Mobile HCI: Basic XR Interactions

Euan Freeman (euan.freeman@glasgow.ac.uk)

1. Introduction

During this lab exercise, you will begin implementing a mixed reality user interface with interactive buttons that the user can activate by 'looking at' them. This builds on the introduction to XR development exercise from the previous lab. If you have not completed the previous lab exercise, then I recommend you start that first.

Users can interact with mixed reality experiences in a variety of ways, which you'll see more of during a later unit of the course. For this lab exercise, we'll be focusing primarily on the use of **gaze input** for activating XR user interface buttons. When buttons are in a fixed real-world position, gaze input allows the user to activate them by looking at them.

Whilst gaze input can be detected using eye-tracking devices that recognise where you are looking on a screen, gaze input often just refers to the use of **head angle** or **head orientation** for estimating where the user is looking. When wearing a virtual or augmented reality headset, head orientation can be tracked by inertial motion sensors in the device. When using a handheld device for mixed reality, the inertial motion sensors can also be used to estimate head orientation, based on the assumption that the device is held in front of the user.

In this lab exercise, you will implement gaze-activated buttons that use device orientation to estimate where the user is looking. Since most of you will be doing this lab with a handheld smartphone, 'gaze' just means moving the device to look in a new direction.

We will be using **Glitch** for this lab exercise, like in the previous lab. You can run your prototypes via the web browser on your mobile device. If you don't have a compatible device, you can still test your prototypes using your desktop web browser – instead of moving a smartphone to 'look around', you can look using the mouse pointer instead by clicking and dragging.

1.1. Helpful resources

- https://aframe.io/docs/
- https://aframe.io/docs/1.5.0/introduction/interactions-and-controllers.html
- https://aframe.io/docs/1.5.0/components/cursor.html

1.2. Intended Learning Outcomes

- **ILO4:** develop and evaluate prototypes of mobile interactive systems using a variety of prototyping methods and evaluation techniques.
- **ILO5:** discuss cutting edge developments in mobile human-computer interaction, such as context-aware systems, sensor-based interaction, location-based interaction, and mixed reality.

