

Spiegel AI

Datenverarbeitung in der Technik

Gruppe 5



Leon Kranner
Marco Kuner
David Vollmer
Marcel Wagner

leon.kranner@st.oth-regensburg.de
marco.kuner@st.oth-regensburg.de
david1.vollmer@st.oth-regensburg.de
marcel.wagner@st.oth-regensburg.de

15. Juli 2024

Inhaltsverzeichnis

Einleitung	3
1 Hardware	4
1.1 Komponenten	4
1.2 Auswahlkriterien	6
1.3 Installation	6
1.4 Konfiguration	7
1.4.1 Konfiguration des Raspberry Pi	7
2 Rahmen	9
2.1 Rahmen Konstruktion	9
2.2 Spiegel	11
2.3 Aufbau	11
3 Display	13
3.1 Display Aufbau	13
3.2 Widgets	14
3.2.1 Widgets Aktualisierung	14
3.2.2 Termine Widget	15
3.2.3 Kalender Widget	16
3.2.4 Wettervorhersage Widget	18
3.2.5 Notizen Widget	19
3.2.6 Uhrzeit Widget	21
3.2.7 Verkehrsinformation	22
3.2.8 Schlagzeilen	23
3.2.9 Tankstellen	24
3.3 Test Verfahren	26
4 Spiegel AI Remote	28
4.1 Die Flutter™ SDK	28
4.2 Funktionen	29
4.2.1 Remote View	29
4.2.2 Widgets View	30
4.2.3 Profile View	30
4.3 Implementierung	30
4.3.1 Websocket	31
4.3.2 Remote	32
4.3.3 Widgets	33

4.3.4	Profile	33
4.3.5	Sonstige Implementierungen	33
5	Gesichtserkennung	35
5.1	Einleitung	35
5.1.1	Projektziel	35
5.1.2	Bedeutung der Gesichtserkennung	35
5.2	Recherche und Anfangsphase	35
5.2.1	Grundlagen der Gesichtserkennung	35
5.2.2	Vergleich von Methoden	36
5.3	Erste Implementierung mit Haar-Cascades	36
5.3.1	Haar-Cascade-Ansatz	36
5.3.2	Vorteile und Nachteile	37
5.3.3	Ergebnisse der Tests	38
5.4	Umstieg auf Dlib für höhere Präzision	38
5.4.1	Wechsel zu Dlib	38
5.4.2	Technologien: HOG und 68-Facial-Landmarks	38
5.4.3	Implementierung und Herausforderungen	39
5.4.4	Verbesserung der Performance	40
5.5	Feature Extraction und Matching	41
5.5.1	Feature Extraction	41
5.5.2	Vergleich der GesichtseMBEDDINGS	42
5.6	Speichern und Verwalten der Profile	42
5.6.1	Speichern der Profile	42
5.6.2	Verbindung zum Websocket	42
5.7	Schlussfolgerung	43
5.7.1	Ausblick	43
5.7.2	Zusammenfassung	44
6	Schnittstellen	45
6.1	Websocket zwischen Remote App und Display	45
6.2	Senden und Empfangen von Profilen	46
6.3	Synchronisation der Profile	46
6.4	Dynamische Widget Anordnung	47
7	Ergebnisse	49
7.1	Zusammenfassung der Ergebnisse	49
7.1.1	Erreichte Ziele	49
7.1.2	Herausforderungen	50
7.1.3	Zukünftige Arbeiten	50
7.1.4	Betriebssetzung	50
	Stundenliste	51

Einleitung

In der Einleitung stellen wir das Projekt **Spiegel AI** vor. Wir beschreiben die Zielsetzung des Projekts, die Motivation und den allgemeinen Aufbau der Dokumentation. Zudem geben wir einen Überblick über die eingesetzte Hardware und Software sowie die geplanten Anwendungsbereiche.

Zielsetzung

Unser Ziel ist es, einen intelligenten Spiegel zu entwickeln, der durch seine benutzerfreundliche und interaktive Oberfläche den Alltag der Nutzer erleichtert. Er soll personalisierte Informationen bereitstellen und somit erkennen, welche Personen vor dem Spiegel sind. Der Nutzer soll durch die zugehörige App das Layout und die Widgets nach Belieben verändern können. Zudem soll der Spiegel automatisch neue Personen erkennen und ein neues Profil erstellen.

Motivation

Die Motivation hinter der Entwicklung eines SMART Mirrors liegt in der Verbesserung des täglichen Lebens durch Effizienzsteigerung und Zeiteinsparung. Nutzer des intelligenten Spiegel sollen am Spiegel bereits die wichtigsten Informationen für den Tag bereitgestellt bekommen. So können Nutzer z.B. während ihrer Morgenroutine gleichzeitig die wichtigsten Informationen für den Tag ablesen.

Überblick

Im ersten Kapitel gehen wir zunächst auf den Aufbau des Rahmens ein. Danach werden wir die Hardware des Projektes genauer ansehen. Dabei werden wir auf die Komponenten und Installation der Hardware den Fokus legen. Im 3. Abschnitt werden wir auf das Display und ihre Widgets eingehen. Des weiteren werden wir die Spiegel AI Remote App beleuchten. Dabei werden wir auf die Funktionen, Implementierung und die Testmöglichkeiten eingehen. Die dazugehörige Gesichtserkennung wird im 5. Kapitel beschrieben. Im nächsten Abschnitt werden wir die Schnittstelle zwischen App, Raspberry und Gesichtserkennung erläutern. Im letzten Abschnitt werden wir auf unsere Ergebnisse eingehen und ein Fazit daraus schließen.

1 | Hardware

erarbeitet von Leon Kranner und Marcel Wagner

In diesem Kapitel beschreiben wir die Hardware Komponenten, die für das Projekt **Spiegel AI** verwendet wurden. Wir gehen auf die Auswahlkriterien, die Installation und die Konfiguration der Hardware ein.

1.1 Komponenten

Im folgenden Abschnitt, werden nun die verwendeten Hardware Komponenten beschrieben und wofür diese genutzt werden.

Raspberry Pi 3 Model B

Der Raspberry Pi 3 Model B ist das Herzstück des Smartmirrors. Für die Speicherung des Betriebssystems und Daten wird eine 64 GB microSD-Karte verwendet, die ausreichend Platz für alle benötigten Software Anwendungen bietet. Nachfolgend kann der Raspberry Pi entnommen werden



Abbildung 1.1: Raspberry Pi Model B
Quelle: siehe **raspberry_pi**

Logitech Kamera zur Gesichtserkennung

Für die Gesichtserkennung wird eine Logitech Kamera verwendet, die eine hohe Bildqualität und eine zuverlässige Leistung bietet. Diese Kamera ist über USB mit dem Raspberry Pi verbunden und ermöglicht es, Benutzer zu erkennen und auf sie zugeschnittene Informationen anzuzeigen. In der nachfolgenden Abbildung kann die Kamera entnommen werden.



