on your laptop using the files downloaded from this link to ensure that all files are up to date:

https://github.com/VR-Makerspace/getting-started

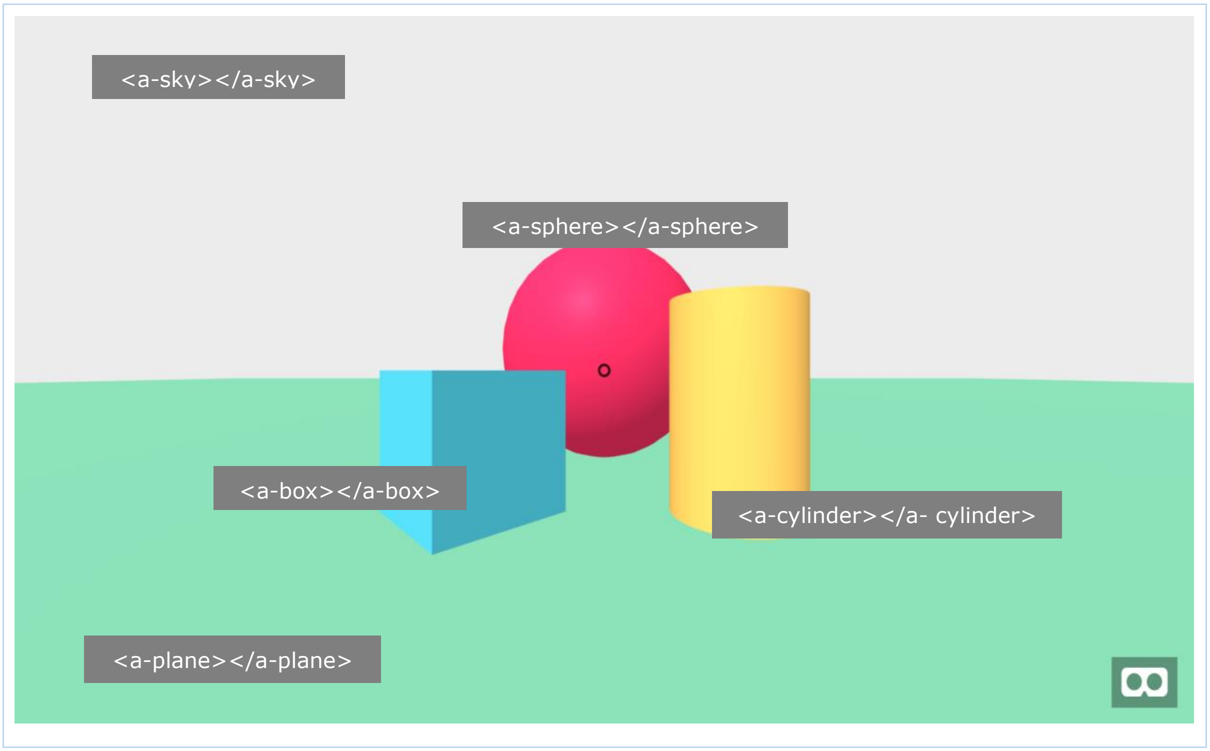
Also, please check out the guide on setting up your local server to work with the HTML files that we will be using for this tutorial. You should be hosting the folder:

…/Beginner's Guide to WebVR and A-Frame

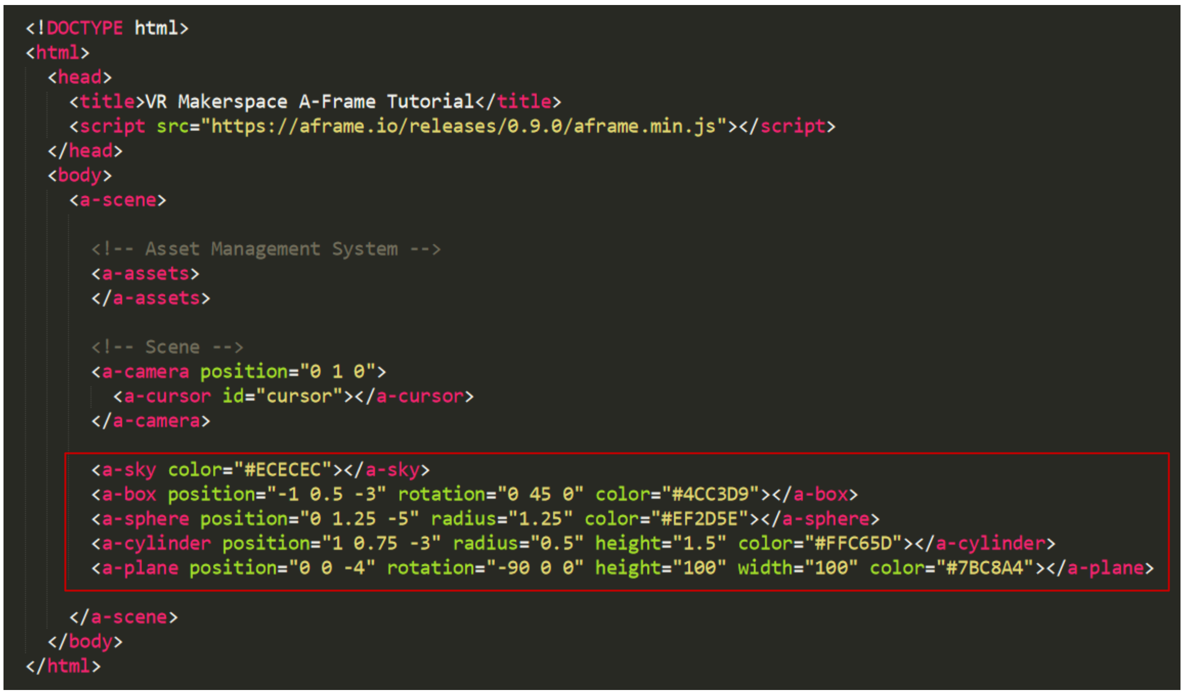
for this Section and the next, found in the Getting Started folder that you've just downloaded. Details on how to set up a local server using command prompt can be found in Page 13 Step 6 of the printed 'Guide to VRNM' manual located in your workspace.

2.1a Introduction to Primitives

01_introduction.html is the template file that we will be using for this tutorial. Opening the file in your browser shows you the following scene. We will first learn how to create the VR environment on your computer without the use of a VR headset. Click and drag to look around the scene, or use WASD keys to move around. You can also click the button on the bottom right to enter full-screen mode.



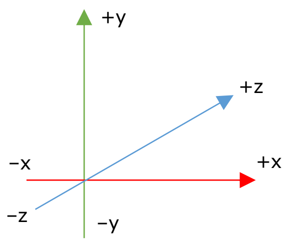
In this scene, there are 5 objects: a box, sphere, and cylinder, as well as the sky and the floor we appear to be standing on. This is given by the following lines of code shown in the red box. You can open ***01_introduction.html*** with your text editor to see its underlying code:



```
<!DOCTYPE html>
<html>
  <head>
    <title>VR Makerspace A-Frame Tutorial</title>
    <script src="https://aframe.io/releases/0.9.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>

      <!-- Asset Management System -->
      <a-assets>
      </a-assets>

      <!-- Scene -->
      <a-camera position="0 1 0">
        <a-cursor id="cursor"></a-cursor>
      </a-camera>

      <a-sky color="#ECECEC"></a-sky>
      <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" height="100" width="100" color="#7BC8A4"></a-plane>

    </a-scene>
  </body>
</html>
```

Each shape, called a "primitive", is created using a pair of opening and closing html tags such as <a-box> and </a-box>. Each object has various attributes (in green text), such as position or colour, that can be modified by assigning a value (in yellow text) to it. A list of A-Frame primitives can be found in the documentation here.

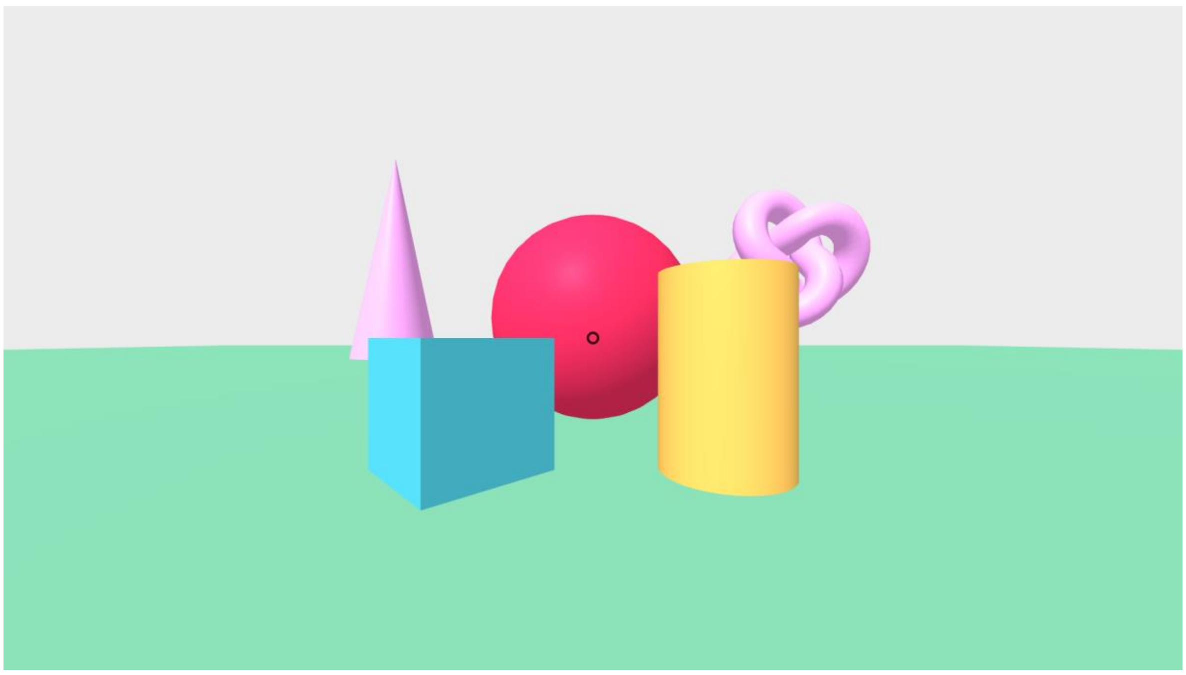
Note that the 3D coordinates used for the position and rotation attributes are in (x, y, z) format. The x-axis, y-axis, and z-axis can be visualized in 3D space with the below image. The best way to learn how 3D space works is to try changing the different values and see what happens!



You can try adding more primitives to the scene. Add the following lines of code within the <a-scene> tags on the line after the plane primitive, then reload the page in your browser. You can also try changing the position, size, and colour attributes of these primitives.

```
<a-cone position="-5 3 -10" height="5" color="#FFAAFF"></a-cone>
<a-torus-knot position="5 3 -10" color="#FFAAFF"></a-torus-knot>
```

See ***introduction-sol.html*** for the final product. You should see the following in your scene.



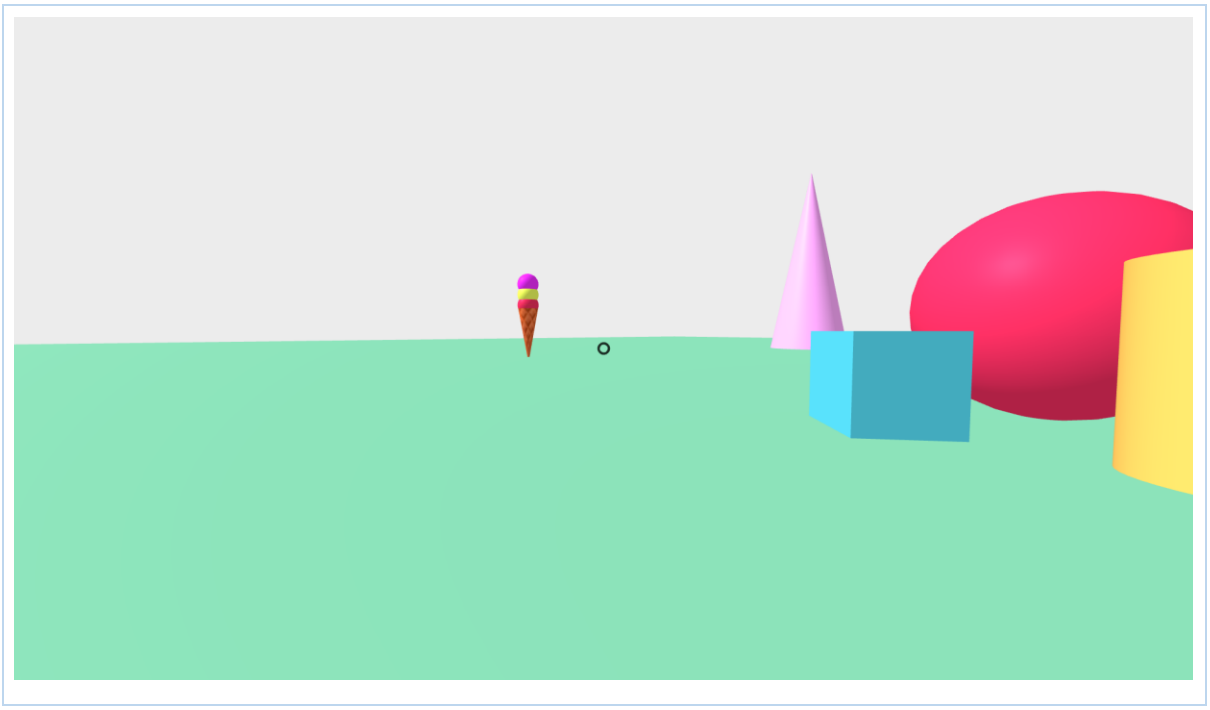
2.1b Relative Positioning

The position of these primitives can be tagged to one another. For example, to create an ice-cream cone, we first create a cone in our scene and then nest three spheres within the opening and closing tags of the cone. This sets the starting position of each nested sphere to the center of the cone rather than the scene’s center. We then only need to change the y-position of the spheres (e.g. -0.25, -0.35, -0.45) to stack them on top of the cone.