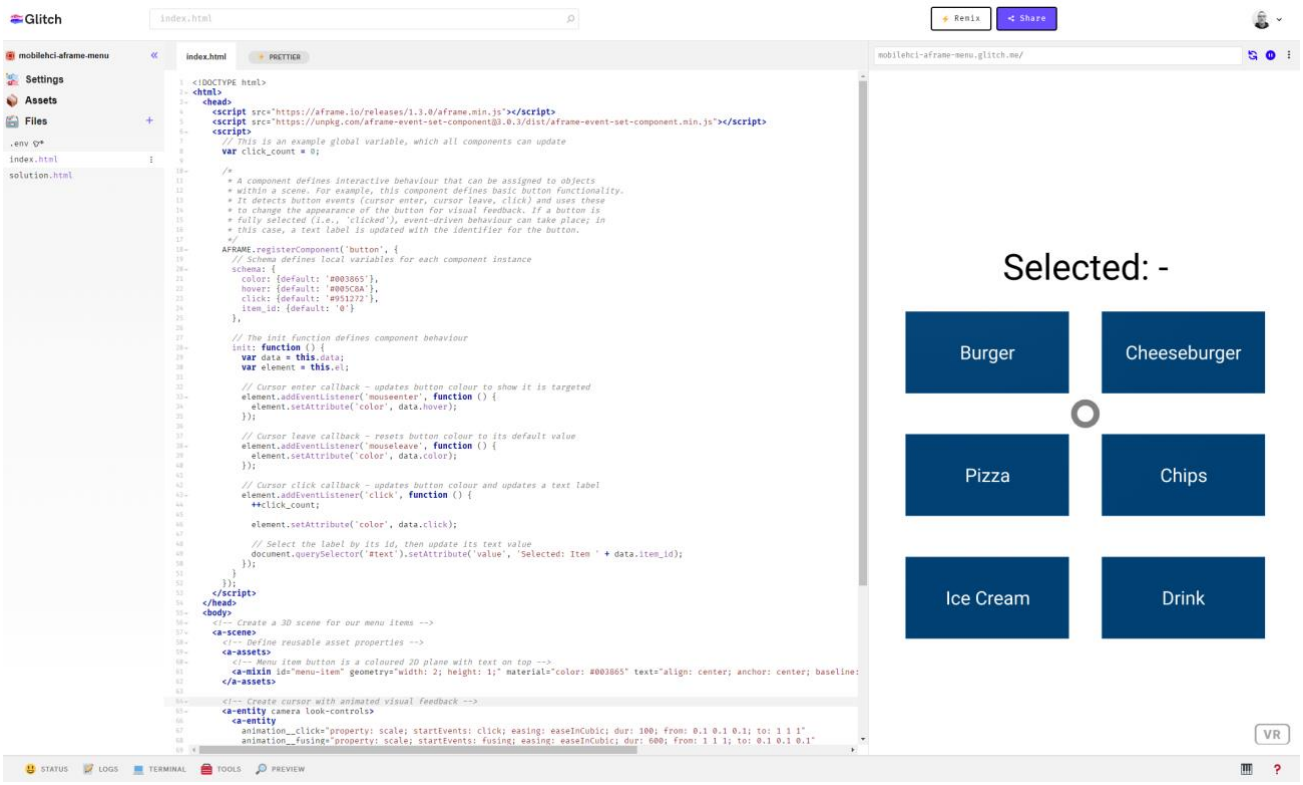
2. Lab Exercise

2.1. Introduction

We are going to be using the **Glitch** development environment (https://glitch.com/) for this lab exercise, just like last week. You do not require an account to use Glitch, but we recommend you create one so you can keep working on your projects at a later date, since temporary projects are deleted after five days.

2.2. Access the Starter Project

Go to https://glitch.com/edit/#!/mobilehci-aframe-menu to access the starter project for this lab exercise. Press the Preview button in the bottom toolbar and open the preview pane. This page should now look something like the following screenshot. There are three main panels in this view: a list of project files on the left, a file editor in the middle, and a live preview on the right.

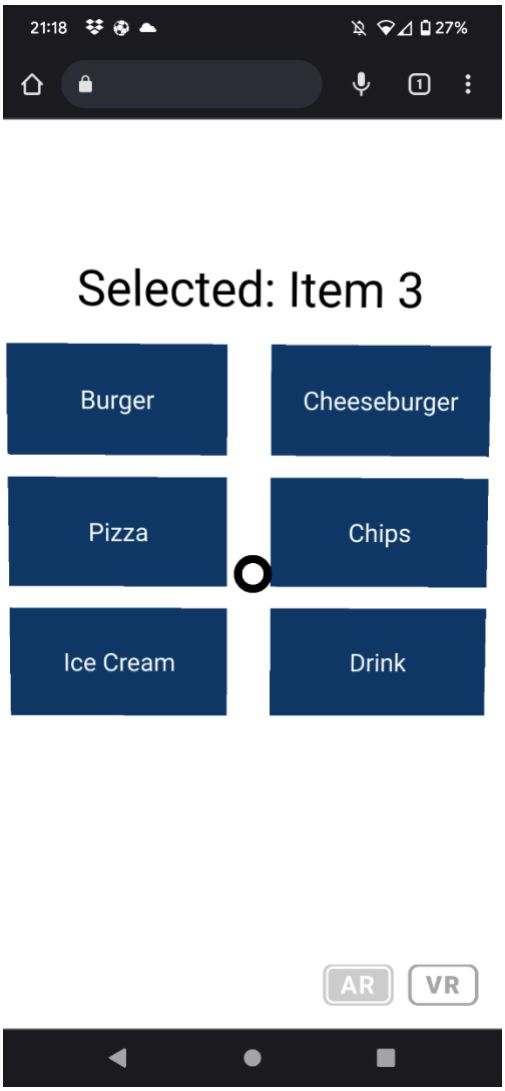


In the preview panel, you should see a menu with six blue buttons, a text label above them, and a black circular cursor in the centre. The black circular cursor is positioned in the centre of the display, representing the assumed direction the user's head is facing. This will stay fixed in the centre of the display, so that it always moves with the device; in contrast, the buttons will move such that they remain in their fixed real world positions.

2.3. Remix the Starter Project

Press the Remix button in the top toolbar to create an editable copy of the starter project. This clones the project and creates a unique instance for you to edit. Open the Preview pane again and note the unique URL shown at the top. Using your mobile device, open a web browser and navigate

to that unique URL – you need to include **https://** at the beginning of the URL because most browsers only allow access to XR features via a secure https (not http) connection. Remember to give the website permission to access motion sensor data, if prompted.



When viewing your project via a mobile device, the content will be anchored in a real-world position relative to your device - normally at head height, so remember to look 'up' if you can't see anything on screen.

When you move your device, notice the virtual content also moves so that it stays in its original position and orientation in the real world. However, the black circular cursor will always remain in the centre of the screen.

