Multi-Modal, Adaptive UI Framework

Contents

I UX Process

The task of the UI team was to research the selection of modalities and elements and develop an approach to enable successful and efficient communication between the pilot, his team and the Gaia system.

At the beginning of the project, we focused on touch-based interactions and focused on solutions that offered easy accessibility. The challenge was to balance good element reachability with clear and visible positioning.

1 Reachability Heatmap

To familiarize ourselves with the subject and create suitable concepts, we measured the existing screen-setup. We then created a reachability heatmap for touch interaction with the left hand, considering the reachability of each screen and the time required for the interaction (see figure 1).

The heatmap illustrates the positioning of the screens for system interaction. The lower three rectangles represent the three touchscreens, divided into four areas by color. Area 1 offers very good reachability. Area 2 offers good reachability, but requires more time and effort. Area 3 is already difficult to reach, requiring significant arm bending or stretching and possibly leaning forward. Area 4 is out of range and should only be used for information display due to the high physical effort required for interaction.

From the reachability heatmap, we deduced that essential interactions should primarily be placed in Area 1 and, if necessary, in Area 2. Placing interactive elements in Area 4 requires high muscle activity, leading to increased muscle fatigue.

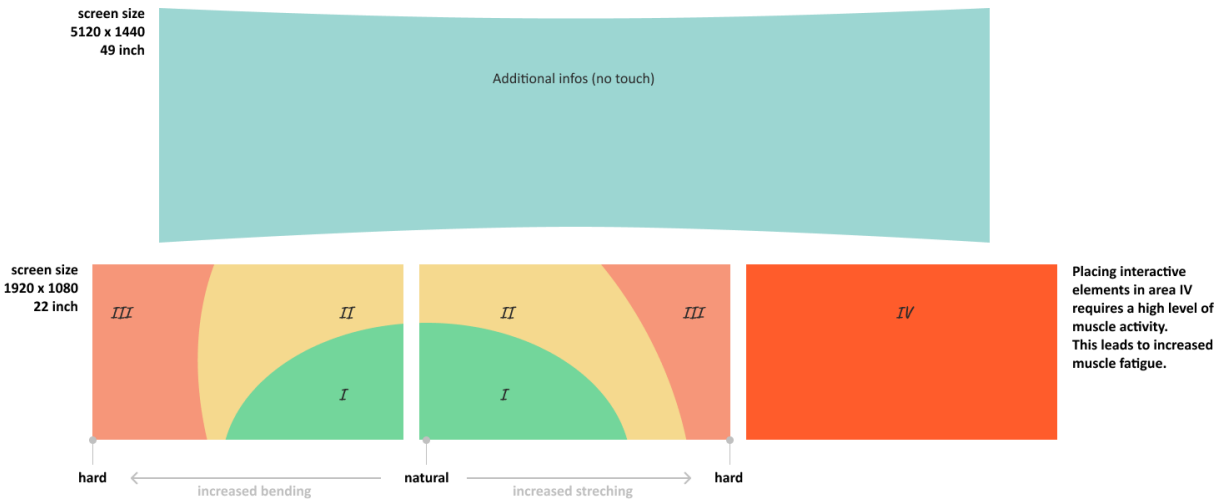


Figure 1: Reachability heatmap for right-handed piloting and left-handed interaction with the interface

The reachability heatmap was created for touch interaction as we assumed this would be our input method. However, as the project progressed, we focused on gaze interaction. Following this, we researched existing concepts and different interaction possibilities in a cockpit.

2 Research

2.1 Dwell Buttons for Gaze-Only Interaction

Findings suggest that placing the button label outside of buttons instead of inside (see figure 2) significantly reduces the Midas Touch problem and improves accuracy [1]. Recommendations include using large buttons and short dwell times when content is outside clickable areas, and using large buttons with long dwell times when content is on buttons. Further exploration is important, such as using visual anchor points like a dot in the center of buttons to enhance accuracy and ease of use. Dwell buttons, which are activated when a user's gaze remains fixed on the button for a predetermined amount of time (typically between 200 to 500 milliseconds), are not recommended. Therefore we decided to use a multi-modal interaction with gaze and a thumbstick.

Figure 2: Buttons with inside and outside label

2.2 Spatially Directed Visual Warnings

A study by Dettmann and Bullinger showed that spatially directed visual warnings are helping to shift the attention of the user towards relevant objects. Multi-modal warnings with spatially directed visual and auditory warnings are increasing the performance even more [2]. Therefore, in our project we want to use a combination of visual guiding of the gaze and auditory alerts and warning messages.

2.3 Design Recommendations for Touch Screen Based User Interfaces

In a further study, Michael Domhardt outlined design guidelines that impact successful human-computer interaction [3]. One key recommendation is the importance of providing auditory or visual feedback for interactions. A simple sound can already indicate that the interaction has been successfully completed.

The goal of each interaction is to achieve high-quality performance that is quick, precise, and requires minimal effort. Therefore, it is also crucial to reduce muscle fatigue. One potential issue is that

not every area on the touchscreen is easily reachable. To address this, we developed a reachability heatmap (see section 1 to identify areas where interactive elements can be placed within easy reach.

Another concern is fatigue resulting from prolonged interaction. Studies have shown that the optimal tilt angle for touchscreens ranges from 30° to 55°. This range has a direct positive impact on reducing perceived eye, arm, and finger fatigue.

Additionally, to prevent "gorilla arm syndrome", it is recommended to provide arm support.

3 UX Concept

3.1 First Sketches

To develop effective concepts, we began with initial sketches to support the idea generation process. We collaboratively discussed the viability and impact of each sketch, iteratively refining them.

The first sketch (see figure 3) illustrates the heads-up display with a possible placement of information. The three interaction screens are positioned below. The left screen demonstrates the placement of the new messages and a message list. The middle screen has a full-screen minimap with possible threats. The right-hand screen is only used to visualize information such as control panels. "Attention getters" are placed above the screens to attract the pilot's attention and guide the pilot's gaze to the correct position on the screens.

Figure 4 demonstrates the change in the elements on the screens for a message with increasing priority. Figure 5 shows a possible layout for the "Escalation Mode".

Figure 6 demonstrates a different approach, in which an attempt was made to place the important messages directly above the minimap in order to maintain the connection between them and the corresponding icon on the minimap.

Figure 7 shows the first attempt to divide the screens into functional areas. The left area contains a clear list of all messages. The middle area displays an interactive map and also includes a non-interactive information area above. The right screen contains additional gauges and can display further non-interactive information.

The left screen area could provide various methods to sort the list of messages, depending on the pilot's requirements and the specific messages being searched for. In an optimal system, the pilot would have the ability to sort messages by priority, latest, unread, read, and those requiring action (figure 8).