Add the following lines of code within the <a-scene> tags. Try playing around with the different position values to see what they do. Do note that if you change the rotation value of the cone, the spheres will rotate along with it as one entity!

```
<a-cone position="-3 1 0" rotation="180 0 0" height="0.5" radius-bottom="0.1" color="#FFAAAA">
    <a-sphere position="0 -0.25 0" radius="0.1" color="#EF2D5E"></a-sphere>
    <a-sphere position="0 -0.35 0" radius="0.1" color="#EFFF5E"></a-sphere>
    <a-sphere position="0 -0.45 0" radius="0.1" color="#EF2DFF"></a-sphere>
</a-cone>
```

See ***01_introduction-sol2.html*** for the final product. The basic ice-cream cone should appear directly on your left (turn towards your left by clicking and dragging using your mouse).



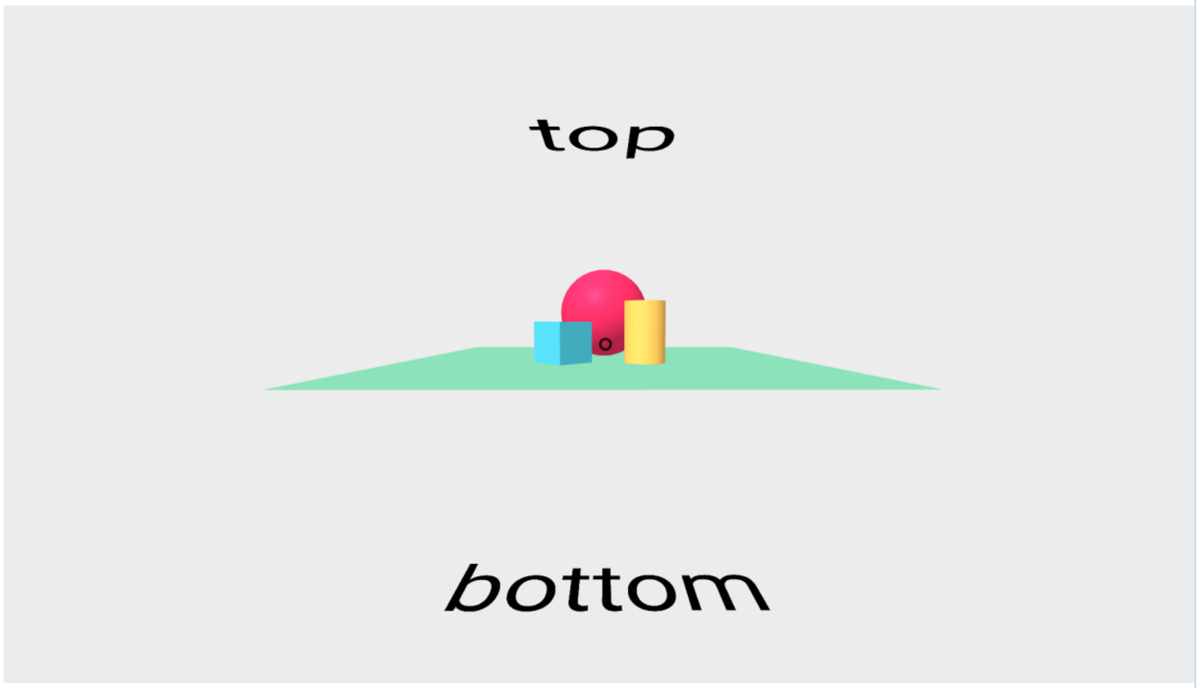
Note that in the solution file, a texture has been applied to the ice-cream cone. We will learn to do this in a later section.

2.2 Adding Text

You can also add text to your scene. For example, add the following lines to ***02_text.html***. These lines add the word "top" above you, and "bottom" below you.

```
<a-text value="top" align=center height="50" width="50" position="0 5 -2" rotation="90 0 0"
color="#000000"></a-text>
<a-text value="bottom" align=center height="50" width="50" position="0 -5 -2" rotation="-90 0
0" color="#000000"></a-text>
```

See ***02_text-sol.html*** for the final product.



2.3 Adding Image & Video

You can also import static images or even play videos in your VR scene. We will be adding the VRNM Makerspace logo and a free stock video of a rotating Earth to our scene, beginning from **03_imgvid.html**. Both asset files can be found in the *\assets\03* folder.

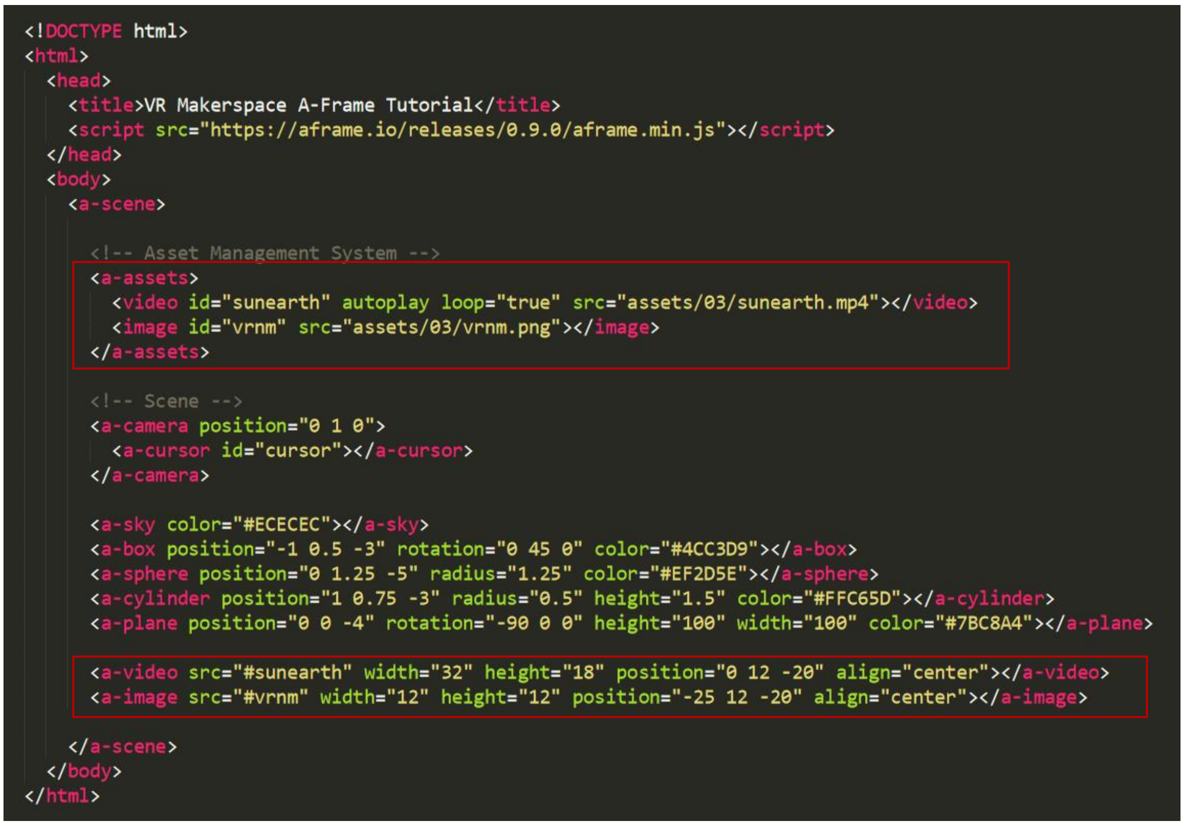
First, we preload both files using the A-Frame Asset Management System. This is done by putting the following lines within the <a-assets> tags.

```
<video id="sunearth" autoplay loop="true" src="assets/03/sunearth.mp4"></video>
<image id="vrnm" src="assets/03/vrnm.png"></image>
```

We then place the image and video in our scene by adding these two lines to the scene.

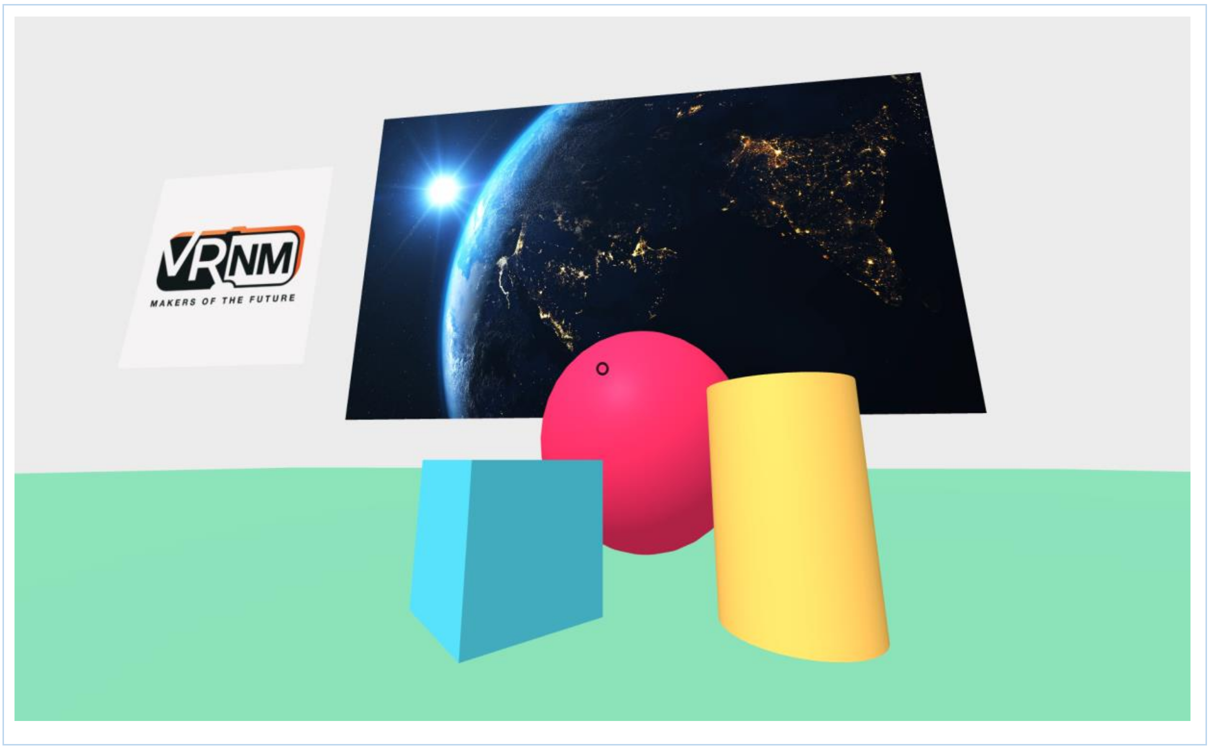
```
<a-video src="#sunearth" width="32" height="18" position="0 12 -20" align="center"></a-video>
<a-image src="#vrnm" width="12" height="12" position="-25 12 -20" align="center"></a-image>
```

This should look like the following:



```
<!DOCTYPE html>
<html>
  <head>
    <title>VR Makerspace A-Frame Tutorial</title>
    <script src="https://aframe.io/releases/0.9.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>

      <!-- Asset Management System -->
      <a-assets>
        <video id="sunearth" autoplay loop="true" src="assets/03/sunearth.mp4"></video>
        <image id="vrnm" src="assets/03/vrnm.png"></image>
      </a-assets>

      <!-- Scene -->
      <a-camera position="0 1 0">
        <a-cursor id="cursor"></a-cursor>
      </a-camera>

      <a-sky color="#ECECEC"></a-sky>
      <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" height="100" width="100" color="#7BC8A4"></a-plane>

      <a-video src="#sunearth" width="32" height="18" position="0 12 -20" align="center"></a-video>
      <a-image src="#vrnm" width="12" height="12" position="-25 12 -20" align="center"></a-image>

    </a-scene>
  </body>
</html>
```

You should be able to view the entire video within the VR scene! Both the video and image are placed behind the primitives we created previously. See ***03_imgvid-sol.html*** for the final product.



2.4 Adding 3D Models

You can also add various types of 3D models to your scene such as glTF models (.gltf or .glb file) or OBJ models and materials (.obj and .mtl files). Here, we will attempt to add a 3D house model to our scene. This can be downloaded from the following link: 'Chalet' by 'Poly by Google', or they can already be found in *\assets\04* in their respective folders.