If you are using your desktop to preview your prototype, you can click-and-drag with the mouse pointer to move the camera, which simulates 'looking around'. If you open the preview on your mobile device, you can 'look around' by moving the device.

Depending on browser compatibility, you may see buttons in your mobile web browser labelled AR and VR (like in the screenshot on the left). If these buttons are visible, try tapping the AR button. This should enable a pass-through feed from the camera, so that virtual content appears to be rendered over the real world. You may need to give the website permission to access the camera. Unfortunately, most iOS browsers don't currently support AR mode and some Android devices don't display camera images despite having the AR button visible.

2.4. Explore the Starter Project Code

If not already visible, open index.html in the Glitch file editor by selecting it from the project files panel. This starter project is more complex than the basic introduction from the previous week and it introduces new A-Frame programming concepts, so your first task in this lab exercise is to form a better understanding of the code.

Mixin Objects: Reusable Asset Properties

During last week's lab exercise, you created virtual objects by adding them to the scene and manually specifying their individual properties - e.g., position, rotation, scale, color. As shown by the screenshots above, there are six menu items with similar appearance due to shared properties (e.g., all have the same background colour, height, and width). We want to avoid duplicating code

because it's just generally good software engineering practice, and so we want to find a way to share these common properties between objects in our scene.

In A-Frame, you can define reusable assets with shared properties using **mixin** objects ("mix in"). On Line 61 we define a mixin object with the identifier "menu-item". This mixin object specifies several properties that can be applied to other objects in the scene. In this instance, we have specified the dimensions (via the *geometry* property), appearance (via the *material* property), and text appearance (via the *text* property). We can now create objects in our scene that *inherit* these properties by referencing the mixin identifier "menu-item".

On Line 77 to 82 we define six objects corresponding to our menu items. These objects reference the "menu-item" mixin identifier via the *mixin* property; this tells A-Frame that these objects should inherit properties from this mixin asset - thus, they take on the size, colour, etc, without us having to explicitly specify them.

Property inheritance works similarly to inheritance in object-oriented programming. The inheriting object can override partial properties from its parent; you can see an example of this with the *text* property, which is declared by both the **mixin** and the **a-plane** objects.

For more information about mixin objects, you may wish to refer to the A-Frame mixin documentation: https://aframe.io/docs/1.5.0/core/mixins.html

Cameras and Fixed-Position Content

When you create a new A-Frame scene, the framework implicitly adds a **camera** which determines the relationship between the screen and virtual content. When no camera objects are explicitly specified in your code, A-Frame adds one with default camera properties. However, you can add your own camera object and change its behaviour, if desired. In this project, we have added a **camera** to our scene on Line 65.

Normally when you add objects to a scene, you want them to maintain position in real world space. However, there may be some user interface elements that you want to always remain visible in the same region of the screen, regardless of where the user looks (e.g., for heads-up display overlays). In this particular project, we always want the circular cursor to remain in the centre of the screen, so that when the user moves their mobile device, the cursor moves with the device and always remains in the middle.

One approach for implementing this desired behaviour is to define fixed-position content as a child element of the camera object, rather than as a child of the scene. On Line 66 to 70 we have defined a *ring* entity as a child of the camera, so that it always remains visible in the same place on screen when the camera moves. Ignore the animation properties - these control the ring's appearance during button selection.

Note that 'child element' means nesting an element inside another element. This can be done with almost any A-Frame object. When you nest a child object inside a parent object, the child's

properties will be *relative to the parent* properties, rather than absolute. So, for example, the position of a child object will place an object relative to its parent object position, not at an absolute position within your A-Frame scene.

For more information about fixed-position content, refer to the Extra Tasks section of Lab 1, which was about creating a mock heads-up display. You may also wish to refer to the A-Frame camera documentation: https://aframe.io/docs/1.5.0/components/camera.html

Adding Interactive Behaviour

Declaratively adding objects to an A-Frame scene allows you to compose a 3D mixed reality environment with static objects. In this lab exercise we are particularly interested in adding *interactivity*, so that the web app can respond to users' actions. This requires adding some Javascript programming to provide object behaviour and functionality.

Earlier in this lab handout we discussed **mixin** objects, a framework that can be used to provide shared properties for objects in your scene. To add dynamic behaviour, we can leverage a similar concept called **components**. Components offer a framework for modifying objects in a scene, e.g., to add functionality in response to user input.

Components are flexible and complex, supporting a range of behaviours and features in an A-Frame app. For this lab exercise, we want to use them to simply add interactive behaviour to the buttons in our scene. Components are defined in Javascript code, by providing a **name** and **definition**. Within the definition, you can override several *lifecycle* methods to implement dynamic behaviour that is executed at various points in the app lifecycle (e.g., upon initialisation, each time the scene is rendered, when updates are explicitly requested, etc).