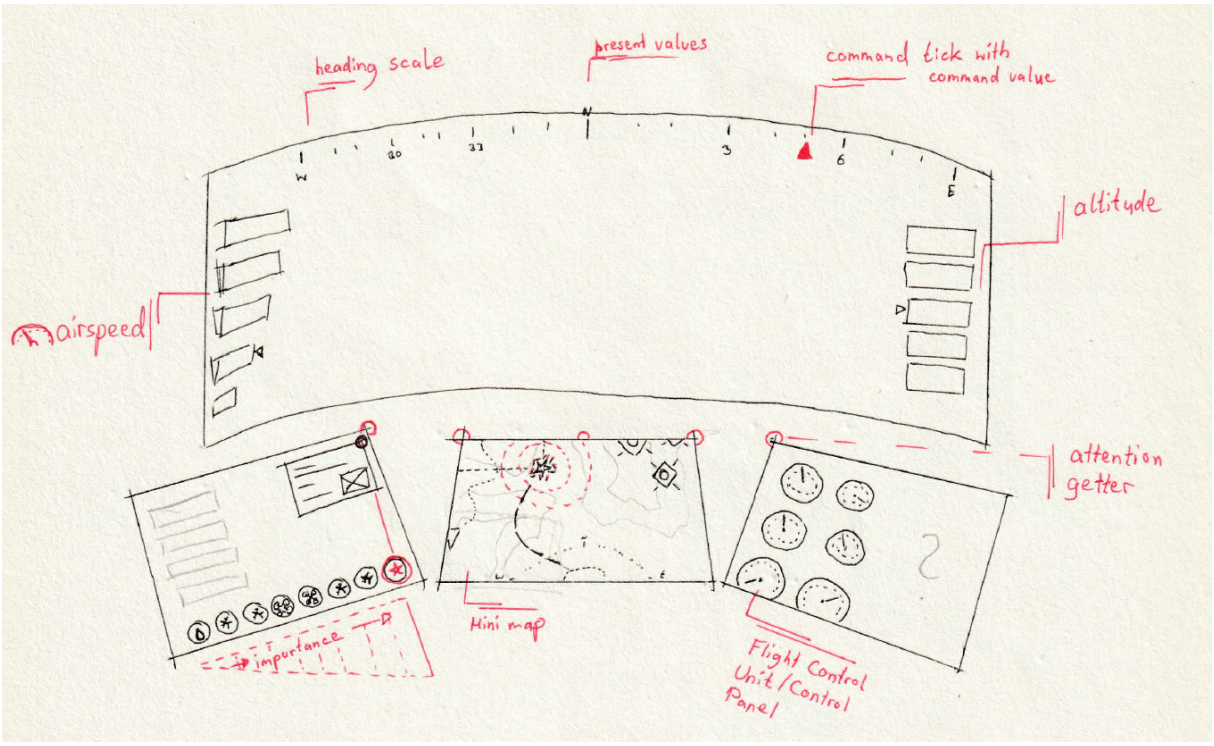


Figure 3: First Sketch – Low Priority

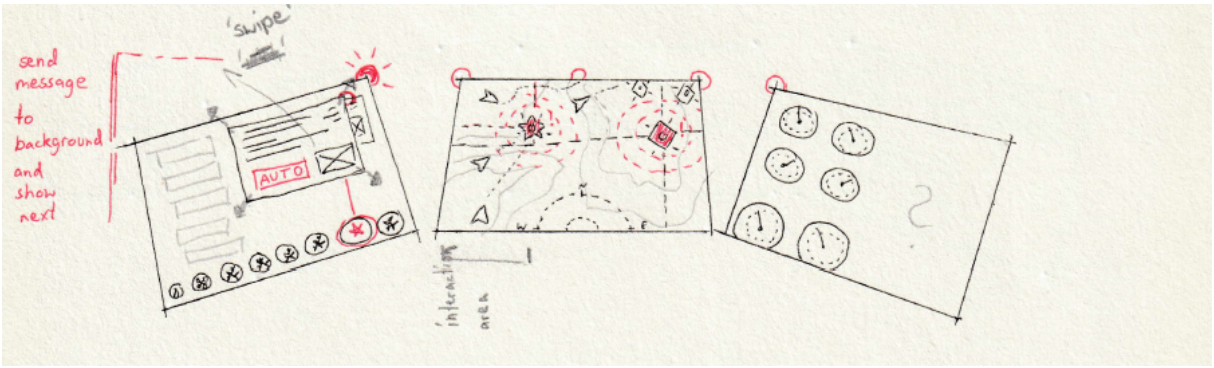


Figure 4: First Sketch – Increased Priority

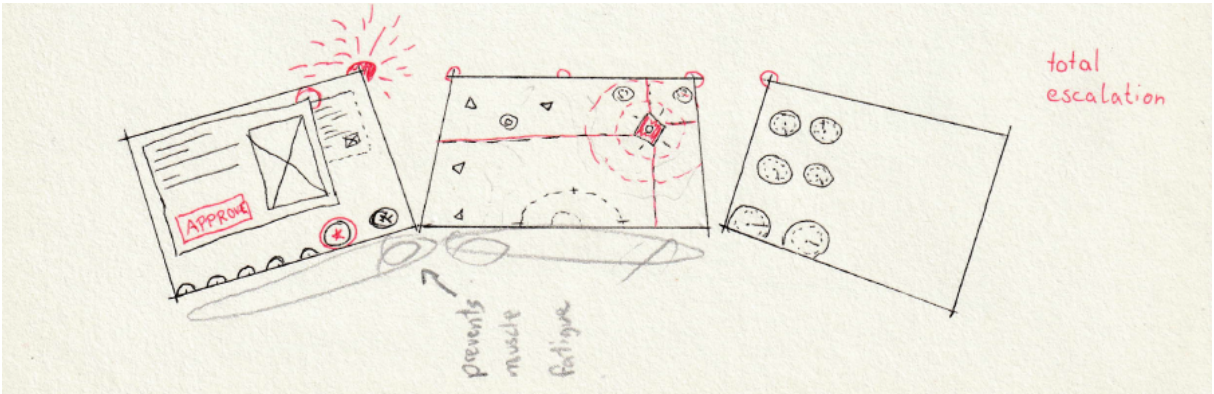


Figure 5: First Sketch – Escalation Mode

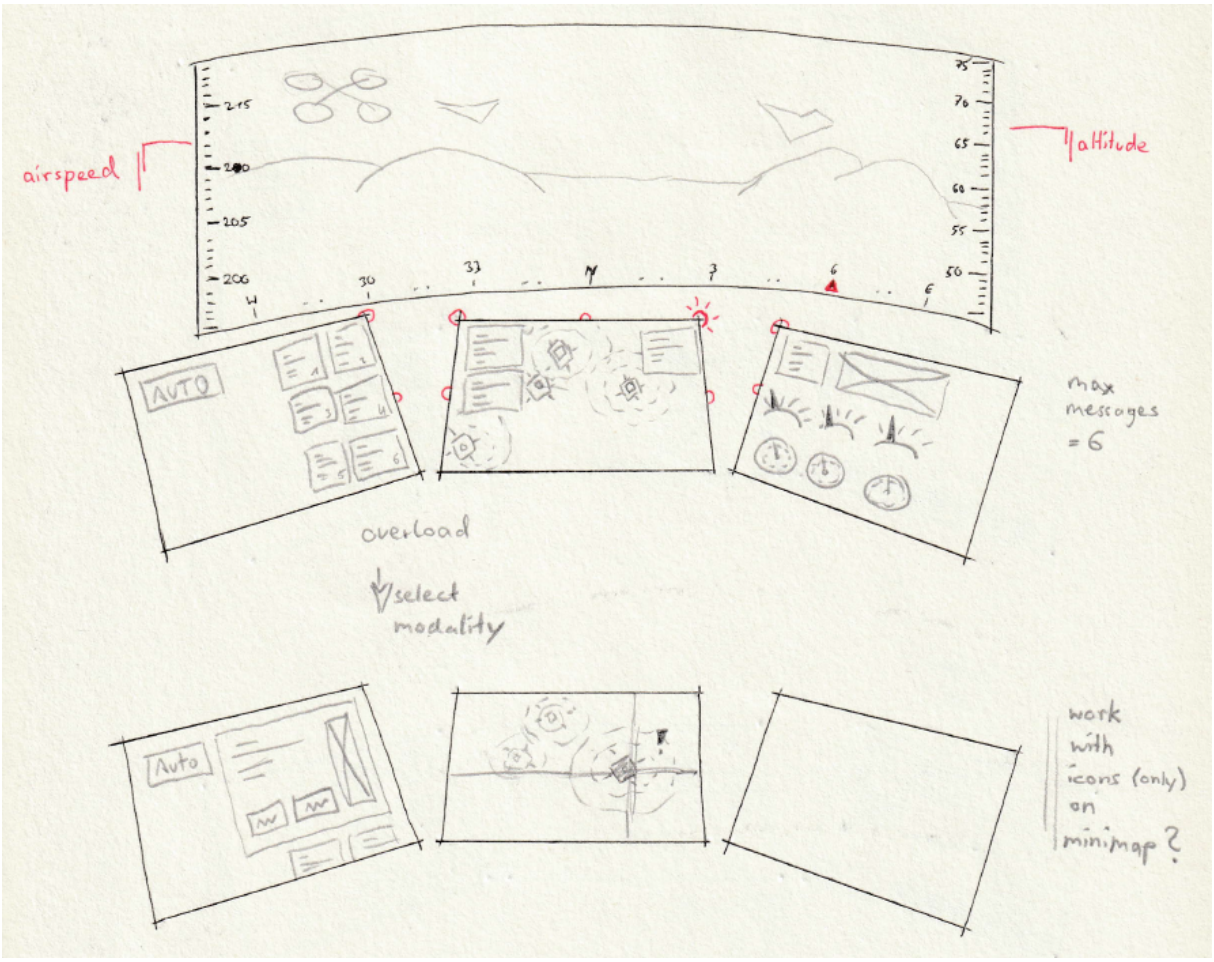


Figure 6: Sketch – Alternative

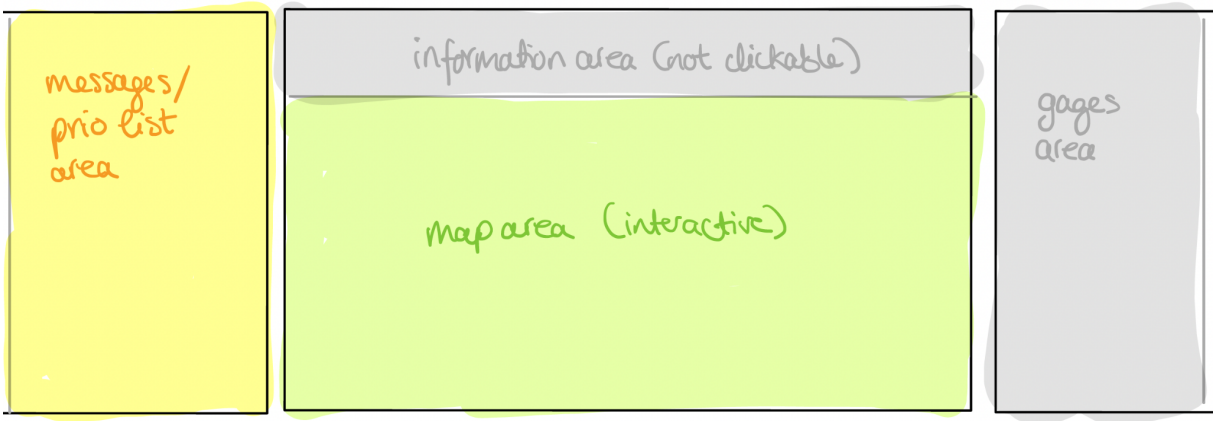


Figure 7: Functional Areas

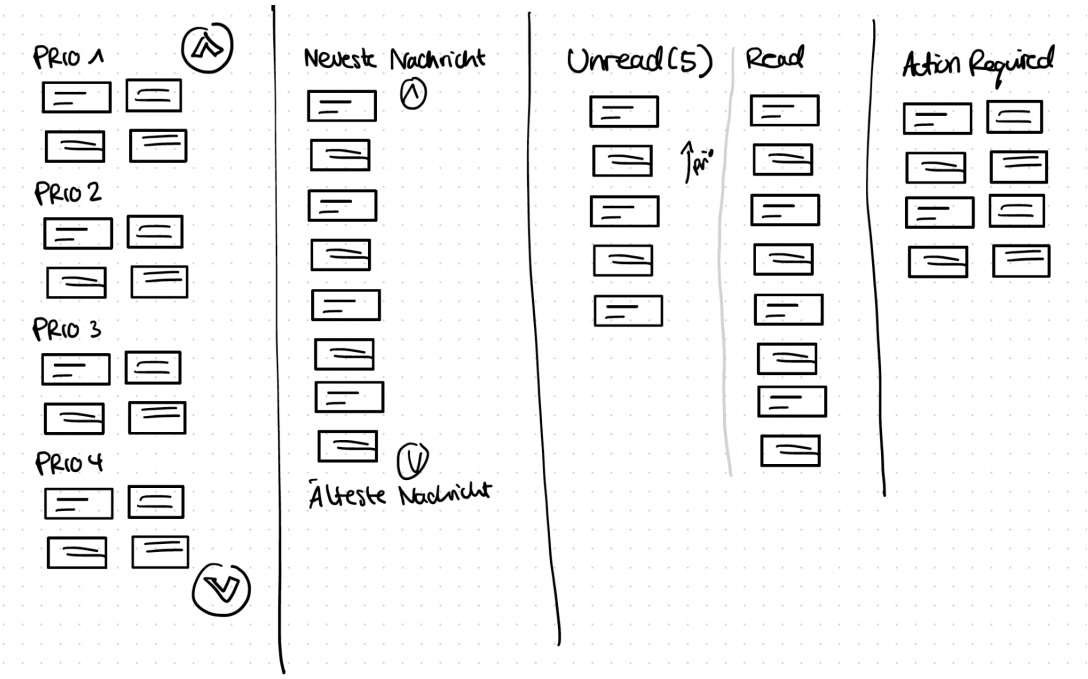


Figure 8: Sketch – Sorting Methods

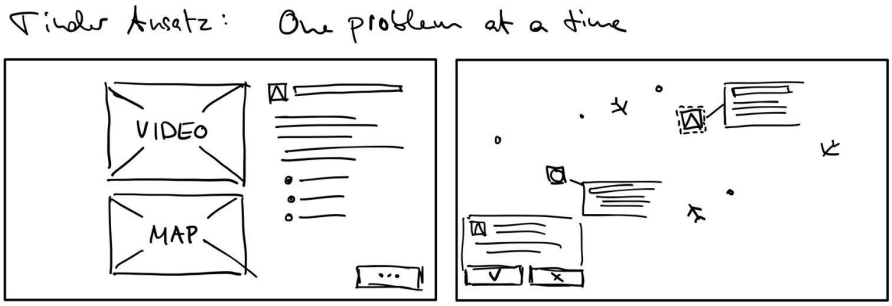


Figure 9: Tinder Approach

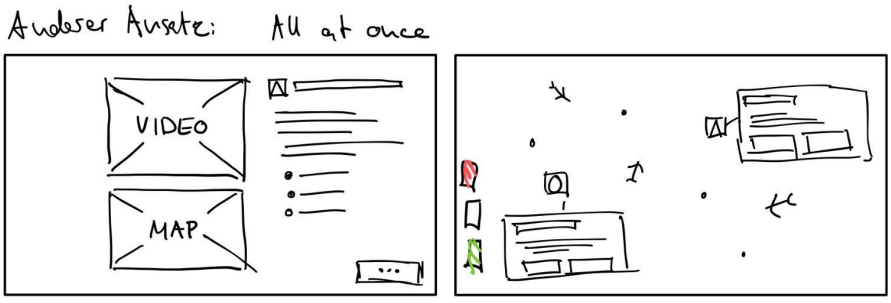


Figure 10: All-at-once Approach

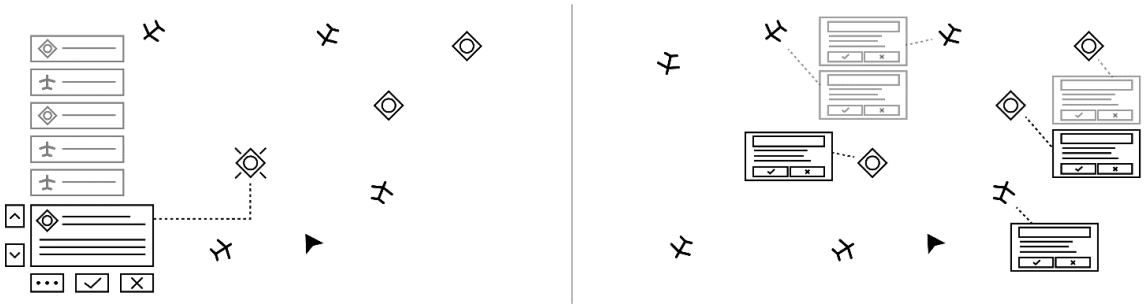
Figure 9 illustrates initial concepts of the “tinder approach”. In this approach, the pilot reviews messages one at a time, with the current message prominently displayed at the bottom right. This consistent placement ensures that the pilot always knows where the message will appear, facilitating quick and efficient interaction.

The “all-at-once approach” displays all incoming messages right next to the icon on the minimap to ensure a quick connection between the message and the icon it belongs to (see figure 10).

3.2 Wireframes – Two Basic Approaches

Our ideation and discussions resulted in two primary approaches for the central screen: the “list approach”, also known as “tinder approach” (see figure 11a) and the “all-at-once approach” (see figure 11b). The left-hand screen (see figure 12) should be used to display further information about a message.

In the list approach, the interaction with messages and the buttons is always on the left side of the central screen, preferably on the bottom to provide a good reachability. The current message is displayed as a large widget with additional info, with other messages collapsed above. Additional information is shown on the left screen, accessible via a “More Information” button. There should be a visual connection between event icons on the map and the currently selected message.



(a) Tinder approach

(b) All-at-once approach

Figure 11: Approaches for the design of the centre screen

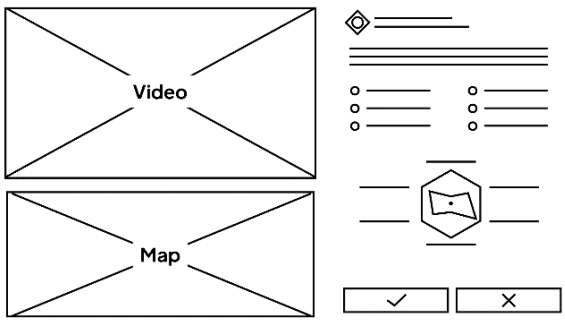


Figure 12: Conversation Details on Left-Hand Screen

As an additional idea, we discussed the use of gaze tracking to determine which icon on the minimap the pilot is currently looking at. This can then be displayed at the bottom left as the current message. A major point of discussion here was the flickering during gaze tracking and the lack of confirmation.