Similar to before, we preload the required assets under the Asset Management System. Only the first line is required if you are using the glTF model. Alternatively, use the bottom two lines if you wish to use the OBJ model. For this tutorial, add all three lines—we will be using both models in our scene, with the glTF model on the left and OBJ model on the right.

```
<a-asset-item id="chalet-gltf" src="assets/04/glTF/CUPIC_CHALET.gltf"></a-asset-item>
```

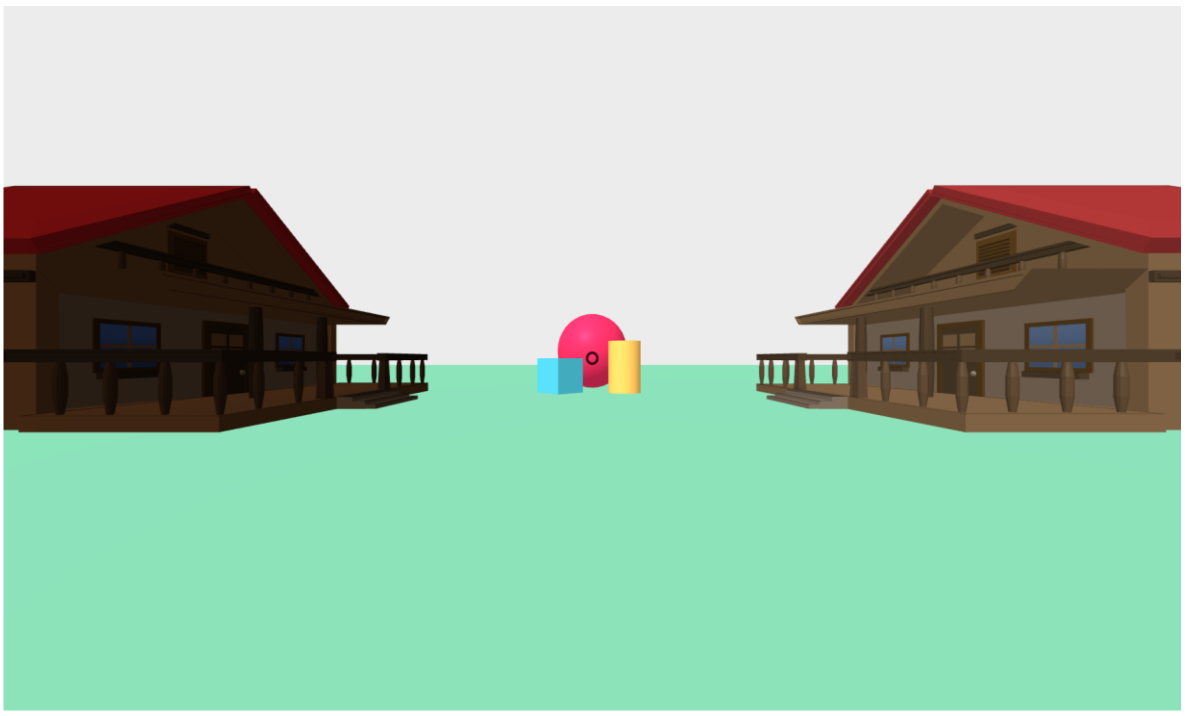
```
<a-asset-item id="chalet-obj" src="assets/04/OBJ/CUPIC_CHALET.obj"></a-asset-item>
<a-asset-item id="chalet-mtl" src="assets/04/OBJ/CUPIC_CHALET.mtl"></a-asset-item>
```

Next, similar to what we did in Section 2.3, we then place the 3D model into our scene. Use the line on the top for the glTF model and the line on the bottom for the OBJ model.

```
<a-entity gltf-model="#chalet-gltf" position="-10 0 0" rotation="0 -90 0" scale="0.01 0.01
0.01"></a-entity>
```

```
<a-obj-model src="#chalet-obj" mtl="#chalet-mtl" position="10 0 0" rotation="0 90 0"
scale="0.01 0.01 0.01"></a-obj-model>
```

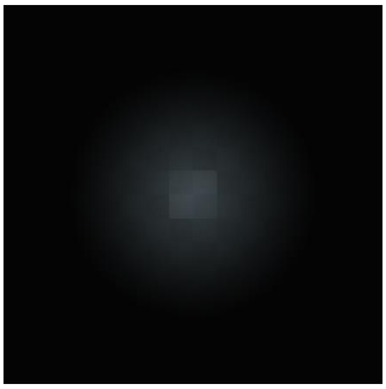
See ***04_model-sol.html*** for the final product. You should see the glTF model on the left and the OBJ model on the right.

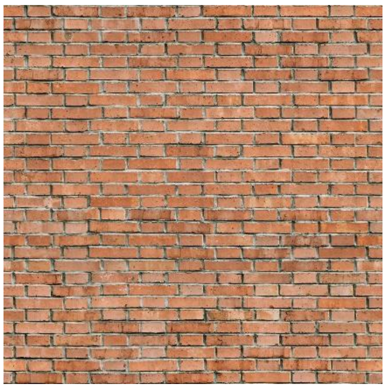


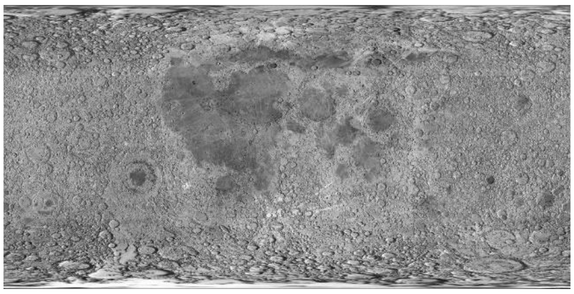
2.5 Adding Textures

We can also add textures to our objects using the *src* attribute under each primitive. For each of the primitives in our scene in ***05_textures.html***, we remove the colour (it isn't required any more) and then specify the image source that we would like to use as our texture. These following are the images we will be using—they can also be found under the *\assets\05* folder.

Sky: https://cdn.glitch.com/b870d9ec-1139-44f9-b462-223e4a2c74e7%2Fsechelt.jpg?1490307995926


Plane: https://cdn.glitch.com/b870d9ec-1139-44f9-b462-223e4a2c74e7%2Ffloor.jpg?1490307896453


Box: https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/bricks/brick_diffuse.jpg


Sphere: https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/space/moon_1024.jpg

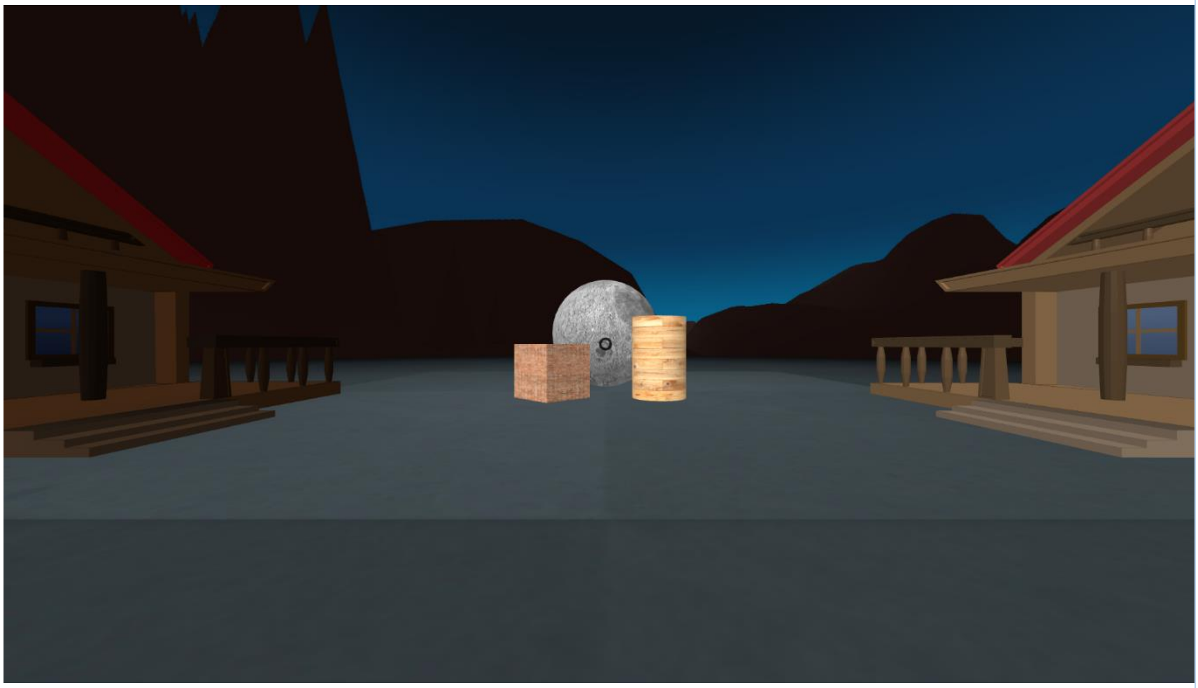
Cylinder: https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/wood/hardwood2_diffuse.jpg

The above images were taken from https://github.com/aframevr/sample-assets and https://glitch.com/~aframe-school-textures/.

This should look like the following:

```
<a-sky radius="30" src="assets/05/texture/texture_sky.jpg"></a-sky>

<a-box position="-1 0.5 -3" rotation="0 45 0" src="assets/05/texture/texture_box.jpg"></a-box>

<a-sphere position="0 1.25 -5" radius="1.25" src="assets/05/texture/texture_sphere.jpg"></a-sphere>

<a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" src="assets/05/texture/
texture_cylinder.jpg"></a-cylinder>

<a-plane position="0 0 -4" rotation="-90 0 0" height="100" width="100" src="assets/05/texture/
texture_plane.jpg"></a-plane>
```

See ***05_textures-sol.html*** for the final product.



2.6 Adding Other Components & Functionalities

The A-Frame Registry is a "curated collection of A-Frame ready-to-use components". With this, we can add other functionalities such as a particle system or animation to our scene. We do this by adding the relevant scripts to the *<head>* section of our ***06_components.html*** file.

For example, in the following *<head>* section we define the source for the A-Frame script, the animation component, as well as the particle system component.

```
<head>
    <script src="https://aframe.io/releases/0.9.0/aframe.min.js"></script>
    <script src="https://unpkg.com/aframe-animation-component@%5E3.2.x/dist/aframe-animation-
    component.min.js"></script>
    <script src="https://unpkg.com/aframe-particle-system-component@1.0.9/dist/aframe-particle-
    system-component.min.js"></script>
</head>
```

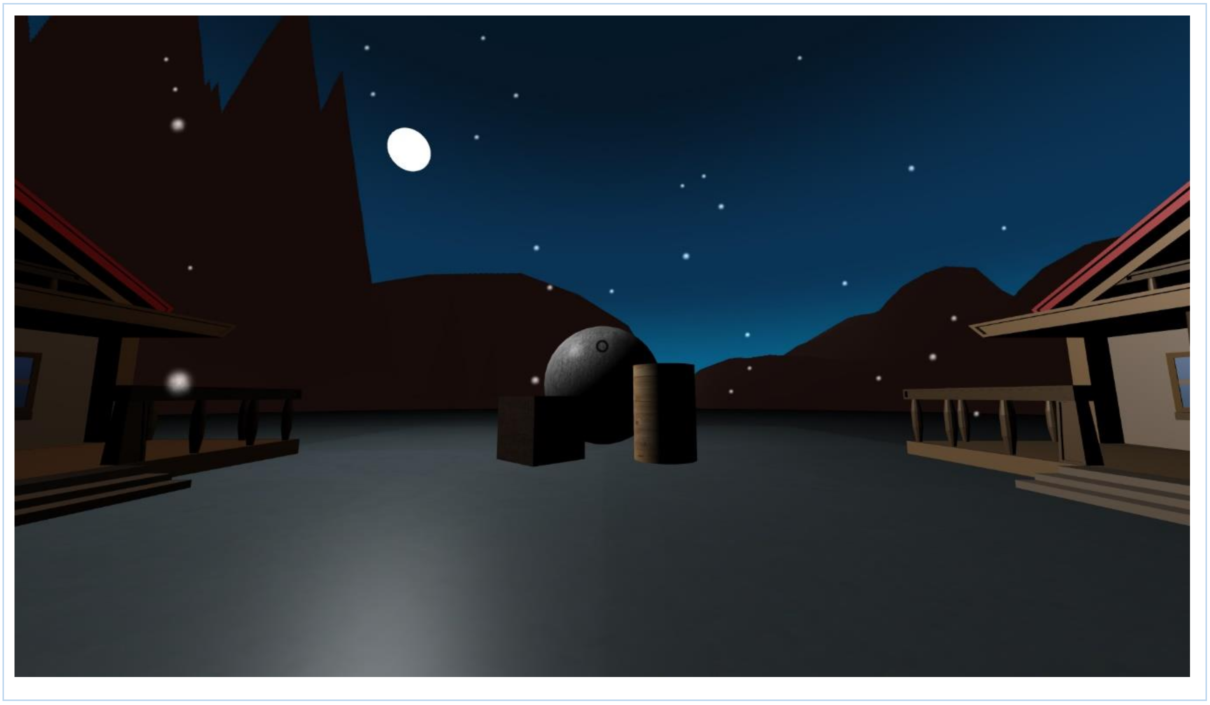
Next, we add these components into our scene. In our first example, we attach the animation component as an attribute of the *lightSphere* entity. You can try adding this animation component to other objects in the scene as well. Read up more on what the animation component can do here.

```
<a-entity id="lightSphere" geometry="primitive: sphere; radius: 0.2" material="shader: flat"
light="type: point; color: #FFF" position="-4 6 -4" animation="property: position; to: 4 6 -4;
dir: alternate; loop: true"></a-entity>
```

We can also add the particle system to our scene. This example uses snow as our particle effect. You can try changing the preset value to "stars" or "rain" as well. More about this can be found here.

```
<a-entity particle-system="preset: snow; particleCount: 10000"></a-entity>
```

You can see them in action in ***06_components-sol.html***.

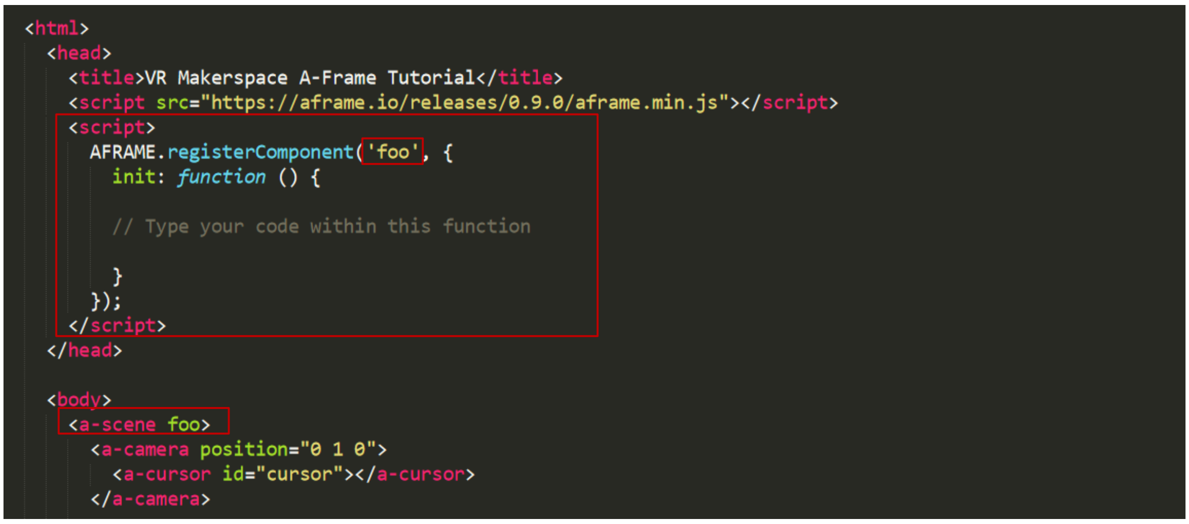


3 Brief Introduction to JavaScript

While our previous examples only consist of creating 3D environments for exploration, the use of JavaScript allows us to design interactions within the VR experience. This, however, may require basic programming knowledge to execute.