Abbildung 1.2: Logitech Kamera
Quelle: siehe **logitech_camera**

Monitor

Der Dell Monitor dient als Display für den Smartmirror. Er ist über den integrierten VGA Anschluss mithilfe eines Adapters mit dem Raspberry PI verbunden und ist in das Spiegelgehäuse integriert. Der Monitor wird in der Nachfolgenden Abbildung ersichtlich.



Abbildung 1.3: Dell Monitor
Quelle: siehe **dell_monitor**

WLAN Stick

Ein WLAN Stick wird verwendet, um den Raspberry Pi mit dem Internet zu verbinden. Dies ermöglicht die Nutzung von Online Diensten und die Kommunikation mit anderen Geräten im Netzwerk. Dies ist essentiell um die Funktionsweise des Smart Mirrors zu gewährleisten.

Smartphone

Ein Smartphone dient als mobile Schnittstelle für den Smartmirror. Über eine App können Benutzer Einstellungen vornehmen. Eine genaue beschreibung dieser Schnittstelle erfolgt im Kapitel 4.

1.2 Auswahlkriterien

Die Auswahl der Hardware Komponenten basierte auf mehrere Kriterien, diese werden nun im folgenden genauer beschrieben:

- **Kompatibilität:** Alle Komponenten mussten kompatibel miteinander und mit der Software Plattform des Smartmirrors sein.
- **Leistung:** Der Raspberry Pi 3 Model B wurde wegen seiner ausreichenden Rechenleistung und Energieeffizienz gewählt.
- **Bildqualität:** Die Logitech Kamera wurde aufgrund ihrer hohen Auflösung und zuverlässigen Gesichtserkennung ausgewählt.
- **Displayqualität:** Der Dell Monitor bietet eine klare und scharfe Anzeige, was für die visuelle Darstellung der Informationen wichtig ist.
- **Konnektivität:** Der WLAN Stick sorgt für eine stabile Internetverbindung, was für die Nutzung von Online Diensten unerlässlich ist.
- **Benutzerfreundlichkeit:** Das Smartphone als mobile Schnittstelle erleichtert die Interaktion und die Anpassung der Einstellungen durch den Benutzer.

1.3 Installation

In diesem Abschnitt werden nun die einzelnen schritte beachtet, welche bei der installation der unterschiedlichen Hardware Komponenten vorgegangen sind.

Raspberry Pi 3 Model B

- Die microSD-Karte wurde formatiert und das Betriebssystem wurde installiert.
- Der Raspberry Pi wurde in das Gehäuse eingebaut und mit dem Monitor über den VGA auf HDMI Adapter verbunden.
- Die Stromversorgung wurde angeschlossen und der Raspberry Pi wurde gestartet.

Logitech Kamera

- Die Kamera wurde über USB mit dem Raspberry Pi verbunden.

Monitor

- Der Monitor wurde in das Spiegelgehäuse integriert und mit dem Raspberry Pi verbunden.

WLAN Stick

- Der WLAN Stick wurde in einen freien USB Port des Raspberry Pi eingesteckt.

Smartphone

- Eine spezielle App wurde auf dem Smartphone installiert, um die Kommunikation mit dem Smartmirror zu ermöglichen.
- Das Smartphone wurde mit dem WLAN des Raspberry Pi verbunden

1.4 Konfiguration

Im folgenden Abschnitt wird nun beschrieben wie die unterschiedlichen Hardware Komponenten Konfiguriert wurden.

Logitech Kamera: Die Gesichtserkennungssoftware wurde installiert und entsprechend kalibriert. Desweiteren wurden die Kameraeinstellungen angepasst, um eine optimale Erkennungsrate zu gewährleisten.

Monitor (Dell): Die Bildschirmeinstellungen wurden so konfiguriert, dass der Monitor im Energiesparmodus bleibt, wenn der Smartmirror nicht verwendet wird. Des Weiteren wurde die Anzeigesoftware für den Smartmirror installiert und konfiguriert.

Smartphone: Die App auf dem Smartphone wurde installiert.

1.4.1 Konfiguration des Raspberry Pi

Erarbeitet von David Vollmer.

Die Konfiguration des Raspberry Pi ist im Vergleich zu der der anderen Hardware recht umfangreich. Da ich zuvor noch nie mit einem Raspberry Pi gearbeitet habe, sind in diesem Prozess einige Probleme aufgetreten. Das erste Problem kam schon beim Versuch, ein Betriebssystem zu installieren. Ich habe zwar schon oft Windows und Linux-Distributionen installiert, bis jetzt jedoch nur auf Laptops und Desktop-PCs. Deswegen ging ich davon aus, dass Raspberry Pi OS, welches ein auf Debian basiertes Betriebssystem ist, via USB eingerichtet wird. **raspi_os** Es stellte sich heraus, dass sowohl die Speicherung als auch die Installation mit Hilfe einer microSD Speicherkarte ausgeführt wird. Nachdem das Betriebssystem aufgesetzt wurde, kam eine weitere Schwierigkeit hinzu. Die Verbindung zum Internet war zwar durch ein Ethernet-Kabel möglich, kabellos funktionierte sie aber nicht. Laut Spezifikation ist der Raspberry Pi 3 Model B mit einem drahtlosen Kommunikationschip, inklusive WLAN, ausgestattet. **raspberry_pi** Der Versuch, dieses Problem durch Updates der Netzwerktreiber zu beheben scheiterte. Daraufhin gingen wir von einem Hardwaredefekt aus. Da der Spiegel AI ohne Internetzugriff nicht funktionieren kann und das Anschließen eines Ethernet-Kabels im Betrieb unpraktikabel ist, entschieden wir uns dazu, die drahtlose Verbindung mithilfe eines USB WLAN-Sticks herzustellen. Um diesen zum Laufen zu bekommen, mussten wir Treiber suchen, welche auf dem Betriebssystem funktionieren. Bei den ersten zwei Treibern, die ich versuchte zu installieren, gab es Dependency-Errors. Das bedeutet, dass zur Installation benötigte Pakete fehlten. In beiden Fällen traten diese Fehler nach über 20 Minuten des Installationsprozesses auf. Der dritte Versuch eines Treiber-Downloads war nach etwa 40 Minuten Wartezeit erfolgreich und der Raspberry