The main advantages of this approach are for instance the consistency. There is also the predictability regarding the message placement because the most important events are always in the same place. Another point is the simple reachability and an uncluttered map. On the other hand, there are disadvantages like no direct association between collapsed messages and dedicated map icons. Only one event's detailed information is visible.

In comparison to this, the all-at-once approach displays all messages on the minimap directly next to the icons. There is a visual connection between the event icon placed on the minimap and the interacted message. Each pop-up has buttons to interact with the message itself. Rather unimportant elements are displayed in a reduced form on the map. The main advantage of the all-at-once-approach is a direct association between messages and icons. It also provides a good overview over the events.

Resulting from the placement of the messages over the entire minimap, those placed on the top right cannot be easily reached. Another problem is the possibility of overlapping messages or a cluttered screen in case of too many messages.

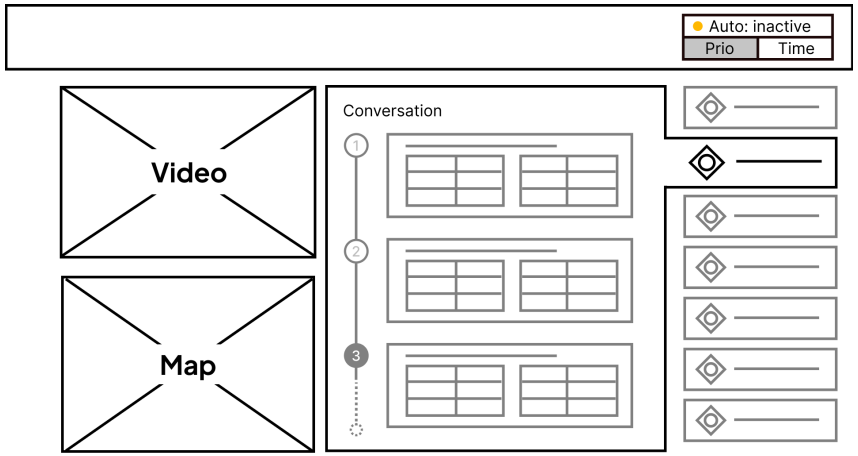
3.3 Final Concept

Given the pros and cons of both approaches, we decided to combine them. Additionally, we transitioned from touch-based to gaze-based interaction with gaze tracking. We developed a concept allowing the pilot to make inputs via a thrust master (for the left hand) and a thumbstick (for the right hand).

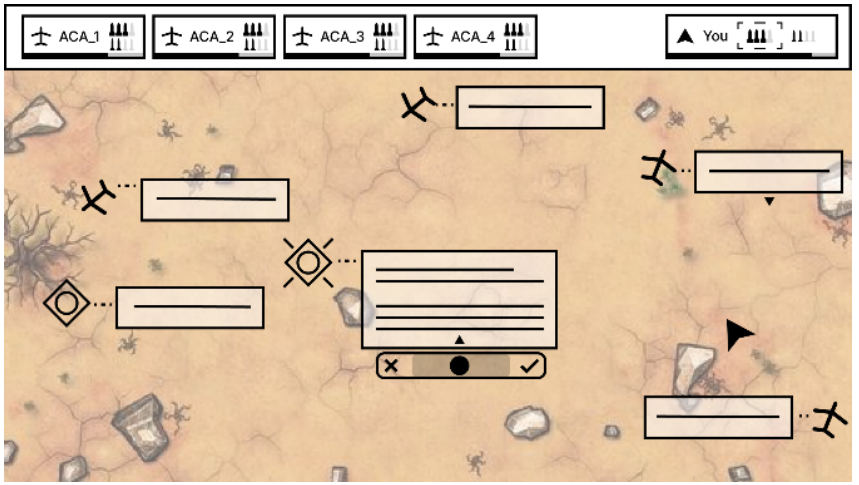
The left screen (see figure 13a) is still used to display detailed information about the selected messages. In addition, on the right side of the screen there is a list for an overview of all messages with a sorting function by priority, time or Gaia's assessment.

This leaves enough space for the minimap on the middle screen (see figure 13b). Messages with a high priority can be placed directly next to the icons to avoid a loss of time when searching for the right message in the list. The status of the drones (ACAs) and their available ammunition is displayed very simply at the top of the middle screen.

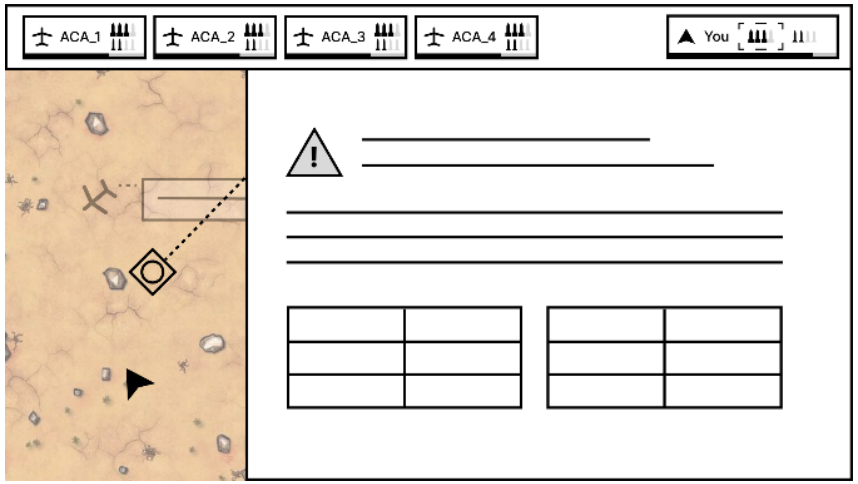
As soon as a threat with a very high priority is detected, the system switches to escalation mode (see figure 13c). The focus should be on the threat displayed on the minimap and the warning message should take up most of the screen. This allows the pilot's attention to be focused directly on the current threat. The right screen (see figure 13d) should remain available for displaying important information about the cockpit.



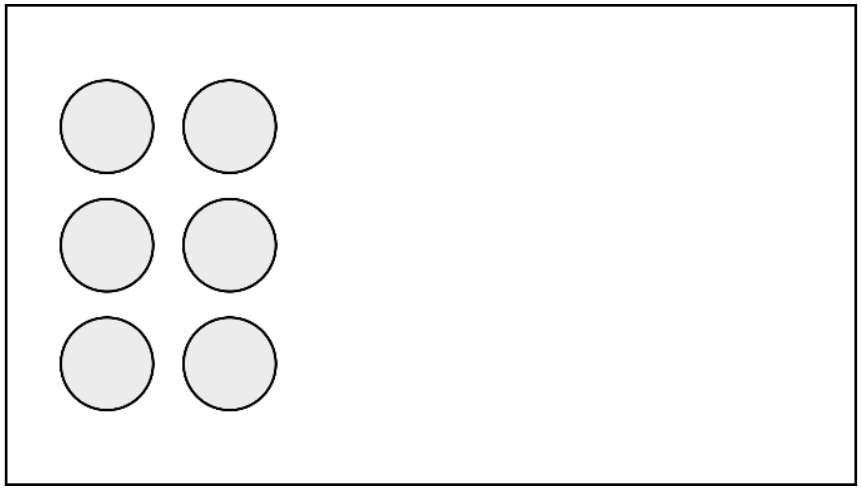
(a) Conversation details



(b) Minimap



(c) Escalation mode



(d) Flight information

Figure 13: Final UI concept wireframes

3.4 Interaction with Messages

To enable adaptive design and context-aware system, messages on the minimap are assigned four different states. In the small (S) state, only the threat icon is displayed on the minimap. This state is used for lower-priority messages to ensure the pilot can focus on higher-priority messages without distraction. In the medium (M) state, the threat icon is accompanied by the title of the message. When the pilot's gaze is directed at this message, it becomes highlighted. The pilot can then interact with the highlighted message by pressing the thumbstick on his right hand downward, expanding the message to the large (L) state.

The L state allows the pilot to read the full content of the message. Up to two messages can be expanded simultaneously. An Approve/Deny button, detailed in chapter 4.2, appears at the end of the highlighted message when the pilot's gaze is directed at it. In this state, the pilot has several interaction options: He can approve the message by pressing the thumbstick to the right, deny the message by pressing the thumbstick to the left, request more information by pressing the thumbstick downward, or close the message and revert it to the M state by pressing the thumbstick upward.

Some threat icons on the minimap are movable, but their associated messages remain stationary for readability until a predefined distance is exceeded. Once this distance is surpassed, the message repositions itself next to the threat icon. Messages should always be placed to the right of the threat icon and at the same height. If this is not possible due to obscuring, the message should be placed to the left of the icon. If that is also not possible, the message should be placed below the icon. Under no circumstances should a message be placed above the icon to avoid conflicts when expanding the message.

When a very high priority message appears, the system triggers Escalation Mode. In this mode, only a small section of the minimap is displayed, focusing on the critical area. A warning message (XL) slides in from the right side, providing all necessary information for the pilot to make a quick decision. This ensures that the pilot's attention is directed towards the most critical threat without delay.

4 High-Fidelity Design

4.1 Icons

Designing the icons used in our system was a tricky task, because they had to fulfill many visual and semantic requirements. The icons needed for the system could be split up into three kinds – threats, drones and the ownship. Each of these groups had a different purpose and needed to be distinctly different from the other groups.

The threat icons had to follow already existent guidelines (NATO Joined Military Symbology) for the use in military context. Due to that requirement, the icons are red and diamond-shaped. This ensures a fast registration of the event as a threat. To show what kind of threat the BM is facing, the inner part of the icon can be exchanged for different pictograms that each represent a special kind of threat. For our system, we decided to create icons for four different kinds of threat that seemed likely to come up in a battle situation. These are: radar station, air defense, artillery and missile. Each pictogram is inspired by the guidelines but slightly altered to fit into the general look of the interface.

To be able to adapt the icons depending on stress level, we introduced two dimensions: size and colour. Each icon is available in a small and a big state, as well as in pure red, light red and gray (see figure 14).

To ensure a good readability on all kinds of backgrounds, a high contrast was necessary. Going through multiple iterations, a double contour of black and white on the outside of the icons proved to be the best static solution. The inside of the threat icons is half transparent to allow the user to see through. A black border around all items inside the icons ensures the contrast there. As a result the threat icons are recognizable on all backgrounds they are put on.

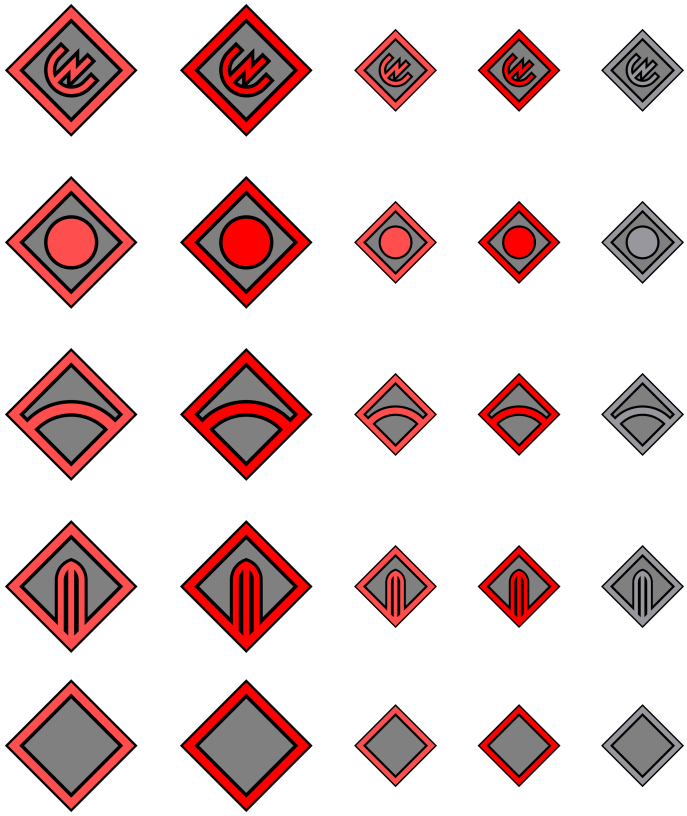


Figure 14: Different threat icons

The other icon group, the drone icons, was not predefined by guidelines, but it needed to be clearly recognizable as a friendly drone. To ensure that, we used the general outline of a drones body with a white border. Around the drone is a half transparent circle to ensure the contrast to the background. Underneath that, we put the drone's name so the battle manager is able to tell each drone apart. At each tip of their wings, the drones additionally have a blue part to indicate that they belong to the pilots team (see figure 15).