3.1 Getting Entities using JavaScript

We will be typing our JavaScript code within a <*script*> section under the <*head*> of the HTML file. A template has already been provided for you. We also add the 'foo' component that we defined in our JavaScript template to <*a-scene*>. We will be using **07_interactions.html** for this section of the tutorial.



```
<html>
  <head>
    <title>VR Makerspace A-Frame Tutorial</title>
    <script src="https://aframe.io/releases/0.9.0/aframe.min.js"></script>
    <script>
      AFRAME.registerComponent('foo', {
        init: function () {

        // Type your code within this function

        }
      });
    </script>
  </head>

  <body>
    <a-scene foo>
      <a-camera position="0 1 0">
        <a-cursor id="cursor"></a-cursor>
      </a-camera>
```

First, we learn how to get a reference to an element in the scene using JavaScript. The *querySelector()* function allows you to refer to the entire <*a-scene*> in your HTML file. Replace the grey comment with the following line:

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

Similarly, now using *sceneEl.querySelector()*, we can refer to individual elements within the scene. Here, we learn to create a reference to an individual box in the scene through its *id*. First, in the HTML code, give the box an *id='box'* attribute. Then, using the '#' symbol for an element's id, we refer to the box in our JavaScript code using '#box' in our querySelector function:

```
var boxEl = sceneEl.querySelector('#box');
```

Do the same for the sphere (with *id='sphere'*) and cylinder (with *id='cylinder'*). Your code should look like that in **07_interactions-sol.html** now. Using these, we can now modify attribute values of these objects through JavaScript in the next section.

3.2 Modifying Entities using JavaScript

Continuing from the above, we can use the *setAttribute()* function to change the value of an attribute of the above elements. This function takes in two parameters—the attribute name, and its new value—separated by a comma. These changes will happen as the HTML page loads, overriding the values you have set in your HTML file.

Using the following lines, we set the colour of the box to grey, rotate the cylinder by 90 degrees, and scale down the size of the sphere to 50%.

```
boxEl.setAttribute('color', 'grey');
cylinderEl.setAttribute('rotation', '0 0 90');
sphereEl.setAttribute('scale', '.5 .5 .5');
```

Refer to **07_interactions-sol2.html** for the final result. Feel free to experiment by modifying the values of other attributes!

3.3 Detecting Interactions using Event-Listeners

Instead of changing the values as the page loads, we can also set these values to change upon user interaction—such as click-based or gaze-based. This is done through the use of *addEventListener()*.

Below is an example of using 'mouseenter' and 'mouseleave' to change the colour of the box. When the mouse is detected over the box, it executes the *setAttribute()* function within it. Do note that the 'mouse' in this case refers not to your mouse-cursor, but the VR-cursor controlled by where you are gazing at in your VR environment.

```
boxEl.addEventListener('mouseenter', function() {
  boxEl.setAttribute('color', 'grey');
});

boxEl.addEventListener('mouseleave', function() {
  boxEl.setAttribute('color', '#4CC3D9');
});
```

Next is an example of using 'click' to change the rotation of the cylinder.

```
cylinderEl.addEventListener('click', function() {
  cylinderEl.setAttribute('rotation', '0 0 90');
});
```

Now try to use either the 'click' or 'mouseenter' and 'mouseleave' interactions to change the size of the sphere. Refer to **07_interactions-sol3.html** for the final result.

Chapter 2:
Guided Project Templates

4 Project #1 – Making a Virtual Garden [Difficulty: Easy]

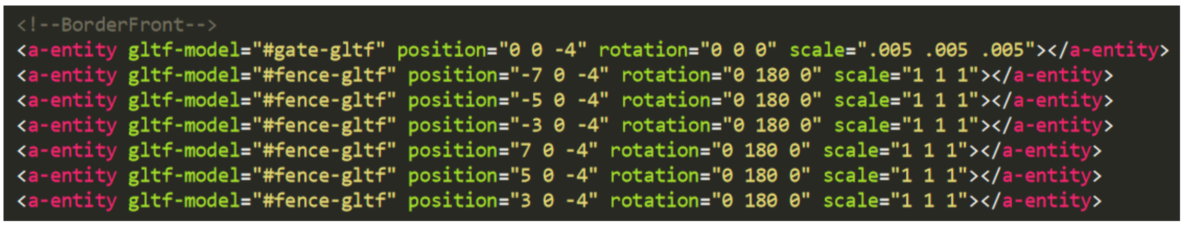
In this project template, we learn how to create our own virtual garden that you can explore with a VR headset. Once you've learnt how you can create a VR world like this, you can apply similar concepts to create other virtual environments from your imagination!

Required assets have already been added to the Asset Management System in the first HTML template. If you get stuck on any step, look at the sample files and see how the HTML code is different from yours.

4.1 Breaking it Down

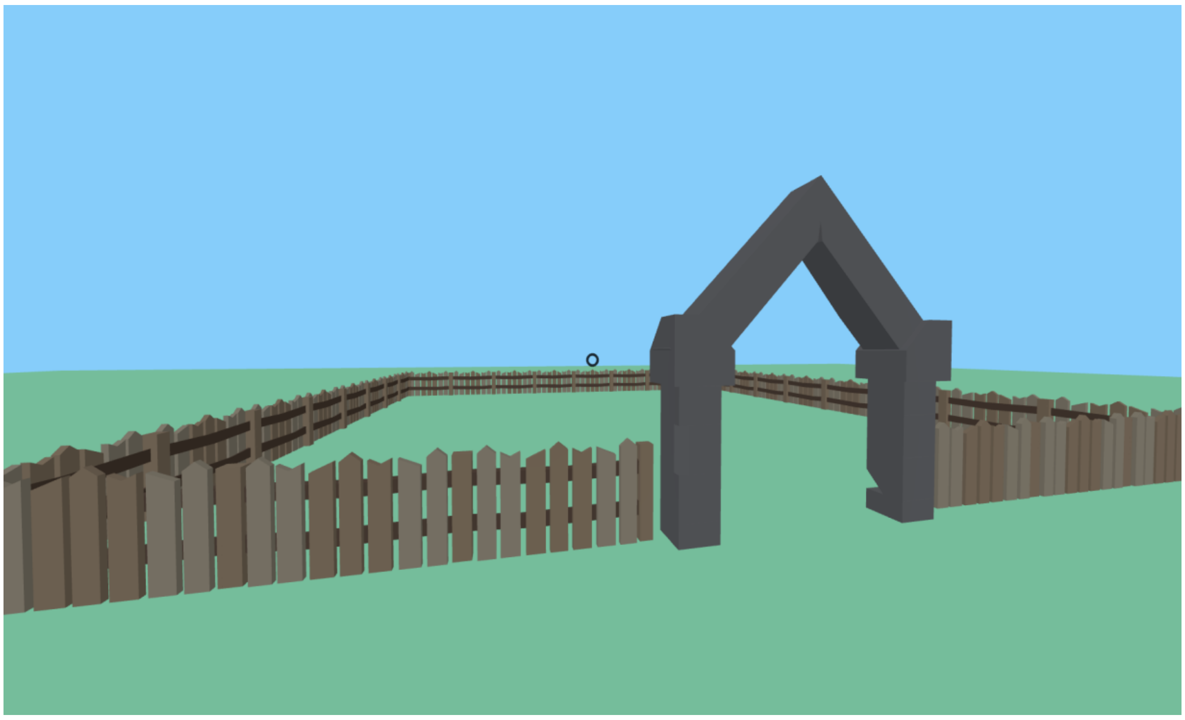
Beginning from **01_garden_empty.html**, we add the "Gate" model to our scene and resize it to fit our garden [Tip: Set its scale to 0.005!]. We place it right in front of us at position "0 0 -4".

Next, we set the boundaries of our garden using our 'Fence' models. At a scale of 1, the models are approximately 2 units long. We place three of these to the left and right sides of the gate. Your front border should be ready:

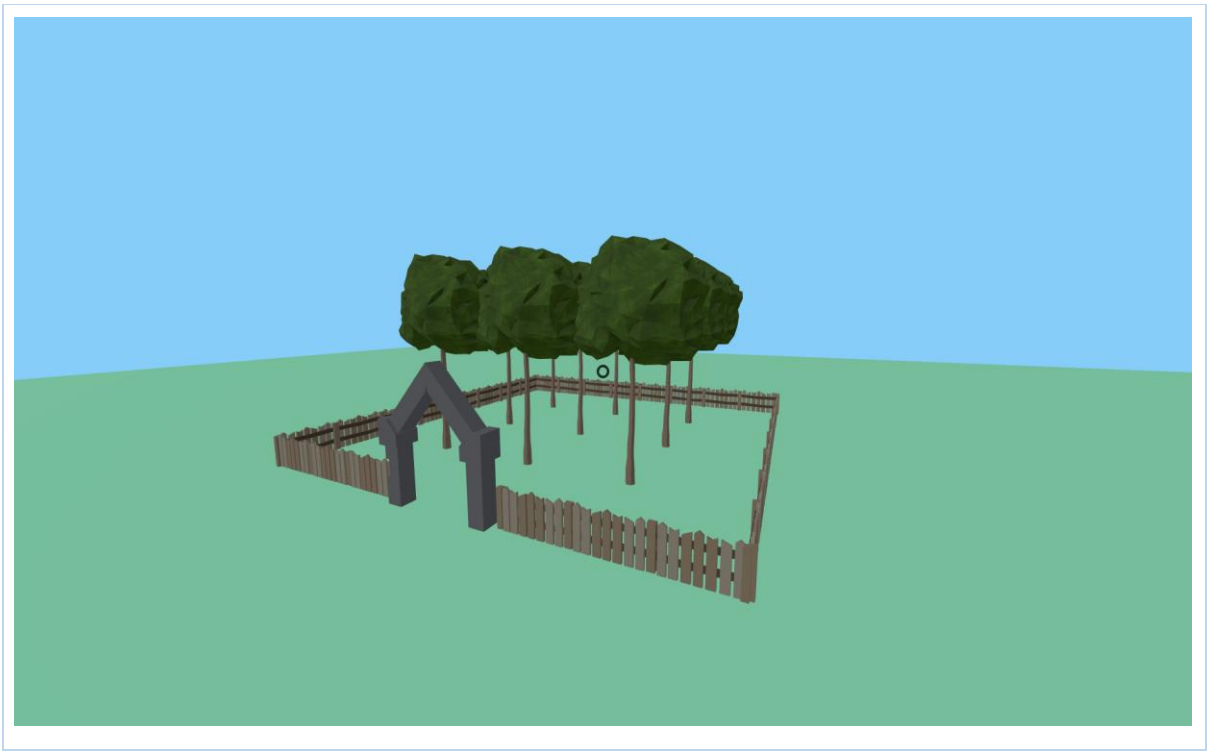


```html
<!--BorderFront-->
<a-entity gltf-model="#gate-gltf" position="0 0 -4" rotation="0 0 0" scale=".005 .005 .005"></a-entity>
<a-entity gltf-model="#fence-gltf" position="-7 0 -4" rotation="0 180 0" scale="1 1 1"></a-entity>
<a-entity gltf-model="#fence-gltf" position="-5 0 -4" rotation="0 180 0" scale="1 1 1"></a-entity>
<a-entity gltf-model="#fence-gltf" position="-3 0 -4" rotation="0 180 0" scale="1 1 1"></a-entity>
<a-entity gltf-model="#fence-gltf" position="7 0 -4" rotation="0 180 0" scale="1 1 1"></a-entity>
<a-entity gltf-model="#fence-gltf" position="5 0 -4" rotation="0 180 0" scale="1 1 1"></a-entity>
<a-entity gltf-model="#fence-gltf" position="3 0 -4" rotation="0 180 0" scale="1 1 1"></a-entity>
```

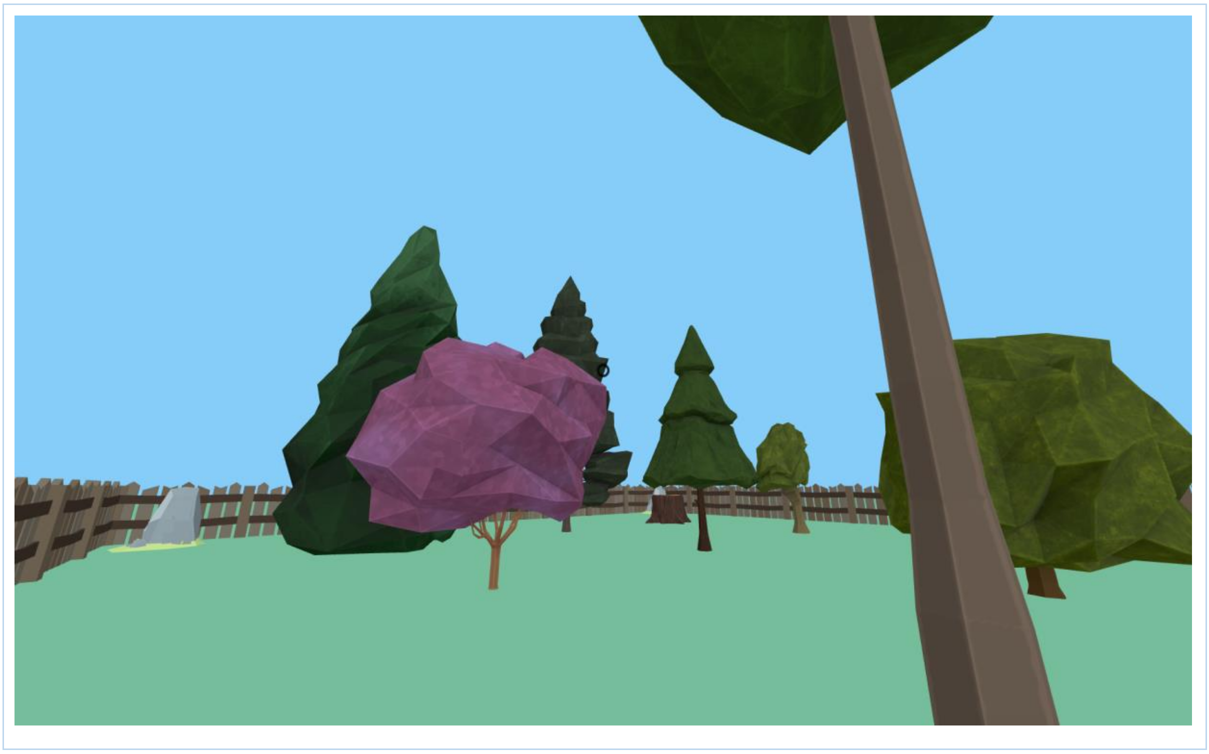
Using the same concept, we fill in the other sides of the boundaries by adding more 'Fence' models and setting their position and rotation attributes. Try to have all sides of equal length! It should look like the that in **02_garden_fence.html**:



In 3D space, the dimensions of this square garden should be from -8 to 8 along the x-axis, and -4 to -20 along the y-axis. Next, we arrange 'Trees' within our garden, in a 3x3 grid, For now, just use the 'brazilNutTree' model at a scale of 0.5. It should look like that in **03_garden_trees.html**.