Pi konnte erfolgreich auf das Internet zugreifen. Um nun auf die Software des Spiegel AI zuzugreifen, klonete ich das git-Repository des Projekts. Im nächsten Schritt war es nötig, Bibliotheken für die verwendeten Python-Programme herunterzuladen. Diese Libraries waren unter anderem benötigt für den Websocket und für die Gesichtserkennung mit OpenCV. Hierbei war meine Erfahrung mit Linux-Distributionen, unter anderem auch Debian, sehr hilfreich, da ich mit dem Suchen und Installieren korrekter Pakete vertraut bin. Außerdem fiel mir die Navigation im Terminal und das Nutzen gängiger Shell-Befehle sehr leicht, was die Arbeitszeit in diesem Schritt vergleichsweise kurz hielt. Ein zukünftiger Schritt, welcher für die Verwendung des Spiegel AI nützlich sein kann, ist das automatische Starten der nötigen Applikationen. Hierbei müssen der Websocket-Server, der HTTP-Server und die Gesichtserkennung im Hintergrund und die Spiegelanzeige im Vordergrund gestartet werden. Erzielt werden kann dies, indem man Service-Dateien beschreibt, welche beispielsweise BASH-Skripte ausführen lassen. Nachdem die Services eingeschaltet sind, führen sie die Programme nach dem Hochfahren des Systems aus. Der Grund, weshalb dies noch nicht konfiguriert ist, liegt an der Volatilität des Gesichtserkennungsprogramm. Dieses tendiert nämlich dazu, abzustürzen. Das bedeutet, dass oft Intervention mit Eingabeperipherie nötig ist, was den Nutzen des automatischen Starts verringert. Sobald dieses Problem behoben ist, führt diese Änderung jedoch zu einer deutlichen Verbesserung des Nutzererlebnisses, da der Spiegel AI dann ohne Eingabe-Hardware vollständig nutzbar ist.

2 | Rahmen

Erarbeitet von: Leon Kranner und Marcel Wagner

In diesem Kapitel wird die Vorgehensweise für die Rahmenkonstruktion des Smart Mirrors beschrieben.

2.1 Rahmen Konstruktion

1. **Planung:** Zu Beginn der Planung haben wir die Abmessungen des Displays genau vermessen, um den benötigten Platz für die Hardwarekomponenten und das Gehäuse bestimmen zu können. Es war wichtig, genügend Raum für den Raspberry Pi, die Kamera für die Gesichtserkennung, den WLAN-Stick und eventuelle Kabelverbindungen einzuplanen.

Nachdem wir die Abmessungen des Displays und den Platzbedarf für die Hardware ermittelt hatten, konnten wir den Maßstab und die endgültige Größe des Rahmens festlegen. Unser Ziel war es, einen Rahmen zu konstruieren, der sowohl funktional als auch ästhetisch ansprechend ist.

Im nächsten Schritt besuchten wir den Baumarkt, um das passende Material für unseren Rahmen auszuwählen. Wir entschieden uns schließlich für Leimholz, da es robust und gut zu verarbeiten ist.

Ursprünglich war geplant, den Rahmen mit Schrauben zusammenzubauen. Nach weiteren Überlegungen und Tests fanden wir jedoch, dass die geschraubte Konstruktion unseren ästhetischen Ansprüchen nicht gerecht wurde. Daher entschieden wir uns, das Leimholz zu verwenden und die Teile zu verleimen. Diese Methode bot uns eine stabilere und sauberere Verbindung der Rahmenteile.

2. **Zuschneiden des Leimholzes:** Zu Beginn der Konstruktion wurde das Leimholz auf die gewünschte Länge zugeschnitten. Dabei war es wichtig, präzise Maße zu verwenden, um sicherzustellen, dass alle Teile des Rahmens passgenau zueinander stehen. Für den Zuschnitt wurde eine Kreissäge verwendet, um gerade und saubere Schnitte zu erzielen.
3. **Gehrungsschnitt der Kanten:** Um eine ästhetisch ansprechende und stabile Verbindung der Rahmenteile zu gewährleisten, wurden die Kanten des Holzes auf Gehrung geschnitten. Hierzu wurde ein Gehrungssägeblatt in einem Winkel von 45 Grad eingestellt. Dieser Schritt ist entscheidend, da die Gehrungsschnitte die Stoßkanten der Rahmenteile so ausrichten, dass sie sich nahtlos und formschlüssig verbinden lassen.

4. **Verleimung der Rahmenteile:** Nach dem Gehrungsschnitt wurden die Kanten mit Holzleim bestrichen. Es ist wichtig, den Leim gleichmäßig aufzutragen, um eine vollständige und feste Verbindung zu gewährleisten. Die Rahmenteile wurden dann sorgfältig zusammengesetzt, wobei darauf geachtet wurde, dass die Gehrungsschnitte exakt aufeinanderpassen.

Zur Sicherstellung einer stabilen Verbindung wurden die verleimten Rahmenteile mit Schraubzwingen fixiert. Die Schraubzwingen wurden gleichmäßig verteilt angezogen, um den Druck auf alle Teile des Rahmens zu verteilen und Verformungen zu vermeiden. Die Rahmenkonstruktion wurde dann für die empfohlene Zeitspanne in den Schraubzwingen belassen, um eine vollständige Aushärtung des Leims zu gewährleisten. In der nachfolgenden Abbildung kann der Spiegelrahmen entnommen werden.

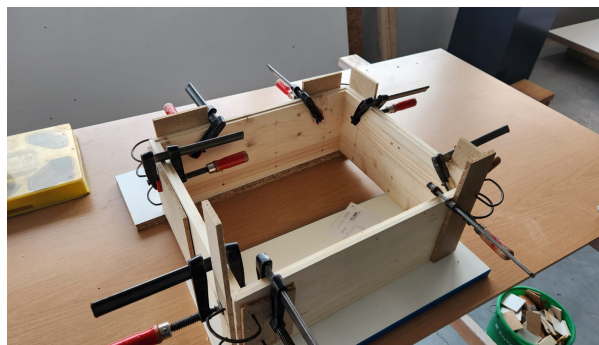


Abbildung 2.1: Rahmen wurde geleimt
Quelle: eigene Darstellung

5. **Befestigung der Halteleiste:** Nach der Aushärtung des Leims wurde eine Halteleiste angebracht. Diese Leiste dient dazu, den Spiegel sicher im Rahmen zu fixieren. Die Halteleiste wurde präzise vermessen und zugeschnitten, um optimal in den Rahmen zu passen. Sie wurde mit Holzleim und zusätzlichen Schrauben befestigt, um eine dauerhafte und sichere Fixierung zu gewährleisten.