Last but not least, the icon for the ownship. This had to be distinguishable from all the other elements on the map. To ensure that, we used a green color and different shapes than with the other two icon groups. The center of the ownship icon consists of an arrow pointing into the direction the ownship is heading towards. Around the arrow is a green circle to make it easier traceable by the human eye. Here we also have a half transparent layer underneath the coloured elements to ensure contrast with every background (see figure 16).

Figure 15: Drone icon

Figure 16: Ownship icon

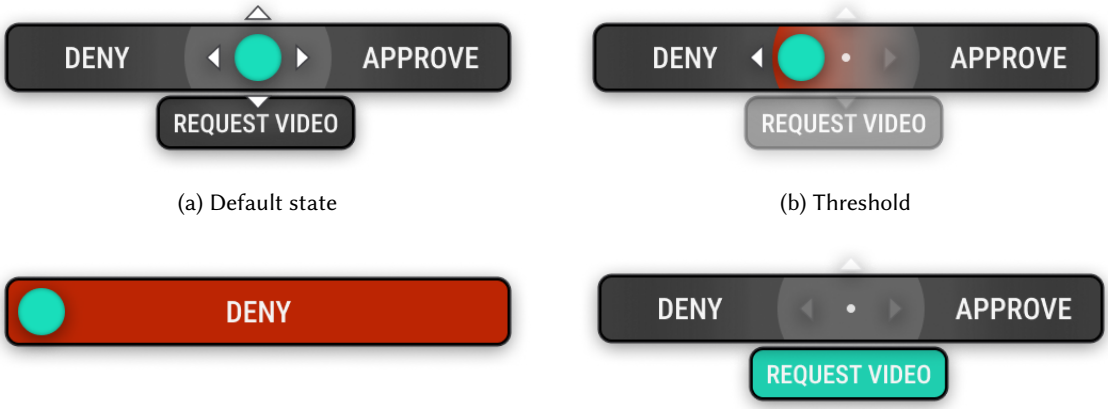
4.2 Approve / Deny Button

We went through multiple iterations with the button design. At the beginning of the project, we designed touch buttons, but later decided on gaze-only interaction, requiring the buttons to be adapted to this new context.

The button is located directly below an L-conversation widget, which specifies the required action. The button offers three options: typically "approve", "deny", and "request video", though these could be adaptive (see figure 17a). Interaction with the button is possible via the thumbstick on the right-hand controller. A turquoise circle in the center of the button indicates the current state of the thumbstick, moving accordingly as the thumbstick is manipulated.

For the approve and deny actions, the thumbstick must be moved left or right. A gray circle in the center of the button shows the threshold. Until this threshold is crossed, the action is not executed, allowing the pilot to change their mind. The circle changes color to red for deny or green for approve to indicate which action will be executed if the threshold is passed (see figure 17b). Once the decision is made, the entire button changes to red or green and then vanishes after a short delay (see figure 17c).

The "request video" option is executed by moving the thumbstick downwards, and it does not have a threshold. When this action is performed, the request video button turns turquoise (see figure 17d) and disappears after a short delay. The requested video then appears in the designated section on the left screen.



(a) Default state

(b) Threshold

(c) Request denied

(d) Request video

Figure 17: Approve/Deny button

4.3 Gaze Highlight

The gaze highlight gives the pilots feedback on where they are currently looking at and which widget is selected. It is displayed through a subtle turquoise border on the right and left edge of the widget (see figure 18). For M-conversation-widgets there is also a turquoise arrow appearing below the widget, that indicates that the widget can be expanded to L-state via the thumbstick.

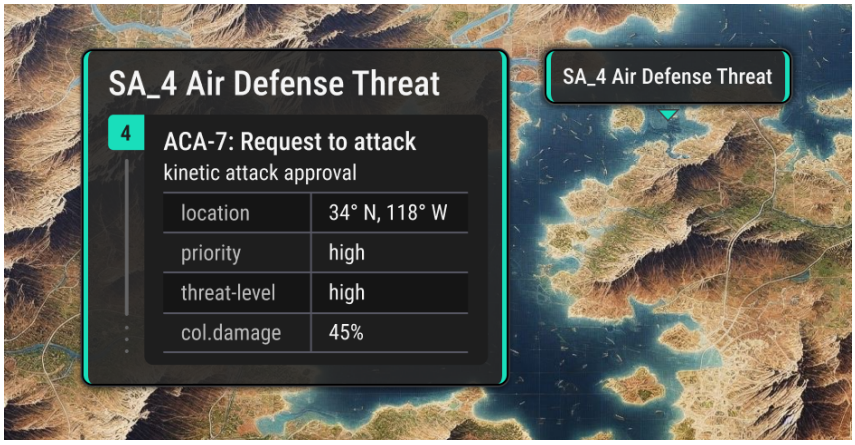


Figure 18: Gaze Highlight

4.4 Final High Fidelity Design

After countless iterations, we finally completed our high fidelity design. Figure 19 shows the set-up of all three screens. The “escalation mode” scenario, which occurs when a very urgent high priority message comes in, can be seen in figure 20. For example that could be a missile heading towards the ownship. In a scenario like this, the pilot has to make a decision within just a couple of seconds and therefor needs to focus solely on this task.

Figure 19: High-Fidelity UI

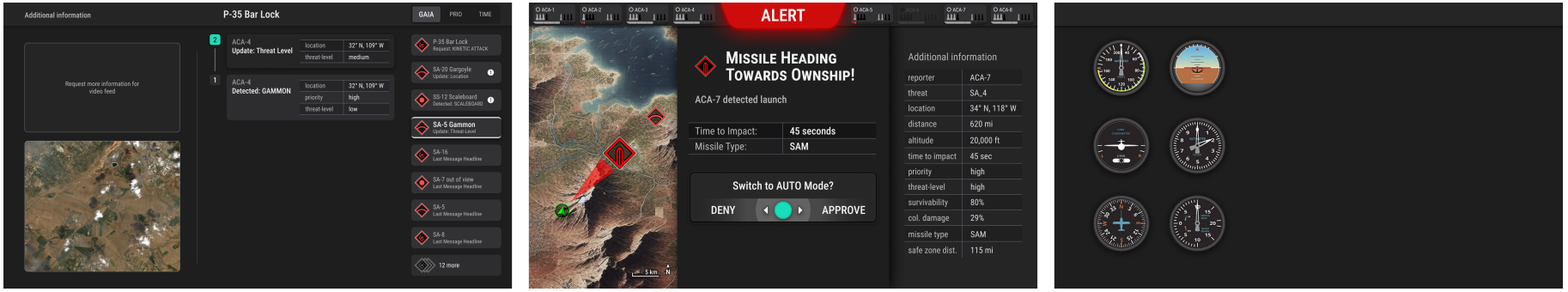


Figure 20: High-Fidelity UI (Escalation Mode)

5 Future Work

5.1 Map Behavior

We discussed, if the ownship should always be in the center of the map, since this is how maps usually behave in an aircraft. A disadvantage of this behavior would be, that the map is always in movement and all map content, such as the conversation widgets, would be hard to read when they are not steady. Also in our scenario, the pilot has to observe an entire team of multiple ACAs within a mission area. Therefor we decided, that it would be more helpful, if there is a map which always shows the whole mission area. In this case, the minimap would always be north oriented and the ownship would not be in the center, but moving around in the mission area as all the ACAs do.

If the ownship reaches the end of the visible area in the minimap, the map would zoom out until everything is visible again. The current scale is shown in the scale indicator.

5.2 Battle Manager Messages

To provide more context to the BM, the display of their own reaction on messages in the conversation history (left screen) would be helpful. This could be seen as some sort of chat where the chosen response on messages are displayed in a slightly different visualization. That way the BM is able to to see what decision have already been made in a conversation and the result of their choices. This is a relatively easy change that offers a big improvement in traceability of a conversation development.

5.3 Conversation Status Indicators

Threat conversations can have different stages in their life cycle, from detection over reaction request to results. To give the BM more context when scanning over the map or conversation list, a label system of some sort could be helpful. Specifically each icon and "conversation tab" (element in the conversation list on left screen) has one label indicating the current status of the conversation. This system allows the BM to see what kind of new information is ready to be read. In the current system we introduced a sub-label to each conversation that shows the last sent message with the type of message as the first word, which was the first step into this direction. Further development could include thinking about using colors for faster recognition of conversation types and incorporation into the icons on the map.

5.4 Spatially Distributed Visual Guidance

With the help of spatially distributed visual guidance, it would be possible to attract the attention and lead it to the current message. This could be designed with a visual warning signal. The light or arrow indicates the direction by lighting up (see figure 21).

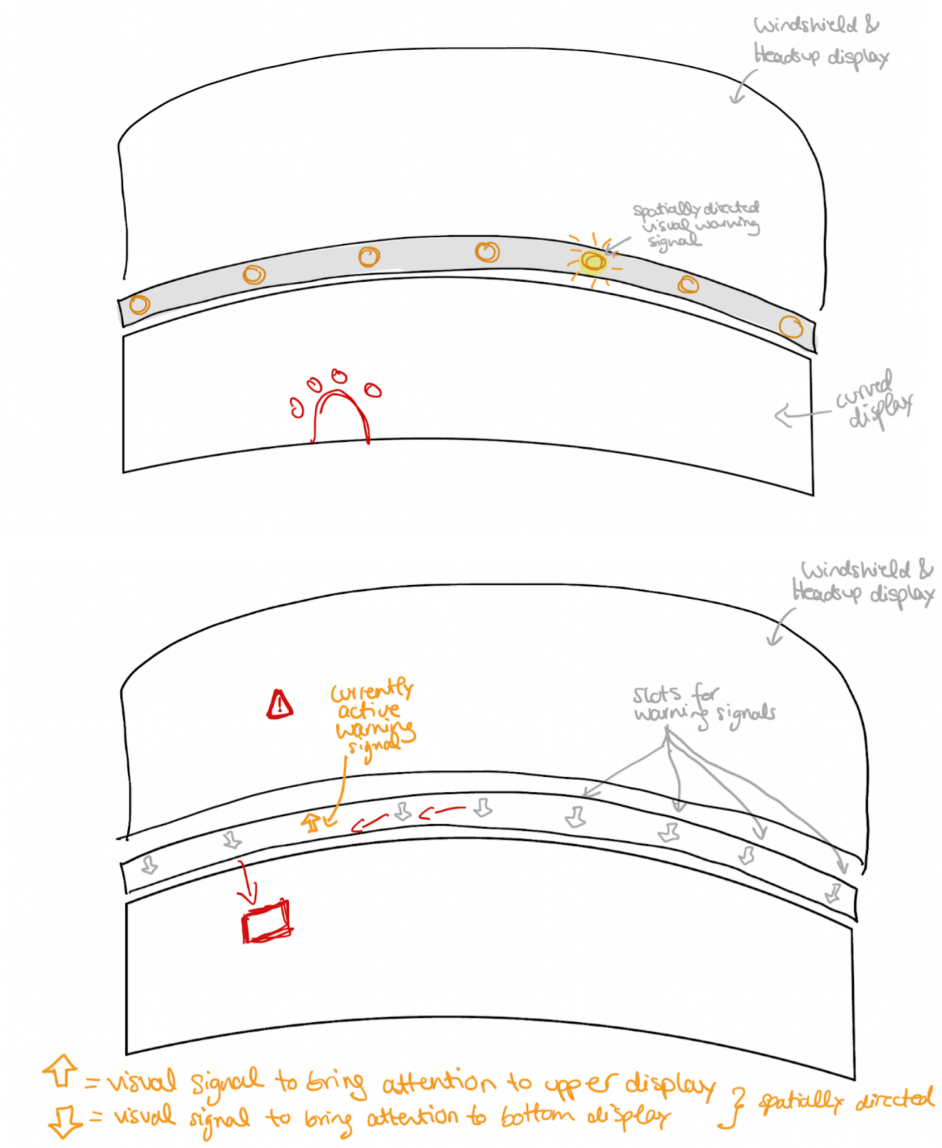


Figure 21: Spatially Distributed Visual Guidance

5.5 Audio

In the future we want to include audio alerts for high priority messages, as well as a simple notification sound for every incoming message. Additionally, there should be a voice, that calls out important alerts. More research is required in order to figure out how visual and audio alerts can be properly combined in the system.

5.6 Thumbstick and Thrustmaster

For our multi-modal gaze interaction approach, we need an additional input method that goes along with gaze interaction. We were thinking about having a scroll wheel on the thrust master, that is used with the non-dominant hand. That scroll wheel can be used to scroll through the conversation list

on the left screen. There should also be two buttons located on the thrustmaster, for selecting a new conversation from the list and for filtering.