You are almost done! Now you can simply change the tree models to any of the other models already in the Asset Management System or even load your own models. You can even rearrange or expand your garden. Try it out! For an example, check out **04_garden_grid_final.html**.



4.2 Other Suggestions

a. Try changing the 3D models in this garden, or changing its arrangement to suit your own tastes. It doesn't have to be a grid, so be creative!

b. Perhaps create a fruit plantation, a savanna, or a rainforest! Looking through Google Images using these keywords may give you the inspiration you need on what worlds you want to create.

c. Try transforming and expanding this garden into a full-fledged zoo! You can add sections for mammals, birds, reptiles, and even mythical beasts.

5 Project #2 – Making a Storytelling World [Difficulty: Medium]

In this project template, we learn how to create a world with NPCs that tell a simplified version of 'The Little Red Riding Hood' when interacted with. With this, you can create other immersive worlds that feel more than just a walking-simulator—with deeper stories and even mysteries behind them! Challenge your storytelling abilities in a 3D virtual world!

Required assets have already been added to the Asset Management System in the first HTML template. If you get stuck on any step, look at the sample files and see how the HTML code is different from yours!

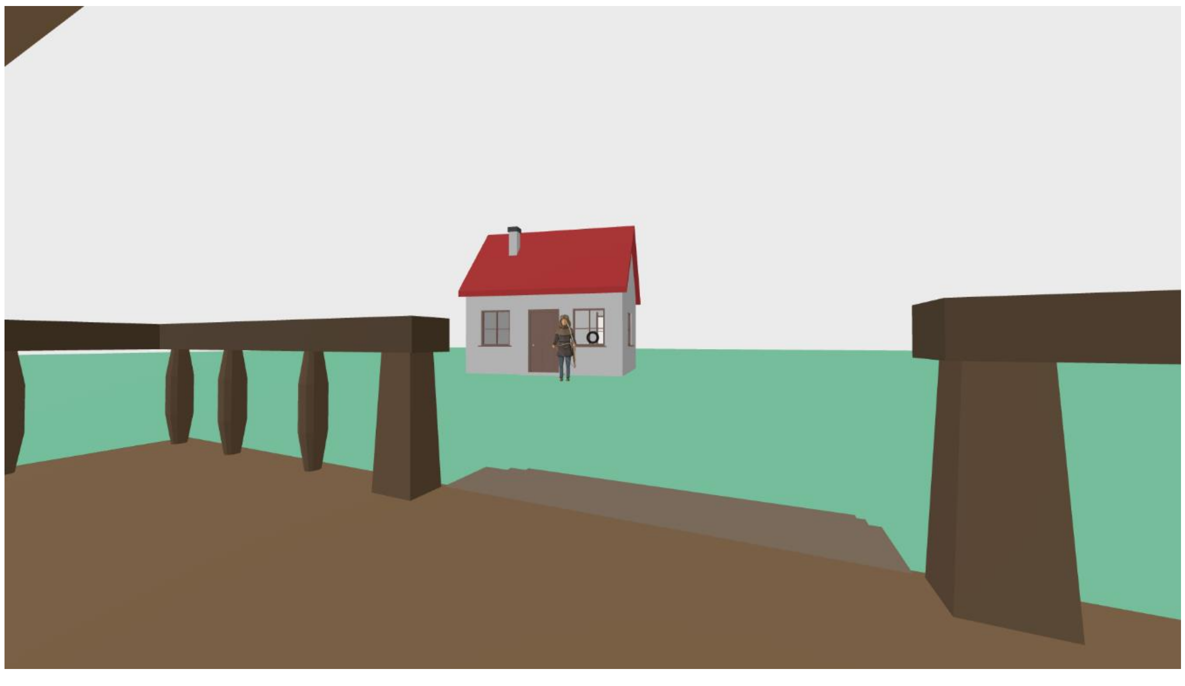
5.1 Breaking it Down

Before you start a story, you'll have to know who your main characters are. Using these, you can form the backbone of your story by deciding the rough order that a player will meet them in. In our version of the Little Red Riding Hood, we will have the following characters:

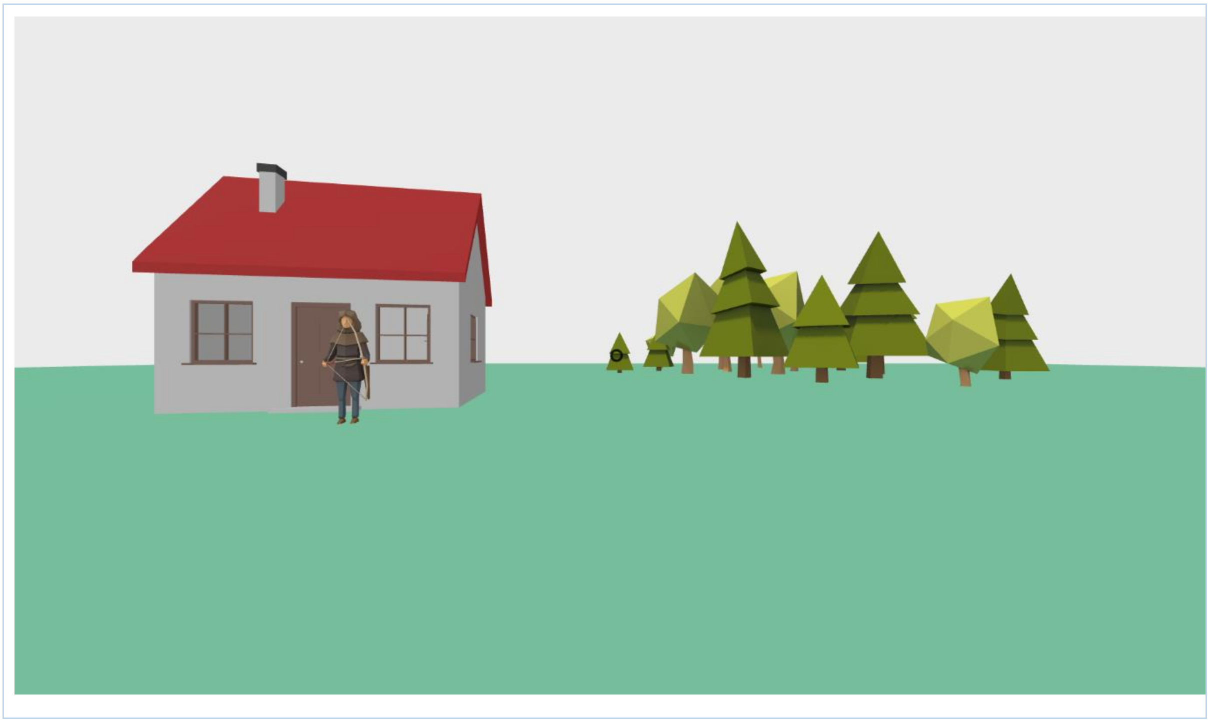
- Little Red Riding Hood starts off at her village.
- A Hunter lives in the village.
- A Big Bad Wolf lives in the forest near her village.
- Her Grandmother lives on the other side of the forest.

Let us first begin by designing our village, starting off with **01_littleredridinghood_empty.html**. This can be done however you like using the available 3D models, but it should have (i) Little Red Riding Hood's house where the player will start, as well as (ii) the Hunter's house with the Hunter standing outside it.

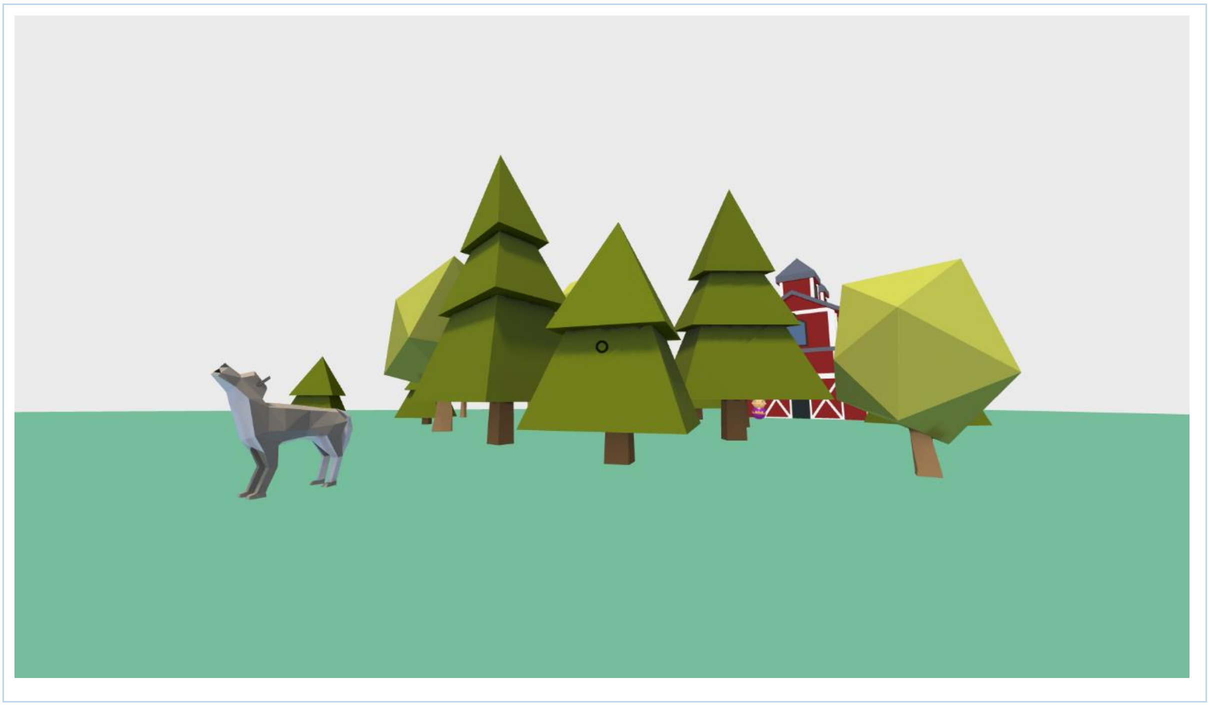
For example, in **02_littleredridinghood_village.html**, we add a 'chalet' model behind us, and a smaller house and the Hunter in front of us and a little to the left.



Next, put a forest in front of you and to the right of the village. You can either do this by using the 3D model of a forest that we've provided, or use individual trees to design your own forest (you can find some from the Garden project in Section 1).



In front of the forest put a 'Wolf' model. Behind that forest, put a 'Grandmother' model and a 'Barn'. Check out **03_littleredridinghood_forest.html** for an example:

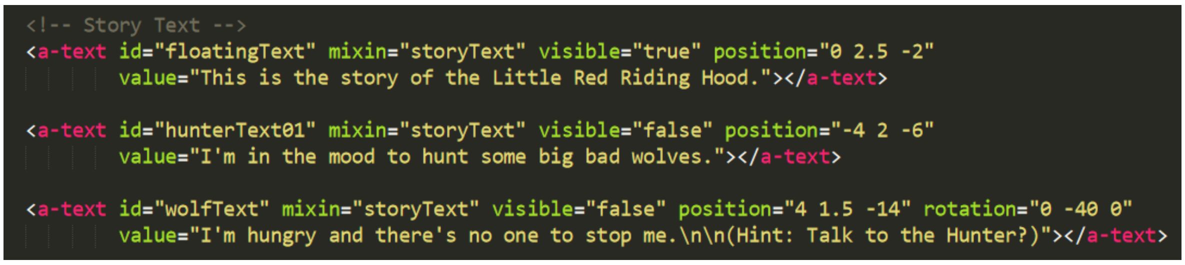


Now we are done with designing our VR space! To add story elements, we will be using *<a-text>* as well as some JavaScript to control the interactions.

ADDING THE STORY – PART 1:

You can change the contents of the story however you like, but the linear story in our example will be as follows:

- When the page loads, the player sees "This is the story of the Little Red Riding Hood".
- The Hunter hints "I'm in the mood to hunt some big bad wolves" when clicked on.
- The Wolf says "I'm hungry and there's no one to stop me" when clicked on.