In the starter project implementation, we declare a "button" component beginning on Line 18, as part of a snippet of Javascript code. Within the component definition, we have declared a schema of local variables (Line 20) and have overridden the **init** lifecycle method, which is executed only once, upon initialisation of the app.

Within the **init** method, we have added three event listeners for our button component:

- The **mouseenter** event is triggered when the cursor (in this case, the circular cursor in the centre of the screen) collides with the component. The basic implementation changes the colour to a variable defined in our schema.
- The **mouseleave** event is triggered when the cursor no longer collides with the component. The basic implementation changes the colour back to its original value.
- The **click** event is triggered when an input selection event is complete. There are various ways of implementing a 'click' in A-Frame, but here we have used 'dwell' - i.e., selection by keeping the cursor over the component for a certain period of time. The basic implementation increments a selection counter, changes the colour of the button component, and updates the value of the text label in the A-Frame scene.

Now look at the declaration of the menu items in the scene. Note that these declarations include a *button* property, which is the name of our component. By including this property name, we are informing A-Frame that this object is an instance of the component named "button".

Note also that we have specified a value for the *button* property ("item_id: 1", etc). Named values for components refer to variables in the component *schema* - this provides us with a mechanism for passing parameters to our component code, similar to the use of *constructors* in imperative programming. In our implementation, we specify the "item_id" value. We can access that from our component definition, e.g., Line 49 uses the item_id value to provide feedback about which button is selected when a click event occurs.

Read through the component code and try to form your own mental model of how this operates. Run the app in the preview pane, or on your own mobile device, and try to follow the code in your head whilst you use the app.

For simplicity, we're going to avoid going into too much detail about A-Frame input methods for now - but we recommend you do some independent reading about this in your own time. There are some links on the first page of this handout which are relevant to this. You may also wish to also refer to the A-Frame component documentation on components:

- https://aframe.io/docs/1.5.0/introduction/writing-a-component.html
- https://aframe.io/docs/1.5.0/core/component.html

Summary

By this point in the lab exercise, you've mostly just been reading the handout and starter project code and trying to figure out what is happening. We don't expect you to fully understand things at this stage, but we hope you are forming a rough idea about the high-level architecture of interactive A-Frame apps:

- Content can be declaratively added to an A-Frame scene via HTML, e.g., shapes, text.
- Fixed-position content can be created by adding as a child of a **camera** element.
- Elements can inherit shared properties from **mixin** objects.
- Dynamic behaviour and functionality can be provided via **component** definitions.

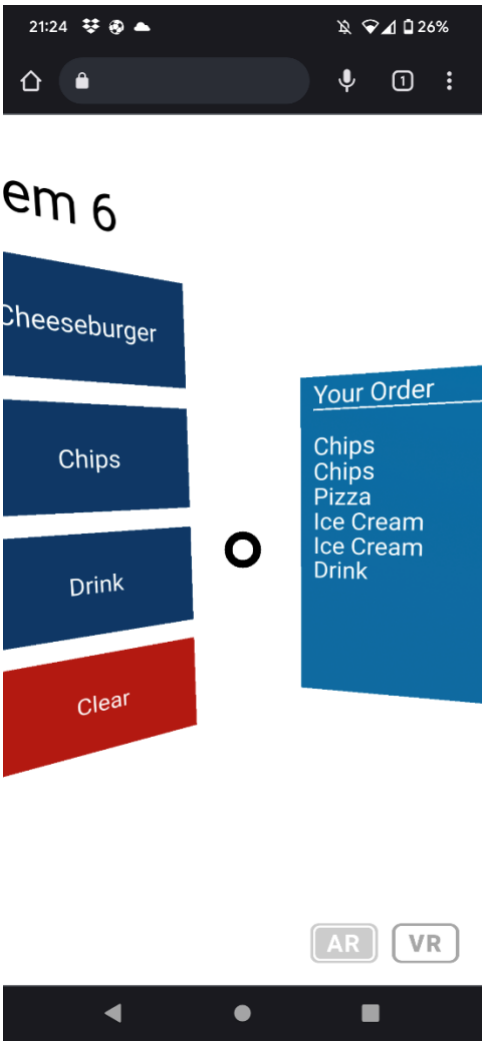
For the remainder of this lab exercise (or in your own time later), we are going to extend this basic implementation with extra functionality. The aim of this lab exercise is to help you develop simple prototyping skills for interactive mixed reality web apps.