The dominant hand of a pilot normally always stays on the aircraft controller. On that controller we want to mount a thumbstick, that can be used to interact with the buttons on the map (approve, deny, request video or expand/collapse widget).

For future work, there is more research and user testings needed in order to understand if and how the interplay of gaze interaction and physical buttons in the aircraft works.

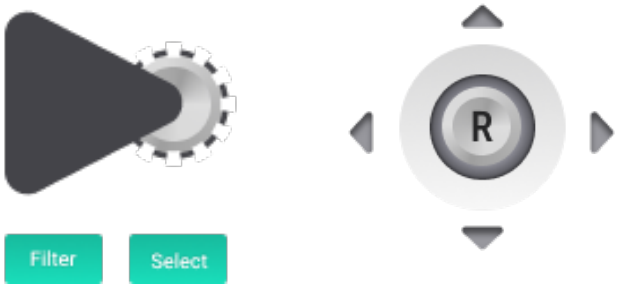


Figure 22: Thrustmaster and Thumbstick

5.7 Ergonomic hand rest

In the beginning of the project, we focused on a touch-based interaction. Therefore we had to think about solving problems like the gorilla arm syndrome or precise input even in a vibrating aircraft. An ergonomic hand rests could help to improve touch input accuracy and reduce muscle fatigue.

Figure 23: Caption

II Theia

1 Introduction

Our goal when creating Theia was to make a suite of tools that can be used beyond just military ownship-drones applications. To this end, we created a generalized framework that can be adapted for any multi-modal configuration where a single operator completes some tasks. A configuration could be as simple as a shopping application UI with little changing data or as complicated as an adaptable network-hacker-defense UI that requires widely varied data to be displayed at different times. Theia is derived from three tools: The World Simulator, the Conversation Manager (CM), and the Widget Element flow (see figure 24). The World Simulator sends simulated stress level data and informational messages to the CM, which uses its own logic to adapt the user's UI to the world's current and past context. The CM decides what to place and how to place it using the Widget Element Flow.

Figure 24: Theia Architecture Overview

To create a new interface, the main functionality and logic of our CM and the World Sim remains untouched. Developers simply must create Widgets and Elements and organize them within the Layout Preference Document (LPD) system.

While we do use React Redux, we created an interface model that is a little different than a typical React interface implementation. Since the UI is not static and could be rapidly changing at any time, we needed a framework that did not have hard-coded components throughout the screen. So, we built a system that allows any Widget to be placed anywhere on the screen at the exact place and time it needs to be there. The general UI development framework works by defining what we call the "Widget Element flow".

Our overall goal was to make Theia "plug and play", where you "plug in" a new Widget Element flow and our CM "plays" the UI by placing Widgets and Elements where they need to go at the time they need to go there. As of writing this, There are only five segments that need to be changed to create any new interface. The only things that need to be changed to create a new UI are the Widget and Elements in the components and UI segments, the assets (images, sounds) in the assets segment, the LPDs in the LPD segment, and the Widget and Element types in the types segment. All of these segments are logically separate from core CM functionalities, ensuring that developers can focus all of their efforts on designing a new interface.

Note that for the ease of writing, the remaining portions of the Theia documentation will mostly assume that Theia is used in an ownship-drones interface, where a battle manager (BM) is piloting their ownship while also handling the operation of multiple armed drones in their region.

Why the name Theia? In astrophysics, Theia is the name of Earth's, or Gaia's, theorized sister planet. Theia and Earth both orbited the Sun in the early Solar System around 4.5 billion years ago, though Earth was much bigger than Theia. Theia would be pulled towards the Earth and crash into it. The remaining debris of Theia that did not stay on Earth would then form the Moon. So, Theia is Gaia's (our reaction model's name) counterpart.

2 Widget Element Flow

To define the flow, we assume that there is some sort of UI wireframe or prototype that the application can be modeled after. Before we start writing any actual code, we first define the Elements that need to be placed across the screen. This requires defining each Element's functionality, what information it needs access to from the Redux state, and what information it needs to share with other Elements. After all Elements have been defined, we can group them into Widgets, where each Widget is some number of semantically grouped Elements. Next, we define Sections, which are regions of the screen where Widgets can be placed.

A simple example that explains the purpose of Sections, Widgets, and Elements is in relation to land ownership where the entire screen can be equated to a country. Sections then become states. Widgets are personally owned properties and Elements are the buildings that are built on personal land property. In this way, we can see that different regions of the screen are always "owned" in some way, often grouped by semantic relationships. Just like buildings and personal land property, Widgets and Elements are not static and can change throughout the UI's runtime. See figure 25 to visualize their relationships.

Theia relies heavily on the notion that stress level changes in the user will require some UI adaptations to make the UI's complexity go down. For developers to handle these changes, we created the LPD system. An LPD is a file that gives directions as to what Widgets should exist and where they should exist within different user stress levels, meaning each stress level has its own LPD file. Theia assumes three stress levels: Not stressed, mildly stressed, and very stressed. Though we use three stress levels, we have four LPD files. The first three are stress level LPDs that define Widgets for the different stress levels, while the fourth is the initial LPD. The initial LPD defines everything that should exist at the beginning of the runtime, often with Widgets that will remain on screen throughout the entire UI runtime. The initial LPD also defines all Sections. Sections are static throughout the entire UI runtime and can overlap if they will be used during different stress levels or for purposeful design usage.

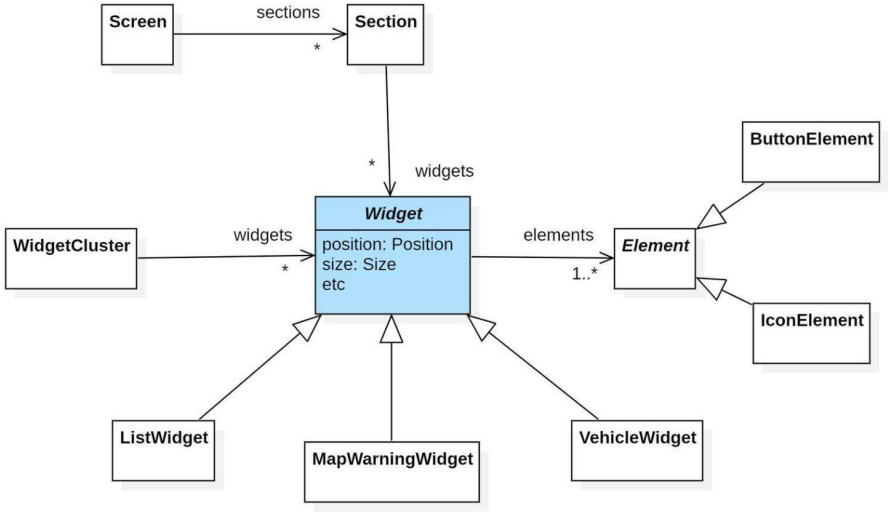


Figure 25: Relationships between Widgets, Elements, Sections, and Widget clusters

Stress level LPDs are written in a message → response layout. This means a response is defined for every message that could come into the CM. A response to a message will always be in the form of an array of Widget Clusters. A Widget Cluster is a container of Widgets that should be placed or updated in response to a message. The response is in the form of an array because each Widget in a Widget Cluster must be able to integrate into the UI to be deployed, otherwise the CM will try to integrate a different Widget Cluster from the array. This array of Widget Clusters gives the CM different options to convey any message.

With the Widget Element flow defined for all stress levels, we can finally start writing the code for the Widgets and Elements. Widgets are fairly simple, mostly acting as containers to group semantically related Elements. Elements are a little tougher. Because Elements can be so dynamic, our CM gives them a lot of autonomy to make their own decisions and manage their own state. This means that there is a lot of logic to implement when programming Elements. Two of the most basic features to implement are reactions to the Element being in gaze and a key being pressed while the Element is in gaze. For more complicated Elements, there may be much more to implement.

If Elements need to share information with other Elements, they can use developer-defined variables within the communication slice of our Redux architecture. This slice is solely for handling information that can be shared across two or more Elements within the UI. For example, if an Element that lists all messages wanted to communicate what message the user selected to a separate Element in the UI, it could update an `activeMessage` variable in the communication slice to reflect the newly selected message.

Another important feature to implement when programming is Widget updates. When our CM tries to place a Widget that already exists and should not have multiple instances, like a list for example, it will update the existing Widget rather than create a new one. The existing Widget will be informed that they need to be updated, taking in the proposed Widget's Elements as input and deciding how it should update itself. This feature represents our CM's ability to be flexible with almost any design that is needed. Once all of these steps are complete, the Widget Element flow, and thus the new interface, is complete.

3 World Simulator

The purpose of the World Simulator is twofold. It defines the messages and context information that are to be sent to the Conversation Manager, and provides a way of editing the message timeline. Here, messages are specific events that should be displayed in the user interface, such as detected threats or important status updates on the ACAs. Multiple messages can make up a conversation, where additional messages are based upon information already provided by the messages sent earlier as part of that conversation. For instance, a conversation might start with a "threat detected" message, which is followed up by a status update on that threat and finally a "request approval to attack" message. The context includes various information required for adapting the UI, such as the current stress level of the user.

For sending the messages and context, the World Simulator provides a WebSocket interface, which the Conversation Manager can connect to in order to receive the messages. To keep things simple for the time being, this communication is a strictly one way, where the CM only receives messages and does not send any information back to the simulator, such that user inputs have to be handled by the application instead of the World Simulator.

Apart from the message sending capabilities, the World Simulator serves as a timeline editor. It provides validation for message data, various features for changing the timeline itself and editing the data of its queued messages, saving and loading timelines to local files, pausing and resuming the timeline as well as manually sending messages and context information for debugging purposes. All in all, it serves as a developer tool for the team to be able to test the CM easily.

In order to make the development easier and more flexible, the World Simulator is implemented using Electron. This brings the additional benefit of providing a cross-platform graphical application while still using the JavaScript environment that we are already employing within the Theia project.

The World Simulator is designed to be flexible and fully reusable. It can be easily adapted for any kind of application that relies on a stream of messages by merely changing the message schema types and the default timeline in most cases.

4 Conversation Manager

The Conversation Manager (CM) is what drives the main logic of any Theia UI. Figure 26 shows the five main modules: the Modality Selector (see section 4.1), the Modality Assimilator (see section 4.2), the Modality Restrainer (see section 4.3), the Modality Monitor (see section 4.5), and the Stress Change Handler (see section 4.6). These five modules work together to ensure that the right Widgets are at the right place at the right time.

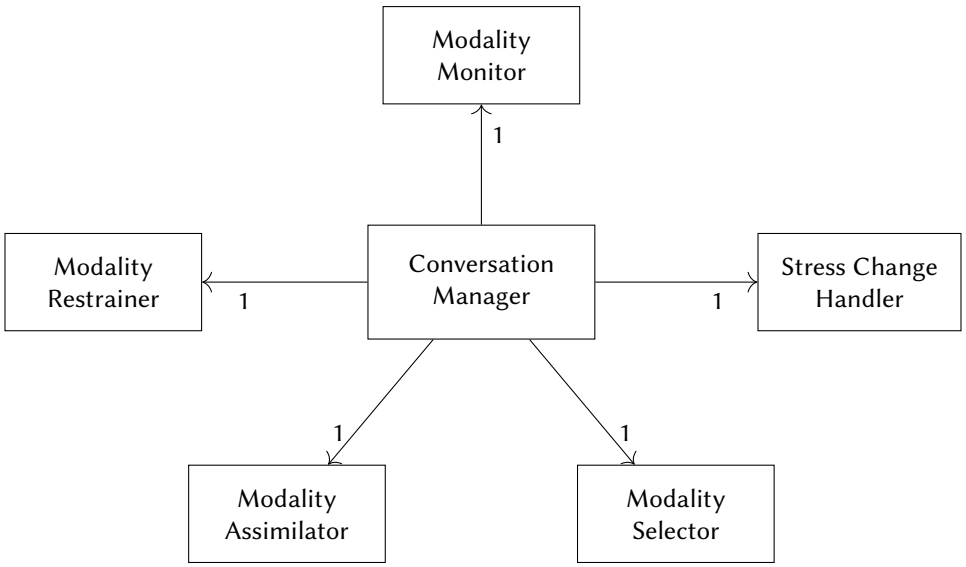


Figure 26: CM Components

4.1 Modality Selector

The Modality Selector (MS) is the interface between the LPDs and the CM (see figure 27). It obtains the possible sections, Widgets, and Elements from the LPD depending on whether there is a message, stress level, or both. The MS distinguishes between three cases as soon as it is called up.

Retrieving initial LPD If the call takes place without a message and a stress level, the selector the initial LPD.