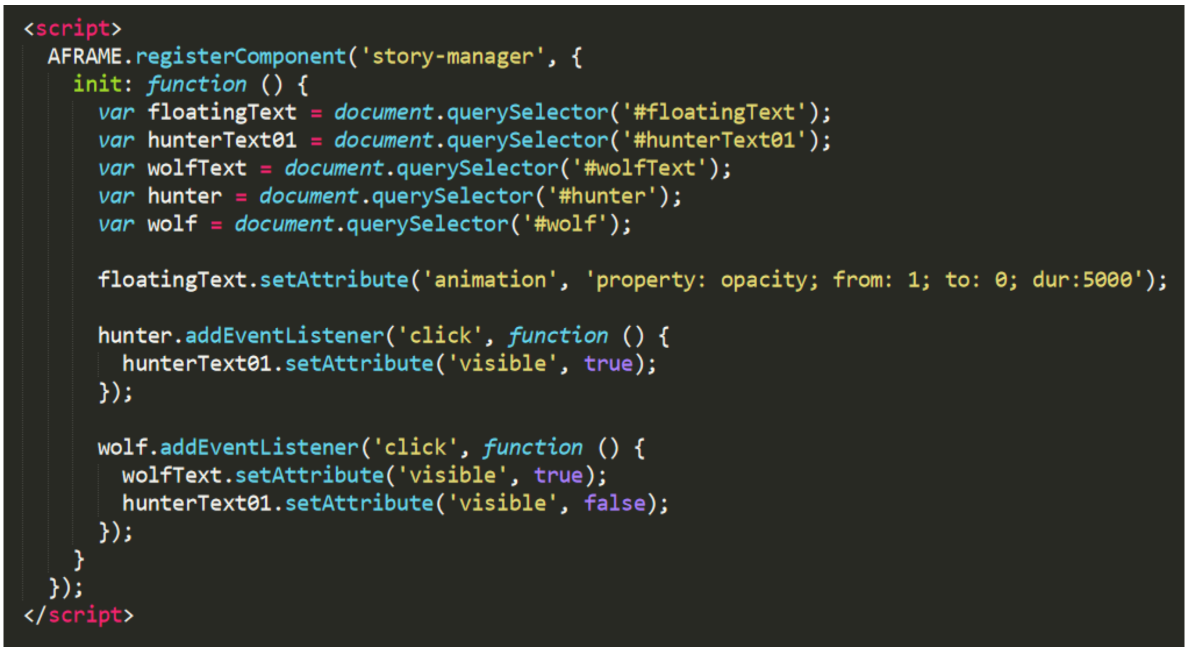


```
<!-- Story Text -->
<a-text id="floatingText" mixin="storyText" visible="true" position="0 2.5 -2"
        value="This is the story of the Little Red Riding Hood."></a-text>

<a-text id="hunterText01" mixin="storyText" visible="false" position="-4 2 -6"
        value="I'm in the mood to hunt some big bad wolves."></a-text>

<a-text id="wolfText" mixin="storyText" visible="false" position="4 1.5 -14" rotation="0 -40 0"
        value="I'm hungry and there's no one to stop me.\n\n(Hint: Talk to the Hunter?)"></a-text>
```

You will notice we are using a mixin to format all our texts. The following line of text is added under the Asset Management System so that the changes to alignment, color, and size are applied to all elements tagged with the *mixin="storyText"* attribute.

```
<a-mixin id="storyText" text="align:center; color:black" scale="0.4 0.4 0.4"></a-mixin>
```

You will notice that under *wolfText*, '*\n*' is used. This adds a line break to your sentence so that the following text is on a separate line. Also note that only *floatingText* is set to *visible="true"*. The other lines are first set to invisible. These are only made visible when clicked on, using the following:



```
<script>
  AFRAME.registerComponent('story-manager', {
    init: function () {
      var floatingText = document.querySelector('#floatingText');
      var hunterText01 = document.querySelector('#hunterText01');
      var wolfText = document.querySelector('#wolfText');
      var hunter = document.querySelector('#hunter');
      var wolf = document.querySelector('#wolf');

      floatingText.setAttribute('animation', 'property: opacity; from: 1; to: 0; dur:5000');

      hunter.addEventListener('click', function () {
        hunterText01.setAttribute('visible', true);
      });

      wolf.addEventListener('click', function () {
        wolfText.setAttribute('visible', true);
        hunterText01.setAttribute('visible', false);
      });
    }
  });
</script>
```

Check out JavaScript tutorial in Section 3 of the ________ to recap on how this works! In the above section of code, there are four main parts to note:

- We first use *document.querySelector()* to get a reference to all the elements we need.
- We use *setAttribute()* to animate *floatingText* so that it fades out after 5 seconds.
- We use *addEventListener()* to detect when the Hunter has been clicked on, and then set the prepared text to visible.
- We use *addEventListener()* to detect when the Wolf has been clicked on, and then set the prepared text to visible, and set the Hunter's previous text back to not visible.

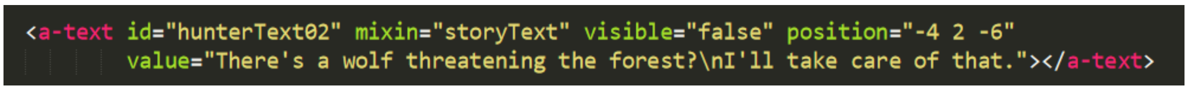
See **04_littleredridinghood_story.html** to see it in action!

ADDING THE STORY – PART 2:

At this point, we now want the Hunter to help get rid of the Wolf. This requires the following interactions:

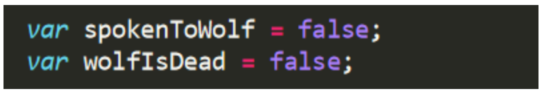
- If the Hunter is clicked on after the player has spoken to the Wolf, he will say: "There's a wolf threatening the forest? I'll take care of that".
- Then we rotate the Wolf so that it looks like it is lying dead on the floor, and ensure it stops responding to any more clicks.

Add the above as a new *<a-text>* element and name it HunterText02:



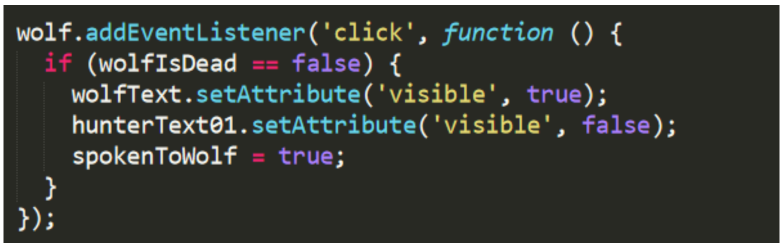
```
<a-text id="hunterText02" mixin="storyText" visible="false" position="-4 2 -6"
        value="There's a wolf threatening the forest?\nI'll take care of that."></a-text>
```

To do the first step, we first have to keep track of whether the player has spoken to the Wolf. Create a variable called *spokenToWolf* and set it to '*false*'. For the second step, we need to keep track of whether the Hunter has gone to kill the Wolf. Create another variable called *wolfIsDead* and set it to '*false*'. We will set them to '*true'* when the events have been fulfilled.



```
var spokenToWolf = false;
var wolfIsDead = false;
```

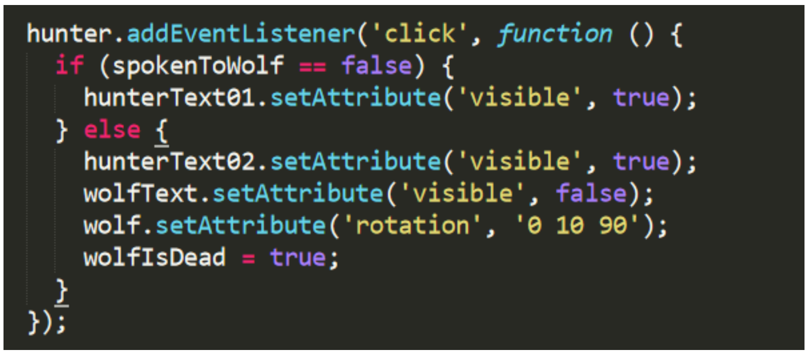
Let us start with how the Wolf should behave. If the Wolf is alive and clicked on, it will do the same things as before AND also change *spokenToWolf* to '*true*'. If the Wolf is dead, it will do nothing.



```
wolf.addEventListener('click', function () {
  if (wolfIsDead == false) {
    wolfText.setAttribute('visible', true);
    hunterText01.setAttribute('visible', false);
    spokenToWolf = true;
  }
});
```

For the Hunter, if the player has not spoken to the Wolf, it will do the same things as before. If *spokenToWolf* is 'true', then:

- Set the new *HunterText02* to visible to show the player that he will get rid of the Wolf.
- Then, set *WolfText01* to invisible and rotate the Wolf so that it looks dead.
- Change *wolfIsDead* to '*true*'.



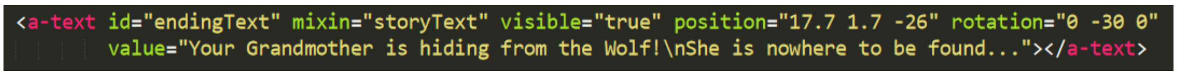
```
hunter.addEventListener('click', function () {
  if (spokenToWolf == false) {
    hunterText01.setAttribute('visible', true);
  } else {
    hunterText02.setAttribute('visible', true);
    wolfText.setAttribute('visible', false);
    wolf.setAttribute('rotation', '0 10 90');
    wolfIsDead = true;
  }
});
```

Remember to use *querySelector()* to make a reference to the *hunterText02* element for this to work.

As for the Grandmother, she will not appear unless the Wolf is dead.

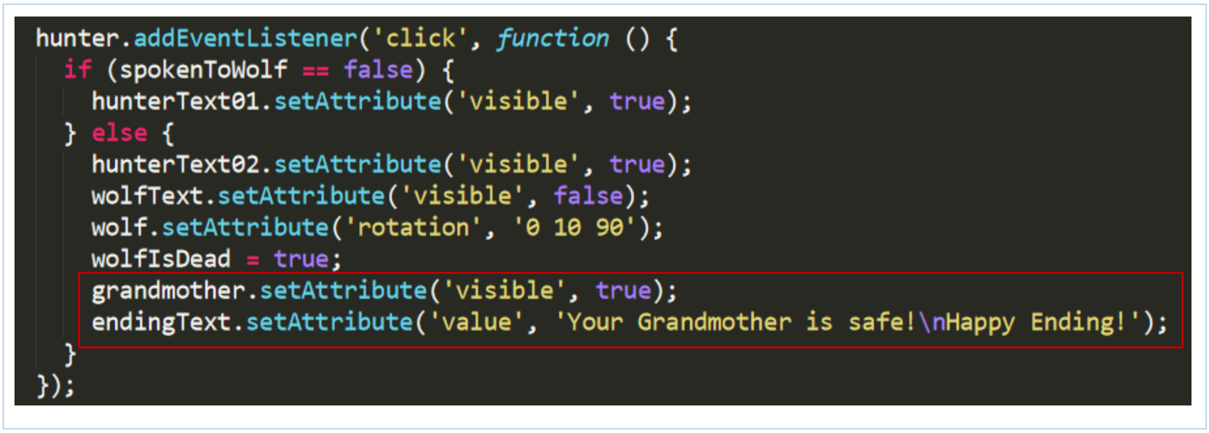
- If the Wolf is not dead, the area outside the barn will say: "Your Grandmother is hiding from the Wolf! She is nowhere to be found..."
- If the Hunter has gotten rid of the wolf, the Grandmother will appear and the text will say "Your Grandmother is safe! Happy Ending!" instead.

To do this, first set the Grandmother element to invisible (under the HTML portion). Also, make a new *<a-text>* element called *endingText*. We will set the value of the ending text to always show the first line of text, then change it when the Hunter kills the wolf.



```
<a-text id="endingText" mixin="storyText" visible="true" position="17.7 1.7 -26" rotation="0 -30 0"
        value="Your Grandmother is hiding from the Wolf!\nShe is nowhere to be found..."></a-text>
```

Now, when the Hunter kills the Wolf, two things must happen. Set Grandmother to visible, then change the value of *endingText* to the new happy ending. Two lines of code have been added to this section.



```
hunter.addEventListener('click', function () {
  if (spokenToWolf == false) {
    hunterText01.setAttribute('visible', true);
  } else {
    hunterText02.setAttribute('visible', true);
    wolfText.setAttribute('visible', false);
    wolf.setAttribute('rotation', '0 10 90');
    wolfIsDead = true;
    grandmother.setAttribute('visible', true);
    endingText.setAttribute('value', 'Your Grandmother is safe!\nHappy Ending!');
  }
});
```

That's it! You've completed your own Little Red Riding Hood story. For an example of how it should work out, check out **05_littleredridinghood_final.html**.

5.2 Other Suggestions

a. Try changing up the story by changing the interaction texts as well as the order in which the player interacts with the different NPCs.

b. You can also try making another storytelling world for other linear short stories such as 'The Boy Who Cried Wolf'.

c. Look at other story-rich games for inspiration on what else you might want to tell a story about!

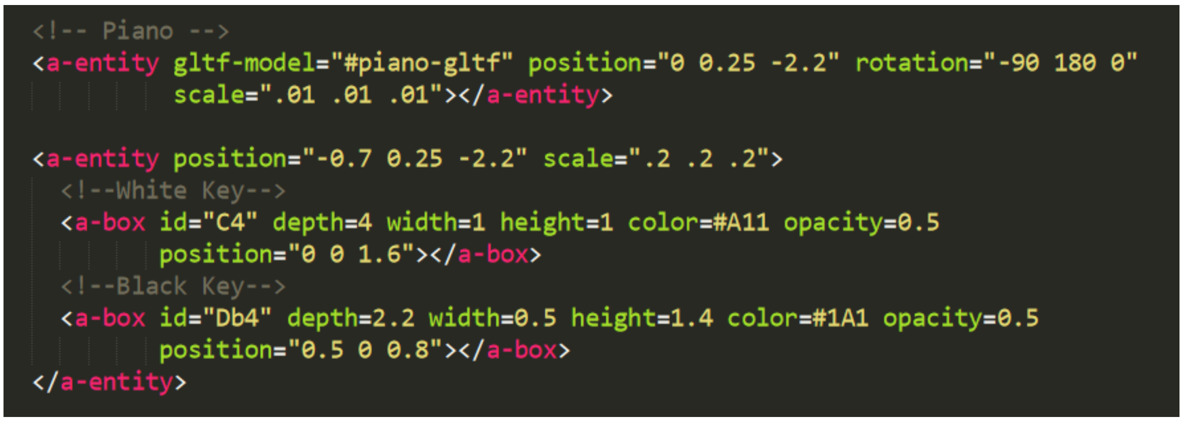
6 Project #3 – Making an Interactive Piano [Difficulty: Hard]

In this project template, we learn how to create an interactive piano, which we further develop into a music-based puzzle game later on. Through this, you will learn even more basic JavaScript skills that you can use for your future projects and games!