Abbildung 2.2: Fertiggestellter Rahmen
Quelle: eigene Darstellung

2.2 Spiegel

1. **Auswahl des Materials:** Ursprünglich hatten wir geplant, eine Glasscheibe für den Spiegel des Smart Mirrors zu verwenden. Da jedoch für unsere Zwecke eine Glasscheibe nicht erforderlich war und wir bereits eine Plexiglasscheibe zur Verfügung hatten, entschieden wir uns, diese zu verwenden.
2. **Zuschnitt und Vorbereitung:** Zunächst haben wir anhand des Rahmens abgemessen, wie groß der Spiegel sein muss, und eine Toleranz von 1 cm festgelegt. Mit Hilfe einer Stichsäge haben wir das Plexiglas auf Maß geschnitten. Anschließend haben wir die Kanten mit einer Feile entgratet und mit Schleifpapier die unebenen Stellen beseitigt.
3. **Bohrungen und Befestigung:** Um das Plexiglas später am Rahmen befestigen zu können, haben wir an den Seiten des Plexiglasses Bohrungslöcher gebohrt.
4. **Anbringen der Spiegelfolie:** Zunächst haben wir eine Spiegelglasfolie auf das Plexiglas aufgebracht. Beim Testen mit dem Bildschirm stellten wir jedoch fest, dass die Helligkeit des Displays nicht ausreichte, um den Bildschirm durch das Plexiglas und die Spiegelglasfolie zu erkennen. Aufgrund der unzureichenden Helligkeit des Displays haben wir uns für eine Sonnenschutzfolie entschieden. Diese Folie spiegelt immer noch, ermöglichte dennoch eine bessere Sicht.

2.3 Aufbau

1. **Befestigung des Plexiglasses:** Nachdem die geeignete Folie angebracht war, haben wir das Plexiglas mit Schrauben fixiert. Diese Maßnahme gewährleistete eine stabile und sichere Befestigung.
2. **Montage des Bildschirms:** Der erste Ansatz zur Befestigung des Bildschirms bestand darin, ein Draht von der linken zur rechten Seite des Spiegels zu spannen. Diese Lösung fixiert den Bildschirm in der gewünschten Position und verhinderte ein Verrutschen. Aufgrund der Anfälligkeit für Spielräume und der Tatsache, dass der Bildschirm nicht immer fest mit dem Plexiglas verbunden ist, entschieden wir uns jedoch, den Bildschirm mit eng anliegenden Haltern am Bilderrahmen zu fixieren. Diese Konstruktion gewährleistet eine klare und stabile Befestigung des Bildschirminhalts durch die Spiegelfolie.
3. **Integration der Elektronik:** An der Seite des Rahmens haben wir den Raspberry Pi befestigt und den Bildschirm angeschlossen. Der Raspberry Pi steuert die gesamte Funktionalität des Smart Mirrors. In der nachfolgenden Abbildung kann der befestigte Raspberry Pi an dem Spiegerrahmen entnommen werden.



Abbildung 2.3: Raspberry Pi am Rahmen befestigt
Quelle: eigene Darstellung

4. **Zusätzliche Komponenten:** Wir haben zudem eine Kamera für die Gesichtserkennung (Eggo AI) angeschlossen. Diese Kamera ermöglicht personalisierte Funktionen und verbessert die Benutzererfahrung. Zusätzlich haben wir einen WLAN-Stick integriert, damit die Widgets Daten aus dem Internet abrufen und mit der App kommunizieren können

Der nachfolgende Bild zeigt den fertigen Smart Mirror.



Abbildung 2.4: Spiegel AI
Quelle: eigene Darstellung

3 | Display

In diesem Kapitel gehen wir auf das Display ein, das im **Spiegel AI** Projekt verwendet wird. Wir beschreiben den Display Aufbau, die Widgets und die Testverfahren.

3.1 Display Aufbau

Erarbeitet von: Leon Kranner

Für das Display unseres Smart Mirrors haben wir verschiedene Technologien und Tools verwendet, um eine benutzerfreundliche und flexible Benutzeroberfläche zu gestalten. Die Logik der Widgets und deren Kommunikation mit der App wurden in JavaScript implementiert. HTML wurde für das Displaydesign und CSS für die Gestaltung der Widgets verwendet. Unsere Entwicklungsumgebung war Visual Studio Code.

Der Displayaufbau folgt einem 3x3-Raster, wobei das mittlere Feld frei bleibt, um eine klare Spiegelreflexion zu ermöglichen. Die umgebenden Felder enthalten verschiedene Widgets. Aktuell stehen acht Widgets zur Auswahl, die dynamisch angeordnet werden können. Das bedeutet, dass der Benutzer die Widgets nach Belieben anordnen oder deaktivieren kann.

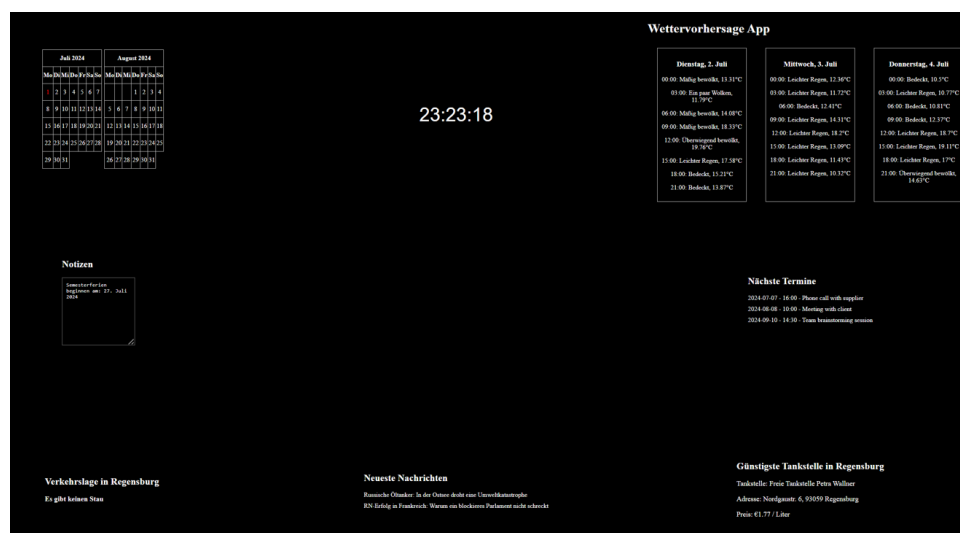


Abbildung 3.1: Display Layout
Quelle: eigene Darstellung

Beim Erstellen des Layouts für den Display kam es zu Komplikationen. Bei den ersten

Versuchen, ein 3x3 Raster zu erstellen, waren die einzelnen Felder dynamisch. Wenn das Displayteam also neue Widgets hinzugefügt hat, hat sich das jeweilige Feld an die Größe des Widgets angepasst. Dies ist natürlich nicht gewollt, denn die Felder sollten am besten eine statische Größe haben. Um dieses Problem zu lösen, haben wir die Felder mithilfe von CSS statisch gemacht. Dadurch konnten wir sicherstellen, dass alle Felder die gleiche Größe behalten, unabhängig von der Größe der Widgets, die sie enthalten.