Retrieving all Widgets in a stress level LPD If only a stress level is sent to the MS, the LPD corresponding to the stress level is called and all Widgets contained in that stress level LPD, along with all Widgets within the initial LPD, are returned.

Retrieving Widget clusters in response to a message If a message and a stress level is sent, the MS queries the LPD corresponding to the stress level and then returns the Widget Clusters associated with the message.

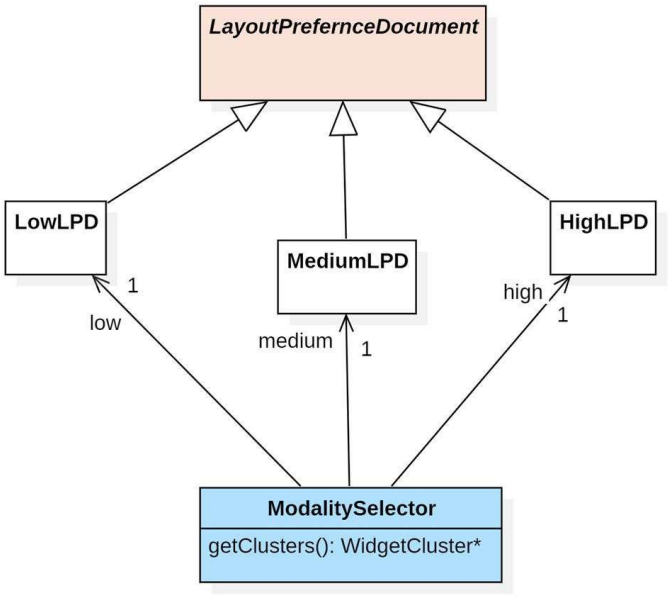


Figure 27: The relationship of the Stress Level LPDs to the Modality Selector when retrieving Widget clusters in response to a message

4.2 Modality Assimilator

The Modality Assimilator (MA) takes in the Widget clusters returned by the Modality Selector (MS) and attempts to integrate one of them clusters into the UI. Upon successfully integrating a cluster, that cluster will be returned with instructions on how to deploy it. If no cluster can be integrated, then an empty Widget cluster is returned.

The MA receives possible Widget clusters, all deployed Sections, all deployed Widgets, and the message that is currently being reacted to. We look at each Widget cluster one by one and in order, trying to integrate all Widgets within the cluster. Within a single Widget cluster, we look at each of its individual Widgets.

We first check if the Widget already exists and is deployed to the UI. This is done by comparing the proposed Widget's type to every deployed Widget's type. If the proposed Widget is a specified Widget, then the MA also checks to see if a deployed Widget with the same type has the same tags as the received message. If both are true, then the found deployed Widget is the same Widget that is being proposed. If the proposed Widget is not specified, then the type just has to match.

If the proposed Widget already exists, the MA checks to see if the existing Widget already handles the message that was received. It checks if the received message id is within the deployed Widget's `handledMessageIds` array. If the message is within the array, then the Widget already exists and handles the message. If it isn't in the array, then the Widget exists and does not handle the message.

Based on the previous two criteria, there are three options that will occur for every proposed Widget within a proposed Widget cluster. Below you can find details on the exact steps of what happens, but the result is that the proposed Widget is integrated into the UI, an existing Widget is updated, an existing Widget does not need an update, or the proposed Widget is rejected. If all proposed Widgets with a cluster fall into one of the first three options, then the proposed Widget cluster was successfully integrated and is returned. If any proposed Widget within the proposed cluster is rejected, the cluster is also rejected. The MA will then try to integrate the next cluster in the list.

Widgets that don't exist When a Widget doesn't exist already, the MA will try to create a new Widget and integrate it into the UI. It will start at the top left of the section that is specified within the proposed Widget, slide down and to the right searching for a spot big enough to place the Widget without overlapping other Widgets (unless a Widget is a `canOverlap` Widget). If the MA can find a spot to place the Widget within the section, then the proposed Widget's entry within the proposed Widget cluster will be updated with the new Widget's location and tags (if its specified), along with the action of `newWidget`. If a proposed Widget cannot be placed, then the proposed Widget is rejected, also rejecting the proposed Widget Cluster.

Widgets that exist and do not handle the message When a Widget exists and does not handle the received message, the proposed Widget's entry within the proposed Widget cluster will be updated with the existing Widget's ID and its action will be set to `updateWidget`.

Widgets that exist and handle the message When a Widget exists and also handles the received message, the proposed Widget's entry within the proposed Widget cluster will be updated with the existing Widget's ID and its action will be set to `messageAlreadyHandled`.

4.3 Modality Restrainer

The job of the Modality Restrainer (MR) is to keep our modality measures within a predefined boundary, where that boundary changes in response to the current context (higher stress, critical situation, etc). In essence, it "restrains" the allowed modalities within the UI. The MR is before a Widget cluster is deployed to the UI.

Modality Measure Each modality has a modality measure, which is the measure of "how much" of a modality is present in a UI. In our implementation of the MR, we currently only have one modality: Visual. The visual modality measure is currently simulated by the amount of Elements deployed to the UI divided by the maximum number of Elements allowed on the screen (currently 100).

Modality Measure Ranges Each modality measure has a set range that is allowed within the UI. The range starts with its minimum indicating that this modality is not used at all and reaches until a

defined maximum that represents that this system doesn't have a possibility to create more complexity for this specific modality. For instance, in our visual implementation we could choose to have a range of 0 to 100, where 0 is a black screen and 100 is the whole screen taken up by Elements overlapping each other with colors rich in contrast and a major amount of text.

Modality Measure Boundaries Modality measure boundaries are used to restrict how much of a certain modality is an acceptable value for the current context. The basic idea here is that they can be adjusted to the respective stress level of the user in the future. This could, for example, prevent the user from being shown extremely complex visual Elements in already high-stress situations to not overwhelm them. Each modality measure has a modality measure boundary within the modality measure range. This boundary contains the minimum and maximum values that a modality measure could exist in to be considered "in bounds". So, while our modality measure range for the visual modality could be 0 to 100, the boundary could be 20 to 80.

The MR is used to restrain "how much" intensity of an individual modality or group of modalities is allowed to exist within the UI.

When the MR receives the integrated Widgets from the MA, the MR checks whether the modality measures would still be within the limits of the modality measure boundaries if the integrated Widgets were to be deployed. The conceptual approach is as follows: the MR retrieves the complexity of the current UI and also retrieves the complexity of the UI if the integrated Widgets were to be deployed. Afterwards, it is checked whether the new complexity is outside the acceptable modality measure boundary. If the complexity is outside of the bounds, the MR returns false. If the complexity is within bounds it will return true.

It is important to note that our MR currents simulates the visual complexity and assumes that all complexity calculations are done from separate complexity modules outside of the CM, just as visual complexity has its own planned module outside of the CM. The complexity is currently simulated via a function within the MR, but later would be done outside of the CM once the separate modules are created.

4.4 Reacting To A Message

In this section, we will discuss the process we go through in order to react to a new message. This process is also illustrated in figure 28.

When we receive an informational message, we first query the MS to get some number of clusters of Widgets that can be deployed to the UI in response to the message. A single cluster of Widgets is a group of Widgets that should be deployed in response to a single message. The MS can return multiple

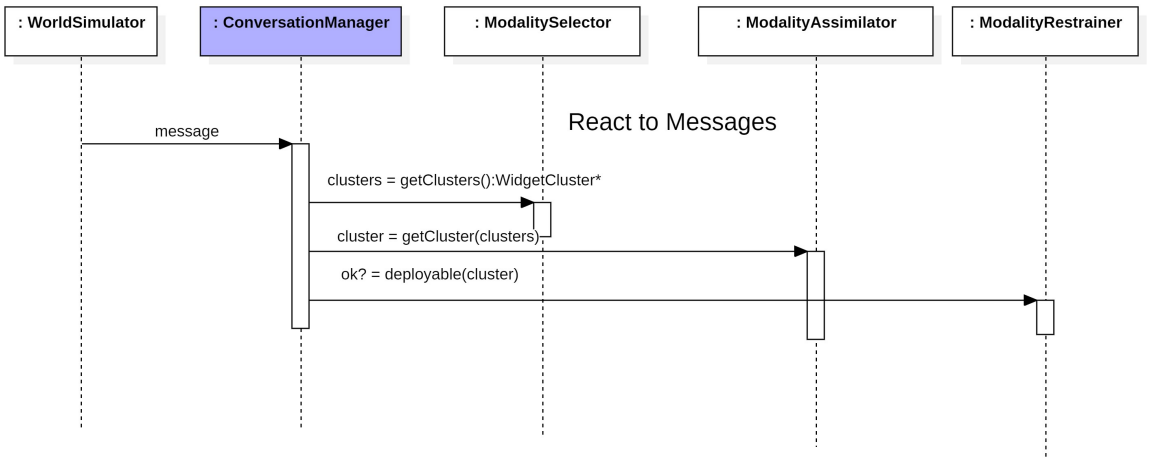


Figure 28: Reacting to an incoming informational message

of these clusters to give the MA options of which cluster to deploy. These clusters are currently selected via the LPD system, which has an entry for every message for every stress level.

The CM then hands the returned Widget clusters to the MA, which decides which cluster to integrate, if any. For each cluster, the MA goes through every Widget and checks if the Widget can be placed on the screen, if the Widget already exists and can be updated with the received message, or if the Widget exists and already handles the received message. If none of these three options are applicable, then the Widget cluster is rejected, and we try to integrate the next one. The MA tries to place the cluster in the first index of the cluster data structure first, then will repeat the same process for every cluster after. If every Widget in a cluster is successfully integrated, then we can integrate the cluster and return it.

Even though we have chosen a Widget cluster to integrate, we cannot yet deploy it to the UI. As a final step, we submit the integrated Widget cluster to the MR and check if the integrated cluster would cause the UI to exceed any modality boundaries. If deploying the cluster remains within boundaries, then we can deploy and update all Widgets within the integrated cluster. If the cluster does cause an overflow, then the integrated Widget cluster is rejected and removed from the list of clusters returned from the MS. The MA will then attempt to integrate a new cluster. If it can't then there is no reaction to the received message.

4.5 Modality Monitor

The main task of the Modality Monitor (MM) is to monitor how Widgets and Elements have been interacted with. It checks through the application components and applies any necessary updates common for all application components. The way this is done will be described in the following section.

According to Hesheng et. al. [4], it takes the average human around 100 ms to visually comprehend an image. So, the MM continuously runs every 100 ms to executes multiple tasks (see figure 29). First, it tracks interactions with gaze and keypresses. When a user looks at an Element for at least 100 ms, the Element is added to the `ElemsInGaze` data structure within the Redux state. When a user presses a key or clicks the mouse while looking at an Element, the Element along with the key/mouse press will be added to the `GazeAndKeys` data structure in the Redux state for Elements to check. In being interacted with via gaze or clicks, the Element's interaction window is updated to mark that the Element was interacted with at the time of interaction.

The MM also detects whether Element interaction windows have expired. Each Element has its own defined interaction window and time of expiration. When that interaction window is surpassed, meaning the Element has not been interacted with by the time of expiration, the MM will trigger one of three events specified by the expired Element. An Element can either be deleted, escalated, or deescalated. Deletion of an Element is controlled directly by the MM, which immediately deletes the Element from the UI if the Element's expiration action is "delete".

When escalating or deescalating, the MM does not currently force any changes upon the expired Element. Instead, the MM directly notifies the Element that is should escalate or deescalate, then the Element has autonomy to change how it chooses in response. For a visual Element, escalation would usually correlate to getting bigger or more visible, while deescalation would correlate to getting smaller or less visible.

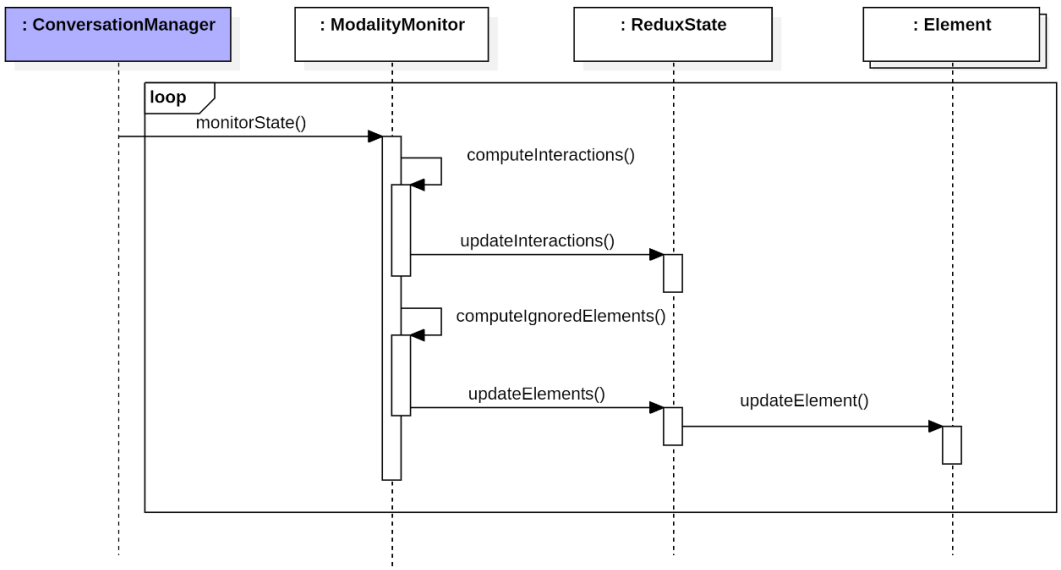


Figure 29: The Modality Monitor's looping cycle. Loop occurs every 100 ms.