Required assets have already been added to the Asset Management System in the first HTML template. If you get stuck on any step, look at the sample files and see how the HTML code is different from yours!

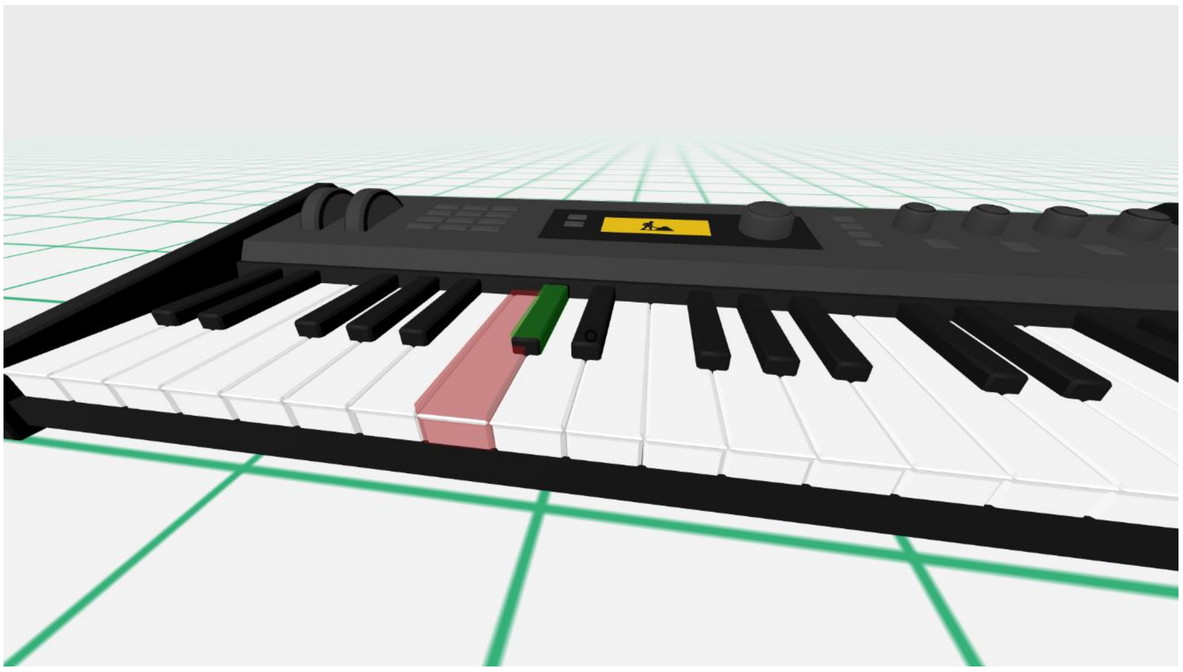
6.1 Breaking it Down

First, starting with **01_piano_empty.html**, we add the 'Piano' model right in front of us at a reasonable size. Since this piano model is just one 3D object as a whole, it cannot detect which key the player is trying to interact with. One way to do this is to attempt to recreate this piano using A-Frame's primitive boxes. Try to resize a box until it fits a piano key, or use the below code:



```
<!-- Piano -->
<a-entity gltf-model="#piano-gltf" position="0 0.25 -2.2" rotation="-90 180 0"
          scale=".01 .01 .01"></a-entity>

<a-entity position="-0.7 0.25 -2.2" scale=".2 .2 .2">
  <!--White Key-->
  <a-box id="C4" depth=4 width=1 height=1 color=#A11 opacity=0.5
         position="0 0 1.6"></a-box>
  <!--Black Key-->
  <a-box id="Db4" depth=2.2 width=0.5 height=1.4 color=#1A1 opacity=0.5
         position="0.5 0 0.8"></a-box>
</a-entity>
```

This should look like this:

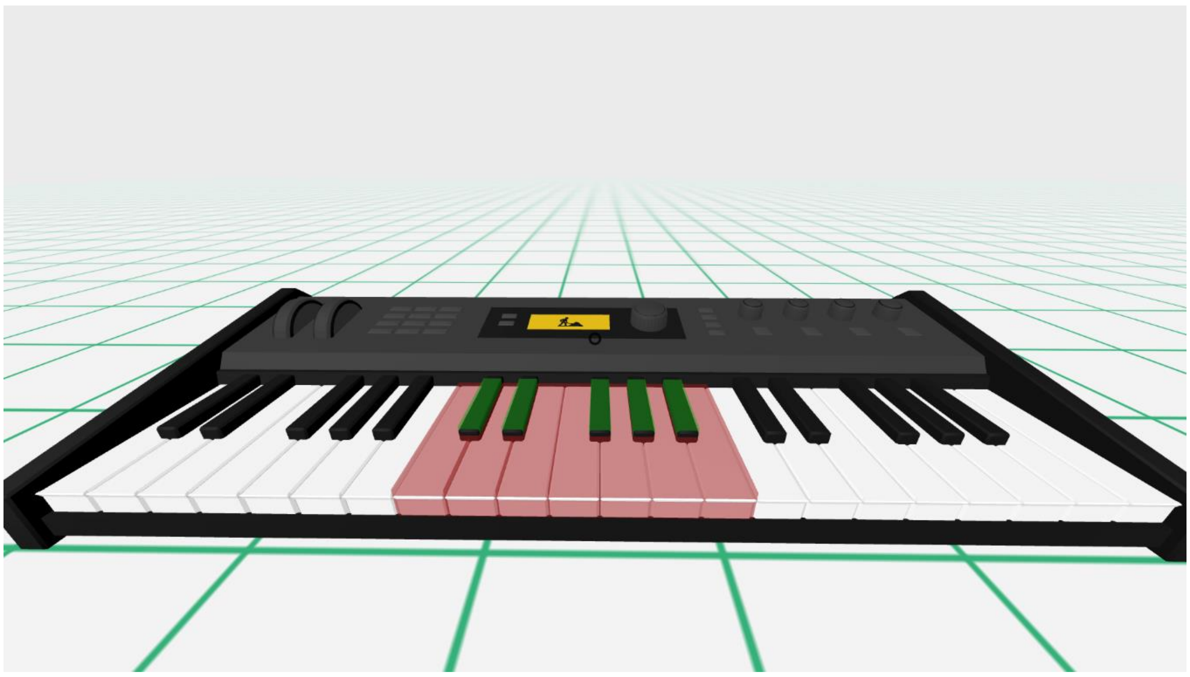


Since we are going to be using many of the same models, we should use mixins for this instead:

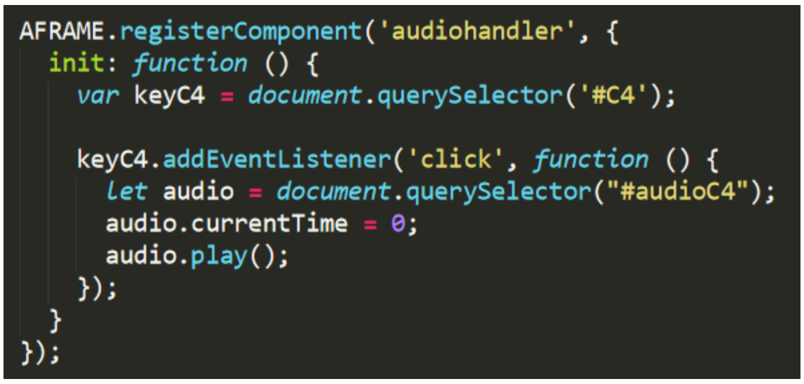
```html
<a-mixin id="whitekey" geometry="primitive: box; depth: 4; width: 1; height: 1"
         material="color: #A11; opacity: 0.5"></a-mixin>
<a-mixin id="blackkey" geometry="primitive: box; depth: 2.2; width: .5; height: 1.4"
         material="color: #1A1; opacity: 0.5"></a-mixin>
```

```html
<!-- Piano -->
<a-entity gltf-model="#piano-gltf" position="0 0.25 -2.2" rotation="-90 180 0"
          scale=".01 .01 .01"></a-entity>
<a-entity position="-0.7 0.25 -2.2" scale=".2 .2 .2">
  <a-entity mixin="whitekey" id="C4" position="0 0 1.6"></a-entity>
  <a-entity mixin="blackkey" id="Db4" position="0.5 0 0.8"></a-entity>
</a-entity>
```

Repeat this for all the keys in one octave. See **02_piano_keyC.html** for how it should look like:

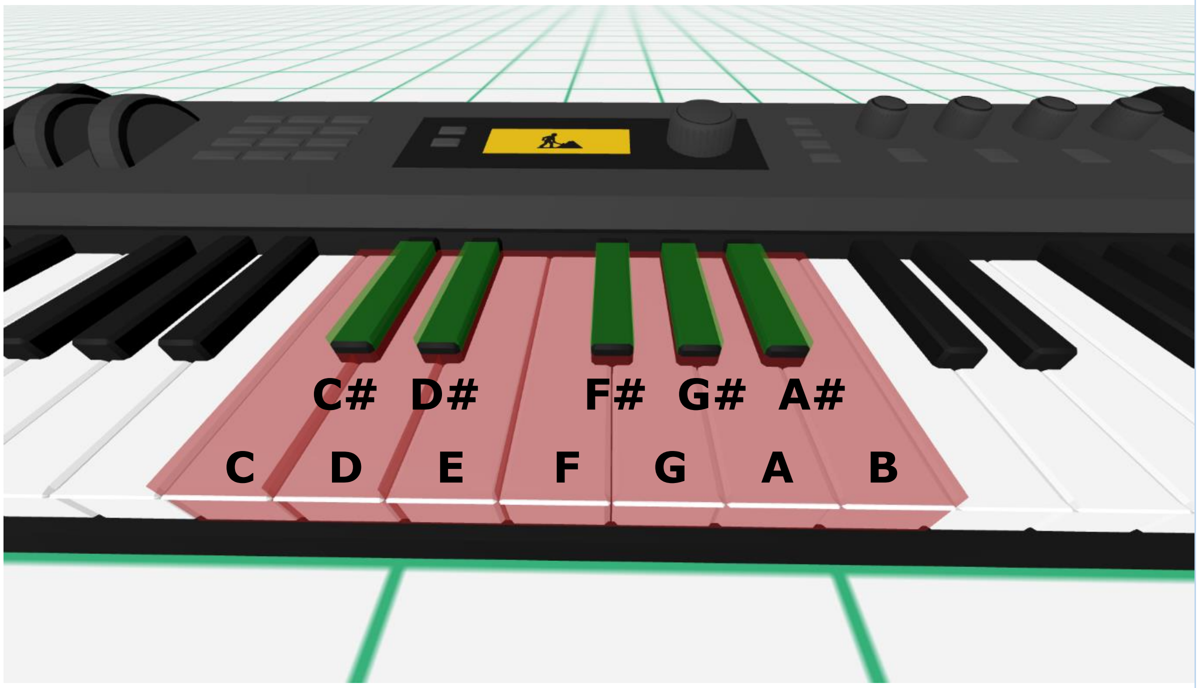


Now, to allow the boxes to detect our mouse-clicks, we use the *addEventListener()* function. A **pianoC.js** file with the above code has already been created for you in the local **/js** folder.



```
AFRAME.registerComponent('audiohandler', {
  init: function () {
    var keyC4 = document.querySelector('#C4');

    keyC4.addEventListener('click', function () {
      let audio = document.querySelector("#audioC4");
      audio.currentTime = 0;
      audio.play();
    });
  }
});
```

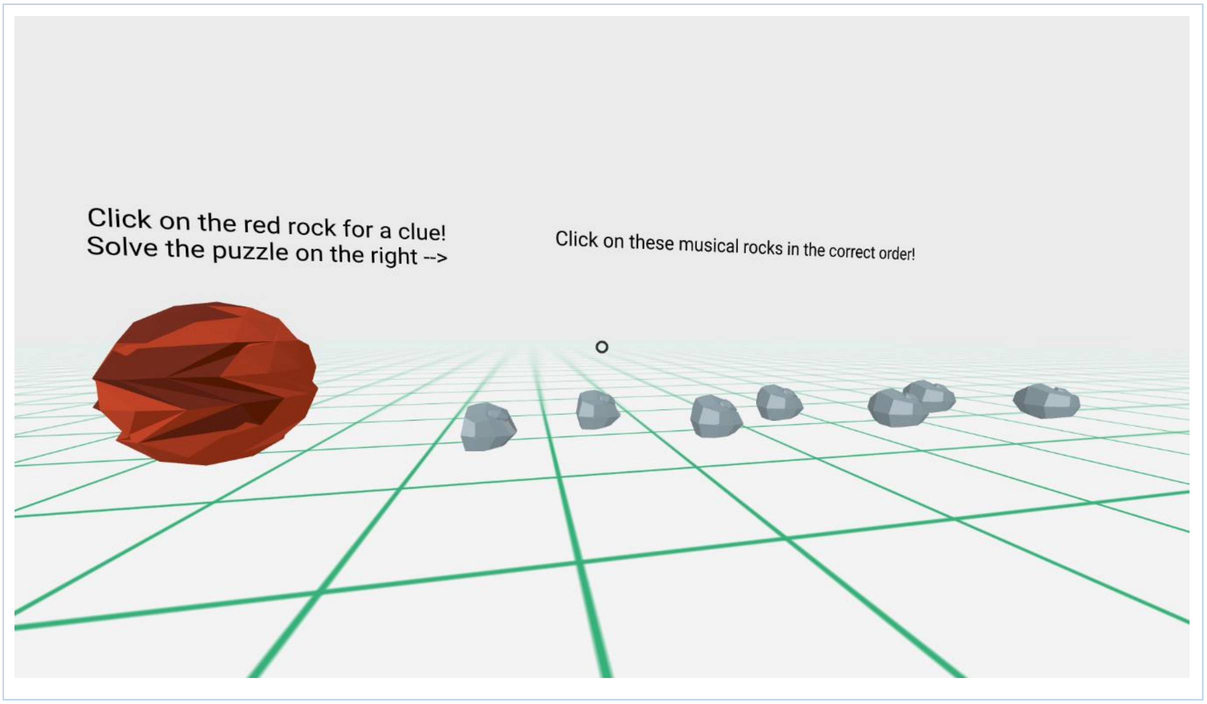
For this project, the audio assets were downloaded from University of Iowa Electronic Music Studios. Since **pianoC.js** has already been attached under the *<head>* section of **02_piano_keyC.html**, a note should play when the C key is clicked:



For all the keys to work, you'll have to repeat the above steps for every single key. This has been done for you in **03_piano_key.html** and its accompanying **js/piano.js**. Now you've learnt how to create an interactive instrument in VR!