Ein weiteres Problem war, dass einige Widgets an den Rändern abgeschnitten wurden, weil der Bildschirm größer war als der sichtbare Bereich des Rahmens. Um sicherzustellen, dass alle Widgets vollständig sichtbar sind, haben wir einen kleinen Abstand um alle Widgets herum eingebaut. Dadurch konnten wir garantieren, dass alle Widgets im Spiegel perfekt dargestellt werden.

3.2 Widgets

In diesem Abschnitt wird auf die einzelnen Widgets genauer eingegangen und zudem erläutert, wie und wann die einzelnen Widgets aktualisiert werden.

3.2.1 Widgets Aktualisierung

Erarbeitet von: Leon Kranner

Alle Widgets müssen in regelmäßiger Zeit aktualisiert werden. Wie regelmäßig dies der Fall ist, hängt vom Widget ab. Dies regelt die `Timer.js` Datei. Hier werden alle Widgets aktualisiert und die neusten Daten geladen. In diesem Abschnitt ist eine kleine Übersicht wie regelmäßig die Widgets geladen werden:

1. **Uhr** Jede Sekunde
2. **Termine:** Alle 60 Sekunden
3. **Kalender:** Jeden Tag um Mitternacht
4. **Wettervorhersage:** Alle 3 Stunden
5. **Schlagzeilen:** Alle 10 Sekunden
6. **Tankstellen:** Alle 10 Sekunden
7. **Verkehrsinformationen:** Alle 5 Minuten

3.2.2 Termine Widget

Erarbeitet von: Leon Kranner

Das Termin-Widget ist eine JavaScript-Anwendung, die dazu dient, die nächsten drei anstehenden Termine auf dem Display anzuzeigen. Dieses Widget filtert die Termine, sortiert sie nach Datum und stellt sicher, dass nur zukünftige Termine angezeigt werden. Die Termine werden dynamisch in eine HTML-Liste eingefügt.

In Zukunft soll die App die nächsten Termine aus der Smartphone Kalender App abrufen und den Raspberry versenden. Somit werden die aktuellen Termine automatisch geladen und müssen nicht manuell eingetragen werden.

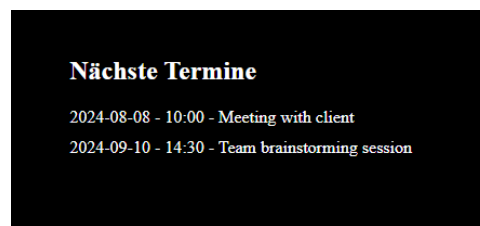


Abbildung 3.2: Termine Widget
Quelle: eigene Darstellung

Die Hauptfunktion des Widgets ist `loadAppointments()`, die beim Laden der Seite ausgeführt wird. Die Funktion filtert, sortiert und rendert die Termine auf dem Display.

Detaillierte Beschreibung der Funktion `loadAppointments`

Die Funktion `loadAppointments()` führt folgende Schritte aus:

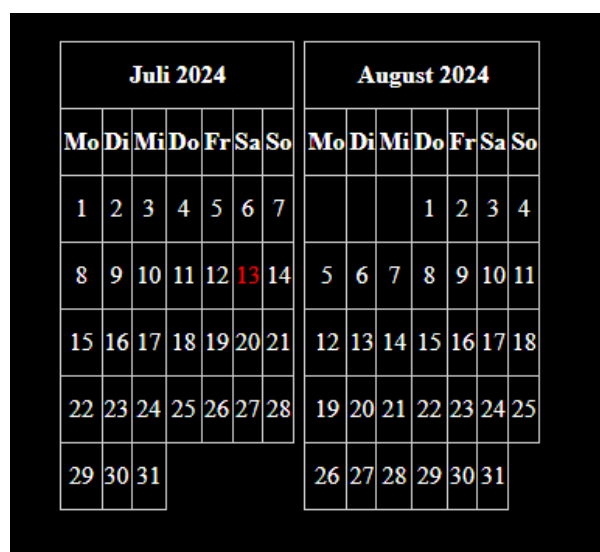
1. **Terminliste initialisieren:** Ein Array von Terminen (`appointments`) wird definiert, das Datum, Uhrzeit und Beschreibung jedes Termins enthält.
2. **Heutiges Datum ermitteln:** Das heutige Datum (`today`) wird mithilfe des `Date`-Objekts ermittelt.
3. **HTML-Elemente vorbereiten:** Das HTML-Element mit der ID `appointmentsList` wird selektiert und dessen Inhalt wird geleert.
4. **Filtern der zukünftigen Termine:** Es werden nur die Termine gefiltert, deren Datum gleich oder später als das heutige Datum ist.
5. **Sortieren der Termine:** Die gefilterten Termine werden nach Datum sortiert.
6. **Hinzufügen der Termine zur Liste:** Für jeden gefilterten und sortierten Termin wird ein `li`-Element erstellt, das die Termininformationen enthält. Diese `li`-Elemente werden zur `appointmentsList` hinzugefügt.
7. **Seitenladezustand:** Die Funktion wird beim ersten mal in der `appointment.js` Datei aufgerufen. Da sich Termine jederzeit ändern können, soll in regelmäßigen Abständen das Termin Widget aktualisiert werden.

Das Termin-Widget bietet eine einfache und effektive Lösung zur Anzeige bevorstehender Termine. Es lässt sich leicht in bestehende Webseiten integrieren und an individuelle Bedürfnisse anpassen.

3.2.3 Kalender Widget

Erarbeitet von Leon Kranner

Das Kalender-Widget ist eine JavaScript-Anwendung, die den aktuellen und den nächsten Monat in einem Kalender nebeneinander anzeigt. Der aktuelle Tag wird dabei hervorgehoben.



Juli 2024							August 2024						
Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So
1	2	3	4	5	6	7				1	2	3	4
8	9	10	11	12	13	14	5	6	7	8	9	10	11
15	16	17	18	19	20	21	12	13	14	15	16	17	18
22	23	24	25	26	27	28	19	20	21	22	23	24	25
29	30	31					26	27	28	29	30	31	

Abbildung 3.3: Kalender Widget
Quelle: eigene Darstellung

Ursprünglich sollte der aktuelle Tag durch einen roten Ring markiert werden. Jedoch gab es Probleme bei der Formatierung, da die Zahl nicht mittig im Ring angezeigt wurde, sondern unübersichtlich an der Seite des Feldes., Gelöst wurde dieses Problem, in dem wir einen anderen Ansatz probiert haben. Der aktuelle Tag wird jetzt rot markiert und nicht eingekreist.

Ein weiteres Problem war die generelle Formatierung des Kalenders: Die Zahlen haben teilweise nicht mit den Wochentag übereingestimmt oder die Tage wurden leicht verschoben und haben nicht mehr gepasst. Außerdem hat der erste Tag des Monats immer mit einem Montag begonnen. Nach einer längeren Bugfixing-Session und längerer Überarbeitung konnten die Fehler gelöst werden.