4.6 Stress Change Handler

In order to accommodate changes in user stress levels, we implemented a module that adapts the UI to a user's new stress level to match the current context. The entire process of the Stress Change Handler is illustrated in figure 30.

Stress levels are represented as a floating point value between 0 and 1. Closer to 0 would be lower stress and closer to 1 would be higher stress. In our implementation we equally divide this floating point into an integer; 0, 1, or 2 represent low stress, medium stress, or high stress. These are the three critical stress levels

We call the Stress Change Handler when there is a change in critical stress level. So the Stress Change Handler is not called every time the stress level changes, only when a stress level change causes a critical stress level change. So a stress level change from .1 to .2 would not call it, while a stress level change from .1 to .6 would call the Stress Change Handler as we would change from the low critical stress level to the medium critical stress level.

First, we get all Widgets that are in the new stress level LPD and initial LPD. These Widgets are the only Widgets that should be allowed in the UI at this stress level. We then go through every deployed Widget and check if it exists within the new stress level LPD or initial LPD. If it does not exist, then it does not belong in the UI in the current stress level and thus is deleted. Widgets that still exist are unchanged by the stress change handler (Note: Elements are in charge of changing their state in response to stress level change).

After pruning the UI of ill-fitted Widgets, we then run through and react to every received message that has not been fulfilled. We do not use fulfilled messages because those messages no longer need

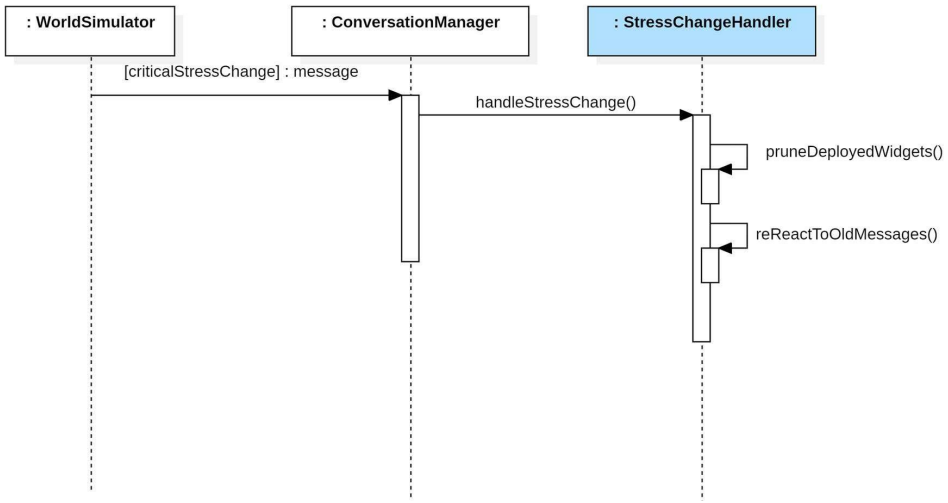


Figure 30: The Stress Change Handler process

to be reacted to by either the CM or the user. By running through every message again, every Widget that should exist in response to incoming messages are deployed as if they were placed at the message's reception.

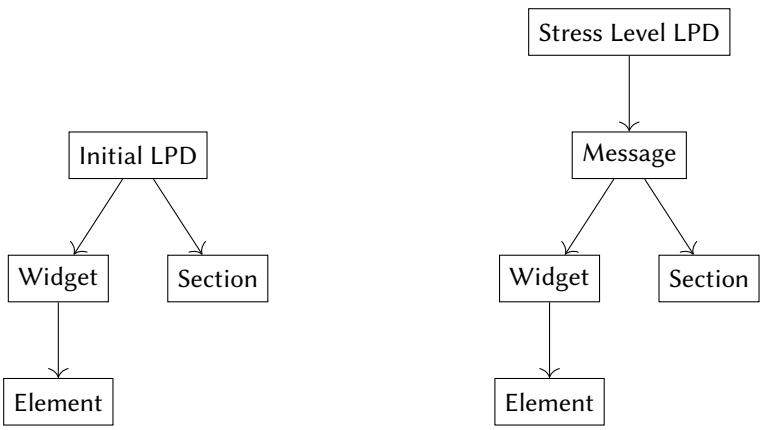
When a message goes through the MA in a repeat run, the assimilator will not repeatedly re-add or update any deployed Widgets due to our message handling system. So, only the Widgets that should be deployed or updated within the current stress level will be integrated.

4.7 Layout Preference Documents

The layout preference documents (LPDs) form the basis for selecting the optimum layout adapted to the current stress level of the user and tries to decrease the visual complexity once the stress level rises. They contain adequate layouts for Widgets and Elements for the different stress levels defined in Theia, as well as for the initial setup of the UI.

Initial LPD The initial LPD (see figure 31a) is called as soon as the application is started. It provides the basic layouts for the UI. This means that the different sections, Widgets, and Elements are created, which remain persistent in the program throughout the application. For example, the top bar with the drone information and the map background are created for the Minimap screen. The icons for the drones and the ownship are then displayed on it. All other changes that are made or called during runtime are handled by the stress level LPDs.

Stress Level LPDs The relevance of LPDs is to design the layout of the application for a user in high-stress situations so that it provides assistance with concentration on the most urgent messages and decisions. It should also be guaranteed that the messages are displayed as intuitively as possible and show an appropriate amount of content that allows the pilot to make important decisions without too much cognitive load and without the loss of information. The stress level LPDs (see figure 31b)



(a) Initial LPD (b) Stress Level LPD

Figure 31: LPD Diagrams

currently comprise three stress levels, and thus three different files: high, medium, and low. There is one LPD for each stress level. Each of these LPDs contains a design for deployable Widgets, adapted to the respective stress level. As soon as a message is received from the World Simulator, the current stress level used to select the appropriate LPD. Then the Widget Clusters corresponding to the message are selected and returned for actual placement.

4.8 Redux State

Being a system that is aimed to be used in high stress and possible life-threatening situations, our CM uses the Redux framework to ensures an easily accessible and consistent global state that enforces predictability. Our global state, handled by Redux, is partitioned into four slices such that each serve a different purpose: conversation manager, conversation, gaze, and communication.

Conversation Manager Slice The Conversation Manager slice serves two purposes. It acts as a central organization source for the sections, Widgets, and Elements deployed in the UI and also as a provider for relevant actions to perform on these UI types. From removing Widgets, escalating Elements, or updating the visual complexity of the UI, the `cmSlice` handles the basics of UI manipulation and organization.

Conversation Slice The conversation slice serves to store all conversations and their relevant information. A single conversation is a collection of messages that are related and thus grouped together. As messages arrive, either from the World Simulator or from the real world, they are grouped into their corresponding conversation. Some additional functionalities include the ability to fulfill a message or update the number of unread messages.

Gaze Slice The gaze slice stores the gaze and key/mouse interactions from the user. When a user interacts with an Element in some way, the gaze slice is responsible for storing that interaction.

Communication Slice The communication slice serves for shared data between Elements. As an example, our ownhip-drone interface provides List, History, and Video Elements that rely on a user selected conversation. The selected conversation number is stored within the Redux communication slice, where all three Elements can use and manipulate the selection as needed to respond to the user's selected conversation.

4.9 Tagging

Tag Tags are simply an array of strings, where each string is a single tag. Tags tend to be all lower case and use a dash in place of a space. Tags are defined in two places: Messages and Widgets. Tags are in Messages to tell the CM which real world instances were affected within a message. In Widgets,

tags are used to identify which real world instances a Widget serves. A Widget that serves a real world instance can then check if incoming messages have information relevant to that real world instance.

Specified Widgets A Widget is specified if there can be more than one of the Widget and if it serves some real world instance. A specified Widget also requires a "specify" tag, which tells the MA to assign it a real world instance when it is integrated into the UI. An example of a specified Widget could be a drone dialogue Widget that represents drones on a minimap. Each drone dialogue Widget serves a different drone, thus each one must have a tag that specifies the real world drone it is serving.

Specifier A specifier is the unique tag that identifies which real world instance the specified Widget is serving. In the case of identifying drones, the tag `"aca-1"` would be a specifier that communicates that the specified Widget serves the real world `"aca-1"` drone

Specified Tag A specified tag is a subset of an accompanying specifier, being identical to the specifier without the unique identification portion. Specified tags work both as a marker for what type of real life instance a specified Widget can serve and as a clear separator for a new segment of different specifiers. A specified tag is immediately followed by its accompanying specifiers. As an example, `"aca"` would be the specified tag to `"aca-1"`, where `"aca-1"` would immediately follow `"aca"` in a tags array. If a received message contained a new threat named `"threat-4"` and information of an existing threat named `"threat-2"` that were both found by `"aca-1"`, then the message's tags array would look like `["aca", "aca-1", "threat", "threat-4", "threat-2"]`.

Tags are used in the MA and in Elements that belong to a specified Widget. In the MA, tagging is used to identify specified Widgets in order to update them or to assign specifiers to specified Widgets when they are integrated into the UI. In Elements, tagging is used as a way to poll for information from each received message. When we receive a new message, Elements within a specified Widget can look at the tags of the message to see if the message's specifiers match their parent Widget's specifiers. If they match, then the received message has information pertinent to the real world instance that the specified Widget, and thus Element, serves.

Limitations of our current tagging system

- Specified tags cannot include other specified tags within them
- When creating a new Widget or checking if that Widget exists, the first specifier in message tags is used for identification. So if a message contains many specifiers for a specified tag, the first specifier should be the real world instance that sent the message/has the most critical role in the message

- Each Widget can only have a single specified tag. If a tag needs to be able to handle multiple, one would need to create a new specified tag that encapsulates both properties
- Can only update one Widget per Widget type per message (Elements can still update themselves)
- Widgets are assigned real world instances at integration, so cannot change to other instances

4.10 Custom Hooks

Our custom hooks track multiple interactions and UI states throughout the runtime of the application. Many of our hooks are stored in some way into the Redux state for communicating user interaction to the entire system.

useWorldSim The `useWorldSim` hook connects the World Simulator and the CM. The hook maintains the CM's connection to the World Simulator via a WebSocket, which uses TCP under the hood. When a new message is received from the World Simulator, `useWorldSim` will check if the message is information or relaying stress information, then will update the Redux state and return the received message.

useGaze The `useGaze` hook is called whenever the user's gaze (or the mouse pointer in our current version) moves. It finds all of the Elements that are currently in the user's gaze and stores them in the Redux state via `setElementsInGaze`. It is important to note that it detects Elements based off of the top level div within Elements. So if an Element has multiple smaller parts within, it will only detect the Element as the top level div within that Element.

Key and Mouse Actions (useKeyUp, useKeyDown, useMouseButtonUp, useMouseButtonDown) These hooks share the active key and mouse presses. These active presses are tracked to add and delete key presses within the `gazeAndKeys` data structure in the Redux state.

useMousePosition This hook shares the active mouse position. This is currently used to simulate the user's gaze.