2.5. Adapting the User Interface

Task 1 Begin by adding two new buttons to the menu, labelled "Checkout" and "Clear" respectively. You can place these anywhere you want by specifying the *position* property. When declaring these new objects, remember to include the appropriate **mixin** and **component** information.

Task 2 Change the colour of the "Checkout" button to green (e.g., #006630) and change the colour of the "Clear" button to red (e.g., #B30C00). Recall from last week that you can change the colour by specifying the *color* property of the object declaration. Also recall from earlier that an inherited property (i.e., the blue button colour) can be overridden by the declared object's properties.

Deploy and test your prototype on your mobile device, or in your web browser. Try using your new buttons. Do they work as intended? Try to fix any issues with the two new buttons. *Hint: schema*.



Task 3 Now create a new information panel to present a list of selected food items; place it somewhere close to the menu. The screenshot on the left shows an example of what this might look like, but the goal here is simply to implement something that works, appearance is not important.

This task will require a bit of trial-and-error and tweaking of position, scale, etc; however, the starter project should have sufficient code snippets that you can use as a starting point for (i) creating a text panel and (ii) updating a list of text items each time a food item is selected.

One possible approach would be to use the menu-item properties as a starting point for a new text element. If you use this approach, the text properties of the mixin will likely need to be changed. Note that *wrap-count* determines the text size if using this approach.

Alternatively, you could add a new <a-plane> object with an <a-text> object as a child element. Text can be positioned within the plane using the *position* property and can be resized using the *scale* property. Note that you will need to specify a z coordinate for the text position so that it gets rendered 'in front of' the plane, otherwise the text and plane will occupy the same z-coordinate and won't render clearly.

The desired behaviour is that when the user selects an item from the menu, it gets appended to the end of the order list. Note that you can concatenate text in Javascript with the + operator and that the newline character (\n) is used to create a new line. You should also implement the functionality for the "Clear" button but can ignore the "Checkout" button.

Having trouble with this task? First, just focus on adding a new text widget with some dummy text. Play around with its properties to create an appearance you are satisfied with. Next, you'll need to start adapting the button component code so that your information panel gets updated with appropriate information. There is already some existing code for updating the "Selected Item" label, which you can adapt.

2.6. Summary

You've now completed the main tasks for this lab exercise, which was an introduction to some advanced A-Frame features for adding interactive behaviour. You should now have a foundational understanding of mixin objects and components, which are useful when creating more complex interactive apps using A-Frame. Mixin objects help simplify code when you begin creating more objects (especially objects that are similar in appearance). Components are necessary for adding dynamic behaviours in response to user actions.

If you completed the main lab tasks, you should now have a somewhat functional menu that can be used to add items to an order and clear the list. There are some additional tasks listed below, which you are encouraged to do at the end of the lab (if there is enough time) or in your own time. These will help develop your A-Frame prototyping skills.

2.7. Extra Tasks

Task 4 Assuming you have not changed the position and size of the original menu items, then you've probably noticed that the current implementation is not very ergonomic, from an interaction perspective. The menu takes up most of the smartphone screen and requires quite a lot of device movement to select items placed at the top of the menu – if this was being viewed through a mixed reality headset then the user would need to make a lot of up/down head movements. You probably also had to position your order list off-screen since there wasn't much room left within the field-of-view.

For this additional task, you should try to improve the ergonomics of the user interface design, so that more information can fit on screen at once and so that items can be more easily selected. There are several approaches you could take to this, e.g., changing the position and/or scale of the user interface items, and rotating user interface elements so that they face the user. There is a lot of scope for creativity here so experiment with different designs and try to find something you consider to be more easily usable.

Task 5 Add item prices. This will require (i) updating the menu buttons, (ii) implementing a list of item prices in the component code, (iii) adding a global variable to the component code to maintain a running total of selected item prices, and (iv) updating the information panel to display individual prices and/or a total price.

Task 6 Add a heads-up display (HUD) overlay that presents a count of selected items and price, e.g., "4 items, 8.00". Note that A-Frame doesn't like the £ currency symbol, so just stick with numerical characters. For advice, refer to the earlier discussion about **camera** objects and also check out the extra tasks in Lab 1.

Task 7 Add a pop-up text panel which is similar to the information panel, but only appears when the user selects the "Checkout" button. Note you will need to use the *visible* property (value of true or false) to ensure this only displays at an appropriate time.

The following screenshots (on the next page) show examples of what the user interface might look like after these additional tasks - though my design is fairly boring, there's a lot of scope for creating something that is both more usable and more attractive.