Zunächst wurden die Monate untereinander angezeigt. Da wir das Programm noch nicht auf dem Raspberry und den Bildschirm übertragen haben, konnten wir noch nicht testen, ob das Layout passt. Beim Testen viel auf, dass der zweite Monat abgeschnitten wurde, weshalb wir uns dazu entschieden haben, die Monate nebeneinander anzuzeigen.

Die Hauptfunktion des Widgets ist `createCalendarWidget()`, die beim Laden der Seite ausgeführt wird. Diese Funktion generiert die Kalender für den aktuellen und den

nächsten Monat und zeigt sie in definierten Containern nebeneinander an.

Detaillierte Beschreibung der Funktion `createCalendarWidget`

Die Funktion `createCalendarWidget()` führt folgende Schritte aus:

1. Elemente vorbereiten:

- Das HTML-Element mit der ID `calendarWidget` wird selektiert.
- Die HTML-Elemente für den aktuellen Monat (`currentMonthCalendar`) und den nächsten Monat (`nextMonthCalendar`) werden selektiert.

2. Kalender generieren:

Die Funktion `generateCalendar()` wird verwendet, um den HTML-Code für die Kalender zu generieren und in die entsprechenden Container einzufügen.

3. Kalenderfunktion `generateCalendar()`:

- Berechnung des ersten Tages des Monats und Anzahl der Tage im Monat.
- Aufbau des HTML-Codes für den Kalender mit den Monats- und Wochentagsnamen.
- Hinzufügen der Tage in die Tabelle, wobei der aktuelle Tag rot markiert wird.

4. Anzeige des Kalenders:

Die Kalender für den aktuellen Monat und den nächsten Monat werden in den jeweiligen Containern angezeigt.

5. Seitenladezustand:

Die Funktion wird beim ersten mal in der `Calendar.js` Datei aufgerufen. Sollte sich das Profil bzw. der Zustand des Displays nicht ändern, wird jeden Tag um 24 Uhr der Kalender aktualisiert, um den nächsten Tag bzw. den nächsten Monat anzeigen zu können.

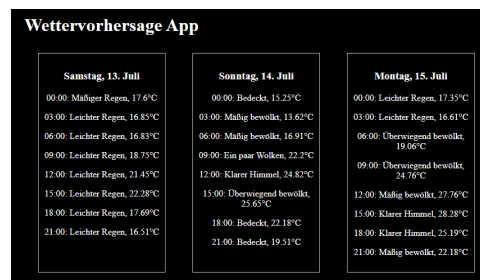
Das Kalender-Widget bietet eine einfache und effektive Lösung zur Anzeige der aktuellen und kommenden Monatskalender. Es lässt sich leicht in den Smart Mirror integrieren und an individuelle Bedürfnisse anpassen. Die Hervorhebung des aktuellen Tages erleichtert die Orientierung im Kalender.

3.2.4 Wettervorhersage Widget

Erarbeitet von: Leon Kranner

Das Wettervorhersage-Widget ist eine JavaScript-Anwendung, die mithilfe der Open-WeatherMap API die Wettervorhersage für die nächsten drei Tage für eine bestimmte Stadt anzeigt. Das Widget lädt die Wetterdaten über eine API und zeigt die Vorhersage für den aktuellen Tag sowie die nächsten zwei Tage an.

In Zukunft soll man in der App den Standort auswählen können, damit der Nutzer entweder den Standort des Spiegels angeben kann oder z.B. auch das Wetter bei der OTH Regensburg ansehen können.



Samstag, 13. Juli	Sonntag, 14. Juli	Montag, 15. Juli
00:00: Mäßiger Regen, 17,6°C	00:00: Bedeckt, 15,25°C	00:00: Leichter Regen, 17,35°C
03:00: Leichter Regen, 16,85°C	03:00: Mäßig bewölkt, 13,62°C	03:00: Leichter Regen, 16,61°C
06:00: Leichter Regen, 16,83°C	06:00: Mäßig bewölkt, 16,91°C	06:00: Überwiegend bewölkt, 19,06°C
09:00: Leichter Regen, 18,75°C	09:00: Ein paar Wolken, 22,2°C	09:00: Überwiegend bewölkt, 24,76°C
12:00: Leichter Regen, 21,45°C	12:00: Klarer Himmel, 24,82°C	12:00: Mäßig bewölkt, 27,76°C
15:00: Leichter Regen, 22,28°C	15:00: Überwiegend bewölkt, 25,65°C	15:00: Klarer Himmel, 28,28°C
18:00: Leichter Regen, 17,69°C	18:00: Bedeckt, 22,18°C	18:00: Klarer Himmel, 25,19°C
21:00: Leichter Regen, 16,51°C	21:00: Bedeckt, 19,31°C	21:00: Mäßig bewölkt, 22,18°C

Abbildung 3.4: Wettervorhersage Widget
Quelle: eigene Darstellung

Die Hauptfunktionen des Widgets sind `getForecast()` und `displayForecast(data)`. `getForecast()` ruft die Wetterdaten von der API ab, während `displayForecast(data)` die Daten verarbeitet und auf der Webseite anzeigt.

Detaillierte Beschreibung der Funktion `getForecast`

Die Funktion `getForecast()` führt folgende Schritte aus:

1. **API-Schlüssel und Stadt definieren:** Der API-Schlüssel und die Stadt, für die die Wettervorhersage abgerufen werden soll, werden definiert.
2. **URL für den API-Aufruf erstellen:** Eine URL für den API-Aufruf wird mit dem API-Schlüssel und der Stadt erstellt.
3. **API-Aufruf durchführen:** Ein Fetch-Aufruf wird durchgeführt, um die Wetterdaten von der OpenWeatherMap API abzurufen.
4. **Fehlerbehandlung:** Falls die Stadt nicht gefunden wird oder ein anderer Fehler auftritt, wird eine Fehlermeldung angezeigt.
5. **Daten verarbeiten:** Die erhaltenen Daten werden an die Funktion `displayForecast(data)` übergeben, um sie auf dem Display anzuzeigen.

Detaillierte Beschreibung der Funktion `displayForecast(data)`

Die Funktion `displayForecast(data)` führt folgende Schritte aus:

1. **Vorhersage-Div vorbereiten:** Das HTML-Element mit der ID `forecast` wird selektiert und dessen Inhalt wird geleert.
2. **Daten nach Tagen gruppieren:** Die Wetterdaten werden nach Tagen gruppiert und in einem Objekt gespeichert.
3. **Vorhersage für drei Tage anzeigen:** Es werden nur die Vorhersagen für den aktuellen Tag und die nächsten zwei Tage angezeigt.
4. **Tagesvorhersagen generieren:** Für jeden Tag wird ein Div-Element erstellt, das die Tagesvorhersage enthält. Jede Tagesvorhersage zeigt die Uhrzeit, eine Beschreibung des Wetters und die Temperatur an.
5. **Vorhersagen in das HTML einfügen:** Die generierten Tagesvorhersagen werden in das HTML-Element `forecast` eingefügt.