4.11 Types

Section A Section is a collection of arbitrary but adjacent cells. Each cell in a Section is of the same type and priority. Sections specify what can go in a given area within a grid. A Section has the following parameters:

- `type: string` - The type of this Section. Only Widgets with matching types can be placed here

- `priority: number` - The priority of this Section. Controls which Widgets with certain priorities can be placed in this Section. If a Widget has this priority or above, and a matching type, it can be placed here
- `screen: Screen` - Which screen the Section is on: `pearce`, `center`, or `right`
- size (`w: number`, `h: number`) - Width and height of the Section
- location (`x: number`, `y: number`) - Top-left coordinates of the Section
- `WidgetIDs: string[]` - Array of Widgets that reside within this Section

Base Widget The base Widget is the foundational Widget for all the specific Widget types and contains these parameters:

- `screen: Screen` - Which screen the Widget is on: `pearce`, `center`, or `right`
- `type: string` - The type of this Widget. This Widget can only be placed in Sections with matching type
- `overrideType: boolean` - If true, the Widget has access to be placed anywhere on the screen, even if there is no room for it in matching Sections
- `Elements: Element[]` - Holds all Elements that reside within the Widget
- `maxAmount: number` - Not all Widgets can be duplicated, some can only have a single instance on the screen at a time, or multiple If set to 0, there can be multiple. If set to 1, there can only be 1
- size (`w: number`, `h: number`) - Width and height of Widget. `0` would mean there is no set size. This does not interfere with `useElementLocation`, it is just used to calculate `locationGrid`
- location (`x: number`, `y: number`) - Top-left coordinates. Aids in placement of Widget
- `useElementLocation: boolean` - Notifies CM that the location parameter of the Widget should not be used, and instead we should use the individual locations of Elements to define this Widget's working location (currently unused in our CM)
- `canOverlap: boolean` - If true, then any other Widget can overlap this Widget within the Section. If false, then no Widget or Element may overlap this Widget within the Section.
- `handledMessageIds: string[]` - The IDs of the messages that are handled by the Widget
- `padding: number` - Padding for the Widget (currently unused in our CM)
- `priority: number` - The priority level of the Widget
- `style: Properties` - The CSS styling of the Widget and is type checked by `csstype`
- `tags: string[]` - Help identify the specific Widget instance

Base Element The base Element is the foundational Element for all other Elements and the parameters are:

- `modality: Modality` - The modality of the Element
- size (`w: number`, `h: number`) - Width and height of the Element
- `WidgetId: string` - The UUID of the Widget that contains this Element
- `priority: number` - The priority level of the Element
- `collapsed: boolean` - Whether or not the Element should be collapsed
- `expirationIntervalMs: number` - The length of time before the Element should expire
- `expiration: string` – The exact time that the Element will expire if not interacted with. Updated when it is interacted with using `expirationIntervalMs` and the time of interaction.
- `onExpiration: string` - What the Element should do on expiration
- `escalate: boolean` - Escalate the Element or not
- `deescalate: boolean` - Deescalate the Element or not
- `interacted: boolean` - Whether or not the Element has been interacted with
- `canOverlap: boolean` - If true, then any Element can overlap this Element within its Widget (currently unused in our CM)
- `style: Properties` - The CSS style for the Element

Base Message The base message is the foundational message for all other messages and the parameters are:

- `conversationId` - Lets the CM know which conversation the message is in
- `priority` - Priority level of the message from 0 to 10, where 0 is the highest priority
- `tags` - Tags to identify what was affected by the message
- `kind` - Specifies which message it is
- `data` - Data for the message which changes based on the kind of the message
- `fulfilled` - Each message has a fulfilled boolean parameter that tells us whether it has been fully reacted to or not; this is primarily used in stress change handling to skip messages that have already been reacted to and no longer need to be seen.
- `read` - A boolean for when a message has been read by the CM

4.12 Future Work

4.12.1 Layout Preference Documents

More Stress Level LPDs There are only three stress levels, so in the future, a larger range of stress levels such as a number between the range of 0 and 1 could be used which means more stress level LPDs need to be added. This would allow the application to be adapted even more precisely to the pilot's amount of stress and guarantee a more adequate display of the relevant information.

Gaia Creates LPDs LPDs are statically created by UX/UI professionals in their current implementation, but in the future, Gaia could try to change what the LPDs return or even create the LPDs themselves based on what it has learned about users, the surrounding environment, and other factors that may impact LPDs.

More Modalities Currently, the LPDs only handle the visual modality, but in the future, they should be able to handle other modalities such as audio or haptic. In this case, the pilot wouldn't be overloaded by visual complexity in extreme situations but could use more of the human senses to carefully navigate through the situation.

4.12.2 Modality Assimilator

Custom geometric placement patterns within Sections Currently the MA places new Widgets vertically going down and going left to right within the Section. This may not be an optimal placement option for all Sections in various UIs, thus a system that allows more control over Widget placement would be helpful for developers. As a simple solution, Sections could contain some options for which directions to start a placement search: horizontally then vertically, left to right, bottom to top, etc.

4.12.3 Modality Monitor

Custom Loop Cycle Times Some users, like pilots, are trained professionals and may have better visual comprehension times than the average 100 ms. A custom loop cycle time could allow developers to fine tune exactly how long they want something to be looked at to deem it as "comprehended" and selected.

4.12.4 Modality Restrainer

Allow LPDs to assign modality ranges and boundaries. Currently the modality ranges and boundaries are hard-coded in the MR. Ideally the initial LPD would have user defined values to set these to in order to give developers more control over their designed UI.

Integrate more modalities As described, we currently only handle the visual modality. It would be interesting to see how we can combine and track more modalities at one time. While our current implementation easily handles a single modality, there may be more possibilities of restraining modalities in tandem.

Handle Widget clusters that exceed the boundaries Currently, Widget clusters that would exceed the modality measure boundaries and have no alternatives simply don't get placed. In a future version of the CM, there should be a different way to handle those situations to prevent cases where a rather important message would not be displayed because adding every integrated Widget would exceed the boundary. A possible way to solve this issue could be reducing or removing already deployed Widgets that aren't necessarily needed to give more room for a new or more urgent message.

4.12.5 Redux State

ACA and Ownship Status Currently, as it stands the status of ACAs and the Ownship is abstracted. The ACA header at the top contains the ACA weapon load, fuel level, and life status. This is not connected to the global state and, therefore does not receive or update based on incoming information. Adding the ACA status to the global state management would allow for the display to be truly dynamic as well as add options for updating the ACA status when messages would come to the Conversation Manager. This could allow for changing ACA weapon loads when a `RequestApprovalToAttack` message comes in, or dynamically draining fuel as ACA fuel information arrives.

4.12.6 Tagging

Multiple specified tags In the future, we would like to allow more than one type of specified tag in specified Widgets. This would allow specified Widgets to serve multiple real world instances at once without creating a new custom specified tag that needs to be added to message tags. This would be the most important limitation to eliminate, as serving multiple real world instances could introduce many more functionalities to our CM.

Eliminating other limitations All of the limitations present in our current tagging system are a result of not needing the limiting features within our current ownship-drone UI design. This is not optimal, as other UI designs may require the use of those missing features. We would like to eliminate all current limitations by updating the logic that our tagging system uses to identify specified tags and specifiers.

4.12.7 Types

More types In the future, more types could be added to accommodate more modalities, Widgets, Elements, messages, and so on that need to be accounted for.

5 General Future Work

While individual modules highlighted their own future work, there are numerous things we would like to change that do not apply to any single module. Below you will find most of the future work we would love to work more on, most of which were not needed for the current UI design or could not be completed within our time frame. The tasks are in no particular order.

Finishing the UI logically We would love to have the opportunity to finish the Widget Element Flow completely for the given ownship-drones UI. Some of the feature had to be hard-coded to meet the deadline. While we did use the Widget Element Flow, not all functionalities are implemented.

Two-Way Communication with World Simulator By implementing a two-way communication in the World Simulator, it could be used to create more realistic scenarios by providing proper feedback for user interactions. For instance, an approval for an attack by the BM might trigger a message in the World Simulator that informs about success or failure of that attack. The application that is using the World Simulator can then use this information to create a more dynamic UI.

Adding Machine Learning This is one of the highest priority tasks that we would love to tackle. When starting this project, we had many ideas of how some sort of artificial intelligence could act within the CM. From human centered reinforcement learning methods that utilize human feedback (RLHF) to replacing the LPD system with a constantly learning computer vision algorithm, there are many ways that AI and machine learning can enhance Theia. An addition like this would require a specified team and resources to conduct studies, but would be undoubtedly valuable not only to Theia, but to the Multi-Modal Adaptive UI framework that Gaia aims to provide.

Conduct and utilize more research Theia is almost completely homemade and invented by our team with little input outside of the weekly SRI meetings. This means that many implementations are biased to what our collective believes, and while we are all very bright individuals, it would be crucial to conduct and use actual research to ensure Theia is of professional quality. Research we would like to conduct includes measuring visual complexity of a UI, measuring auditory complexity of given sounds, measuring the complexity of the combination of visual and auditory modalities, the response times of users when using different modalities, and more. Existing research that we would like to explore more includes response time of trained professional versus the average human, how to physically tell if a user is cognitively overloaded, and more.

More comprehensive documentation While this documentation goes over most of our work and explains the main points and modules of Theia, there is still much that can be documented to create a

in depth developer experience. Our main priority would be to create a sort of guide in making a new UI within the Theia suite.

Complete/implement functionalities While we created a comprehensive and versatile suite, there are still some major functionalities that were not able to be implemented due to time constraints. A major feature yet to be implemented is the ability for the CM to integrate new Widgets even if the MR returns false. There are other functionalities, which are mostly documented within individual sections.

Make an intuitive interface for designing and editing new Widget Element Flows and multimodal adaptive UIs One of the major struggles we faced when creating the LPDs was readability and editability. If we wanted to find which Widgets were returned when or add an extra Element to a proposed Widget, it would be a journey to find which entry corresponded to what. A solution could be a UI — or at the very least a domain-specific language (DSL) — that organizes LPDs into a more visually appealing and intuitive way. Such an UI could consist of organized entries for each LPD file and may even include the option to view what a UI would look like at certain stages.

Add other modalities As mentioned throughout the documentation, we only were able to implement the support of the visual modality within Theia. While our system is flexible and could easily be changed to accommodate other modalities, we did not have time to add them. The first modality to add support for would be sound, which would add a more diverse range of options to communicate information to the user.

Polishing our code Being engineers, we have chanted the mantra "as long as it works" many times. However, we think we can make our code not only prettier, but more efficient. This would make it easier for slower hardware to run Theia UIs and would also let us improve the precision of some functionalities (i.e. Detecting Elements within gaze or placing Widgets).

Create modality complexity modules As described in other sections, the calculation of modality complexities, like visual complexity, is given to modules outside of the CM. This leaves us with having to simulate these calculations from within the CM. Implementing these complexity modules, either fully or via "best effort", would allow for a more adaptive CM.

List of Figures

Acronyms

ACA Uncrewed Air Combat Aircraft.

BM battle manager.

CM Conversation Manager.

LPD Layout Preference Document.

MA Modality Assimilator.

MM Modality Monitor.

MR Modality Restrainer.

MS Modality Selector.

UI user interface.

References

[1] Aulikki Hyrskykari, Howell Istance, and Stephen Vickers. "Gaze gestures or dwell-based interaction?" In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ETRA '12. Santa Barbara, California: Association for Computing Machinery, 2012, pp. 229–232. ISBN: 9781450312219. DOI: 10.1145/2168556.2168602.

[2] André Dettmann and Angelika Bullinger-Hoffmann. "Spatially distributed visual, auditory and multimodal warning signals – a comparison". In: *Proceedings of the Human Factors and Ergonomics Society Europe Chapter 2016 Annual Conference*. Vol. 4959. 2017. URL: https://www.hfes-europe.org/wp-content/uploads/2016/11/Dettmann2017.pdf (visited on 05/31/2024).

[3] Michael Domhardt. *Gestaltungsempfehlungen für touchscreenbasierte Benutzungsschnittstellen*. kassel university press, 2018. ISBN: 9783737604284. DOI: 10.19211/KUP9783737604291.

[4] Hesheng Liu et al. "Timing, Timing, Timing: Fast Decoding of Object Information from Intracranial Field Potentials in Human Visual Cortex". In: *Neuron* 62 (2009), pp. 281–90. DOI: 10.1016/j.neuron.2009.02.025.

Links

- **Theia / CM Source Code**

  https://git.tjdev.de/thi-sjsu-project/theia

  Mirror on GitHub: https://github.com/thi-sjsu-project/theia

- **World Simulator Source Code**

  https://git.tjdev.de/thi-sjsu-project/world-sim

  Mirror on GitHub: https://github.com/thi-sjsu-project/world-sim

- **Figma – High Fidelity Design**

  https://www.figma.com/design/xoQC4cosSylyZeWTZ5IQU5/Dev-Exchange?node-id=55-72&t=xjMEv8QJrgQV58ms-11

- **Figma – Components**

  https://www.figma.com/design/DpPo8USWg0j02gaKnmmWy5/Components?node-id=0-1&t=fHboEJR8QKC3AQaS-1

- **Figma – LPDs**

  https://www.figma.com/board/jxYp7BOn3T7M4Ao13ODS3V/Layout-Prefences-Documents?node-id=0-1&t=knRsS4OmDgAEHxNI-0

- **Final Demo Video**

  https://drive.google.com/file/d/11hBYSQ5Mdta77CsacoUwENh9F9rT_7Ju/view

- **Final Presentation Slides**

  https://drive.google.com/file/d/1lq7rXQkzqZjBlkQMG3zBFhbiomNkM_j1/view