6.2 Extending the Project: A Music-Based Puzzle

With what you've learnt in the above section, you can easily make a music-based interactive puzzle as well. In this puzzle, the player is supposed to listen to a series of notes, then click on the rocks in order of the notes played. The level layout has been done for you in **04_piano_rocks.html**.

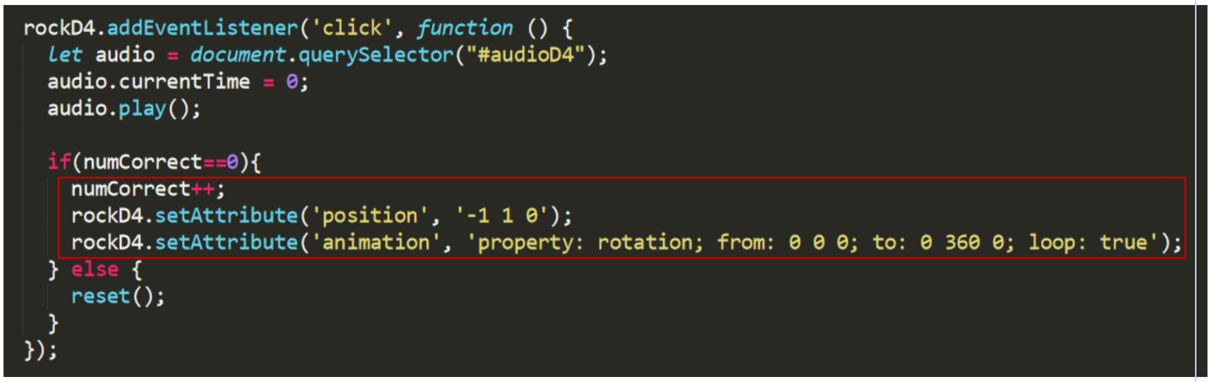

Click on the red rock for a clue!
Solve the puzzle on the right -->
Click on these musical rocks in the correct order!

Like above with the piano keys, we first make each rock play a different note. The red rock should in the middle should play **Piano.ff.Clue.mp3**, which plays the notes **D F B A G** in that order—this is the series of notes that the player must click on correctly to solve the puzzle. This has been done for you in **rocks.js**.

Next, we then ‘code’ the logic behind this puzzle. There are many different ways this can be done, but here is a simple example. For every correct rock clicked, the following lines need to be executed:

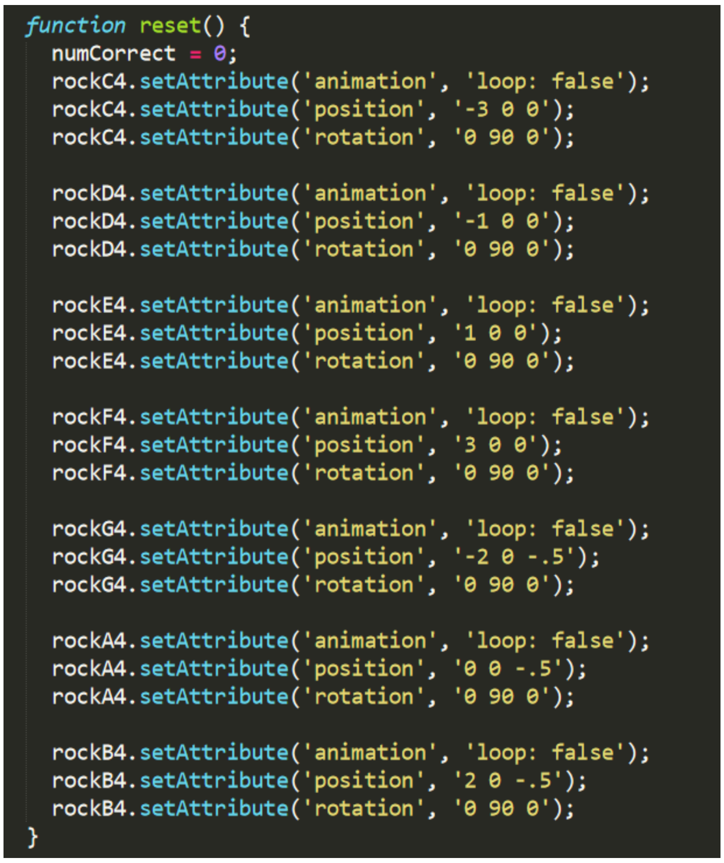
- Increase *numCorrect* by one to track the number of correct rocks clicked.
- Increase the height of the rock to indicate that it is correct.
- Animate the rock to make it spin and look like its floating.

The *numCorrect* variable is used to keep track of which rock should be clicked next. The above steps for RockD4 should only activate when it is the first rock clicked (when *numCorrect=0*). For RockF4, numCorrect should be 1, and so on for the B A and G rocks.



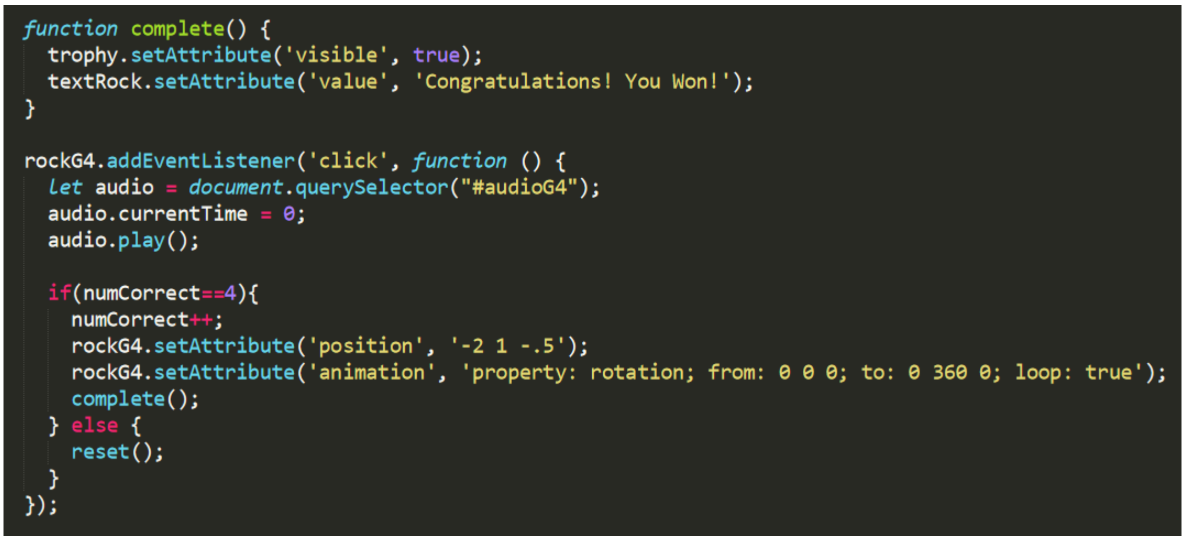
```
rockD4.addEventListener('click', function () {
  let audio = document.querySelector("#audioD4");
  audio.currentTime = 0;
  audio.play();

  if(numCorrect==0){
    numCorrect++;
    rockD4.setAttribute('position', '-1 1 0');
    rockD4.setAttribute('animation', 'property: rotation; from: 0 0 0; to: 0 360 0; loop: true');
  } else {
    reset();
  }
});
```

Additionally, if a wrong rock is clicked, all rocks should be reset to their original position. This can be done by defining a new reset() function:



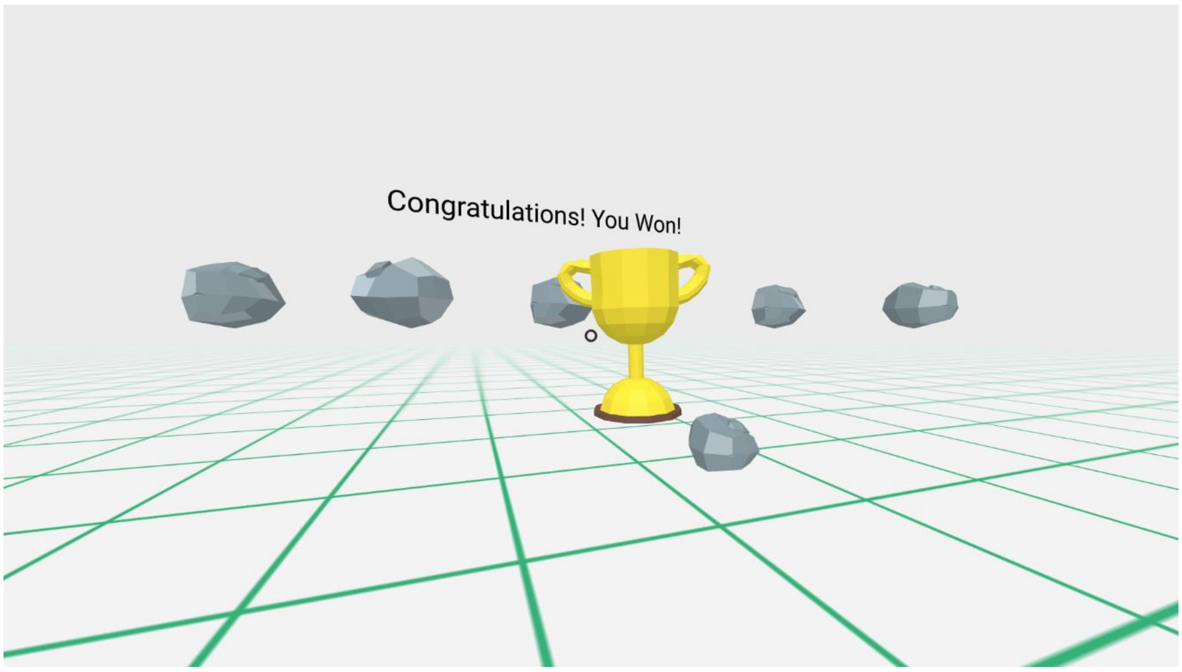
```
function reset() {
  numCorrect = 0;
  rockC4.setAttribute('animation', 'loop: false');
  rockC4.setAttribute('position', '-3 0 0');
  rockC4.setAttribute('rotation', '0 90 0');

  rockD4.setAttribute('animation', 'loop: false');
  rockD4.setAttribute('position', '-1 0 0');
  rockD4.setAttribute('rotation', '0 90 0');

  rockE4.setAttribute('animation', 'loop: false');
  rockE4.setAttribute('position', '1 0 0');
  rockE4.setAttribute('rotation', '0 90 0');

  rockF4.setAttribute('animation', 'loop: false');
  rockF4.setAttribute('position', '3 0 0');
  rockF4.setAttribute('rotation', '0 90 0');

  rockG4.setAttribute('animation', 'loop: false');
  rockG4.setAttribute('position', '-2 0 -.5');
  rockG4.setAttribute('rotation', '0 90 0');

  rockA4.setAttribute('animation', 'loop: false');
  rockA4.setAttribute('position', '0 0 -.5');
  rockA4.setAttribute('rotation', '0 90 0');

  rockB4.setAttribute('animation', 'loop: false');
  rockB4.setAttribute('position', '2 0 -.5');
  rockB4.setAttribute('rotation', '0 90 0');
}
```

Lastly, when the last rock G4 has been clicked, you should indicate to the player that they have completed the puzzle. Make a trophy visible, and change the instructions to a congratulatory message! The below complete() function should also be run in RockG4.



```
function complete() {
  trophy.setAttribute('visible', true);
  textRock.setAttribute('value', 'Congratulations! You Won!');
}

rockG4.addEventListener('click', function () {
  let audio = document.querySelector("#audioG4");
  audio.currentTime = 0;
  audio.play();

  if(numCorrect==4){
    numCorrect++;
    rockG4.setAttribute('position', '-2 1 -.5');
    rockG4.setAttribute('animation', 'property: rotation; from: 0 0 0; to: 0 360 0; loop: true');
    complete();
  } else {
    reset();
  }
});
```

When the puzzle is complete, the below should happen. See how it works using **05_piano_final.html** and **puzzle.js**:



6.3 Other Suggestions

a. Try adding other instruments to your virtual world, or perhaps even create a band room with other percussion-based instruments like drums, cymbals, and xylophones.

b. Try making a more complicated musical puzzle than what we've done in Section 3.2.

c. Look at other types of simple point-and-click puzzles that you might want to add to your game.

7 End Note

With this tutorial, you are now ready to begin creating projects of your own! Once you have drafted your own project, the final step should be to check out the documentation on Interactions and Controllers to make your environments work using VR equipment. Have fun create new projects with your newly learnt VR making skills!

- The End -