Das Wettervorhersage-Widget bietet eine einfache und effektive Lösung zur Anzeige der Wettervorhersage für die nächsten drei Tage. Es lässt sich leicht im Display integrieren und an individuelle Bedürfnisse anpassen. Die Nutzung der OpenWeatherMap API ermöglicht eine zuverlässige und aktuelle Wettervorhersage.

3.2.5 Notizen Widget

Erarbeitet von: Leon Kranner

Das Notizen-Widget ist eine JavaScript-Anwendung, die es dem Benutzer ermöglicht, Notizen zu erstellen und zu speichern. Diese Notizen werden im lokalen Speicher des Browsers gespeichert, sodass sie auch nach dem Schließen des Browsers erhalten bleiben.

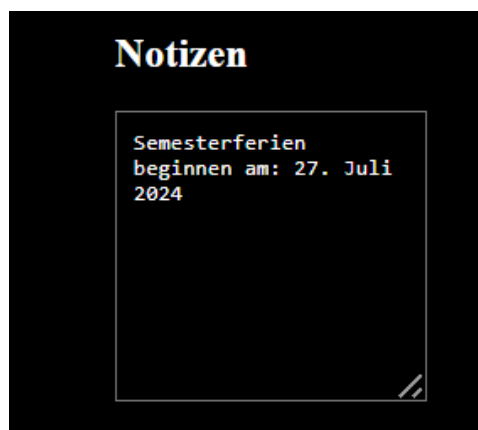


Abbildung 3.5: Notizen Widget
Quelle: eigene Darstellung

Die Hauptfunktionen des Widgets sind `loadNotes()` und `saveNotes()`. `loadNotes()` lädt die gespeicherten Notizen aus dem lokalen Speicher, während `saveNotes()` die Notizen im lokalen Speicher speichert.

Detaillierte Beschreibung der Funktion `loadNotes`

Die Funktion `loadNotes()` führt folgende Schritte aus:

1. **Gespeicherte Notizen abrufen:** Die Notizen werden aus dem lokalen Speicher des Browsers abgerufen.
2. **Notizen im Textbereich anzeigen:** Wenn gespeicherte Notizen vorhanden sind, werden sie im Textbereich (`notesTextarea`) angezeigt.

Detaillierte Beschreibung der Funktion `saveNotes`

Die Funktion `saveNotes()` führt folgende Schritte aus:

1. **Notizen aus dem Textbereich abrufen:** Der Inhalt des Textbereichs (`notesTextarea`) wird abgerufen.
2. **Notizen im lokalen Speicher speichern:** Die Notizen werden im lokalen Speicher des Browsers gespeichert.

Eventlistener

Die Anwendung nutzt zwei Eventlistener:

- **DOMContentLoaded-Event:** Lädt die Notizen, sobald die Seite vollständig geladen ist.
- **input-Event:** Speichert die Notizen, sobald der Benutzer den Text ändert.

Das Notizen-Widget bietet eine einfache und effektive Lösung zur Erstellung und Speicherung von Notizen im lokalen Speicher des Browsers. Es lässt sich leicht in bestehende Webseiten integrieren und an individuelle Bedürfnisse anpassen. Die Nutzung des lokalen Speichers ermöglicht eine persistente Speicherung der Notizen, auch nach dem Schließen des Browsers.

3.2.6 Uhrzeit Widget

Erarbeitet von: Marcel Wagner

Die Implementierung des Uhrzeit Widgets für den Smart Mirror ist ein wichtiger Schritt zur Verbesserung der Funktionalität und Benutzerfreundlichkeit des Geräts. Ziel dieses Widgets ist es, die aktuelle Uhrzeit exakt und zuverlässig anzuzeigen. Wobei die Anzeige in Echtzeit aktualisiert werden muss, um stets die genaue Uhrzeit widerzuspiegeln. In der Nachfolgenden Abbildung kann das Implementierte Uhrzeit Widget entommen werden.

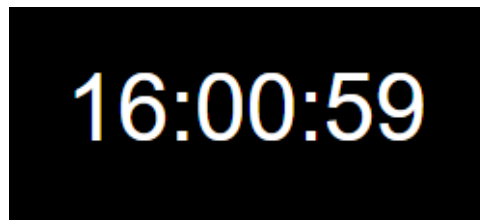


Abbildung 3.6: Uhrzeit Widget
Quelle: eigene Darstellung

Die Implementierung des vorher gezeigten Widgets basierte auf der Nutzung von JavaScript zur Echtzeitaktualisierung der Uhrzeit und HTML zur Einbettung des Widgets in die Benutzeroberfläche des Smart Mirrors. Desweiteren wurde CSS benutzt um das Widget zu formatieren. Die JavaScript Funktion sorgt dafür, dass die Uhrzeit jede Sekunde aktualisiert wird, während das HTML Dokument die Struktur definiert. Abschließend definiert die CSS Datei das Styling des Widgets.

Während der Entwicklung des Widgets traten mehrere Herausforderungen auf. Eine der größten Herausforderungen bestand darin, sicherzustellen, dass die Uhrzeit in Echtzeit und ohne Verzögerung aktualisiert wird. Dies war besonders wichtig, um die Genauigkeit der angezeigten Zeit zu gewährleisten. Die Verwendung der 'setTimeout' Funktion in JavaScript ermöglicht eine wiederholte Ausführung der Aktualisierungsfunktion in einem festgelegten Intervall von einer Sekunde, wodurch eine kontinuierliche und genaue Aktualisierung der Uhrzeit sichergestellt wurde. Eine weitere Herausforderung war die exakte Zeitanzeige, insbesondere hierbei ist wichtig die Erwähnung der Formatierung der Uhrzeit, um sicherzustellen, dass Stunden, Minuten und Sekunden stets zweistellig angezeigt werden. Durch die Verwendung der 'padStart' Methode konnten die Zahlen auf eine konstante Länge von zwei Stellen gebracht werden, indem bei Bedarf führende Nullen hinzugefügt werden. Dies gewährleistete eine konsistente und gut lesbare Anzeige.

Das Implementierte Widget wurde im Anschluss auf ihre Funktionalitäten getestet. Auf diesen Bereich wird im Kapitel 3.3 genauer für alle Widget eingegangen.

3.2.7 Verkehrsinformation

Erarbeitet von: Marcel Wagner

Die Implementierung des Stau Widgets auf dem Smart Mirror stellt einen wichtigen Schritt dar, um den Nutzern eine umfassende und zuverlässige Quelle für aktuelle Verkehrsinformationen zur Verfügung zu stellen. Das Widget wurde speziell entwickelt, um eine Echtzeitübersicht über die Verkehrslage in Regensburg zu bieten, was insbesondere für Pendler von großem Nutzen ist. Durch die Verwendung von JavaScript wurde eine nahtlose Integration mit der OpenStreetMap Overpass API realisiert, die als zuverlässige Datenquelle für Verkehrsdaten dient. Bevor die Vorgehensweise der Implementierung betrachtet wird kann der Nachfolgenden Abbildung nun das Implementierte Widget für dem Fall das aktuelle Kein Stau in Regensburg vorhanden ist entnommen werden.



Abbildung 3.7: Verkehrsinformations Widget
Quelle: eigene Darstellung

Die Strategie hinter der Implementierung war zweigleisig: Zum einen wurde eine sofortige Aktualisierung der Verkehrsinformationen beim Laden der Seite implementiert, um den Nutzern bei jedem Besuch des Smart Mirrors die aktuellsten Daten bereitzustellen. Zum anderen erfolgt eine regelmäßige automatische Aktualisierung alle fünf Minuten, um sicherzustellen, dass die angezeigten Informationen kontinuierlich aktuell gehalten werden. Dieser Ansatz gewährleistet eine hohe Aktualität und Relevanz der bereitgestellten Verkehrsinformationen.

Während der Entwicklung wurden mehrere Herausforderungen gemeistert, darunter die robuste Fehlerbehandlung, um sicherzustellen, dass Netzwerkprobleme oder API Ausfälle die Funktionalität des Widgets nicht beeinträchtigen. Ein besonderes Augenmerk lag auf der Gewährleistung einer stabilen und zuverlässigen Datenaktualisierung, die für eine nahtlose Benutzererfahrung entscheidend ist.

Das Verkehrs Widget präsentiert die Verkehrslage in einer klaren und intuitiven Benutzeroberfläche. Es informiert die Nutzer klar verständlich darüber, ob derzeit ein Stau vorliegt oder nicht, und bietet gegebenenfalls zusätzliche Informationen über Verkehrshindernisse oder Verkehrswarnungen. Diese klare visuelle Darstellung hilft den Nutzern, schnell zu erfassen, wie die aktuelle Verkehrssituation ihre geplante Route beeinflussen ist.

Das Implementierte Widget wurde im Anschluss auf ihre Funktionalitäten getestet. Auf diesen Bereich wird im Kapitel 3.3 genauer für alle Widget eingegangen.

Potenzielle Erweiterungen

Ein bedeutendes Potenzial für eine Weiterentwicklung des Verkehrs Widgets liegt in der Integration mit der bereits bestehenden Smartphone Schnittstellen, um eine dynamische Anpassung des Standorts zu ermöglichen. Diese Erweiterung würde es den Nutzern gestatten, die Verkehrsinformationen auf ihrem Smart Mirror basierend auf ihrem aktuellen Standort abzurufen. Dies ist besonders relevant für Pendler, die täglich unterschiedliche Routen verwenden oder deren Start- und Zielorte variieren.

3.2.8 Schlagzeilen

Erarbeitet von: Marcel Wagner

Die Implementierung des Nachrichten Widgets für den Smart Mirror stellt einen wichtigen Schritt dar, um den Nutzern eine aktuelle und relevante Informationsquelle direkt auf seinem Smart Mirror zur Verfügung zu stellen. Das Widget wurde in JavaScript entwickelt und verwendet die 'RSS2JSON-API', um die neuesten Nachrichtenartikel eines ausgewählten RSS Feeds abzurufen und auf dem Smart Mirror anzuzeigen. Dies ermöglicht eine dynamische und automatische Aktualisierung der Nachrichteninhalte, sobald der Nutzer den Spiegel nutzt. In der nachfolgenden Abbildung kann das Funktionierende Widget entnommen werden.



Abbildung 3.8: News Widget
Quelle: eigene Darstellung

Ein zentrales Element der Implementierung ist die Verwendung des 'DOMContentLoaded' Events, das sicherstellt, dass das Widget erst aktiv wird, nachdem die gesamte Seite vollständig geladen ist. Dies ist notwendig damit alle notwendigen Ressourcen und Elemente bereitstehen, bevor die Datenabfrage und die Darstellung der Nachrichten beginnen.

Die Funktionalität des Widgets umfasst die Asynchronität der Datenabfrage über die Fetch API, die die RSS Feeds von Nachrichtenquellen in ein JSON Format umwandelt, das vom JavaScript Code weiterverarbeitet werden kann. Dies ermöglicht eine schnelle und effiziente Bereitstellung der neuesten Nachrichteninhalte direkt auf dem Smart Mirror, ohne dass der Nutzer zusätzliche Schritte unternehmen muss, um sich auf dem Laufenden zu halten.

Eine besondere Herausforderung während der Implementierung war die unterschiedliche Verfügbarkeit von RSS Feeds bei verschiedenen Nachrichtenseiten. Viele führende Nachrichtenagenturen und Zeitungen bieten zwar RSS Feeds an, einige jedoch

nicht oder beschränken den Zugang zu ihren Inhalten über diese Schnittstelle. Dies erforderte eine sorgfältige Auswahl geeigneter RSS Feeds, die eine kontinuierliche und zuverlässige Datenversorgung gewährleisten konnten. Die Ausgegeben Nachrichten dieses Widget sind aus der Frankfurter Allgemeinen Zeitung.

Um die Benutzerfreundlichkeit zu maximieren, wurde die Benutzeroberfläche des Widgets bewusst einfach und intuitiv gestaltet. Die angezeigten Nachrichten werden in einer geordneten Liste präsentiert. Das Implementierte Widget wurde im Anschluss auf ihre Funktionalitäten getestet. Auf diesen Bereich wird im Kapitel 3.3 genauer für alle Widget eingegangen.

3.2.9 Tankstellen

Erarbeitet von: Marcel Wagner

Die Implementierung des Tankstellen Widgets für den Smart Mirror verfolgt das Ziel, den Nutzern eine praktische und zeitnahe Information über den günstigsten Kraftstoffpreise einer Tankstelle in der Nähe zu bieten. Diese Funktionalität wurde durch die Integration von JavaScript und die Nutzung der Tankerkoenig API realisiert, die speziell auf die Abfrage von Tankstellenpreisen und Tankstelleninformationen ausgerichtet ist. In der Nachfolgenden Abbildung kann das Implementierte Widget entnommen werden.



Abbildung 3.9: Tankstellen Widget
Quelle: eigene Darstellung

Zu Beginn des Implementierungsprozesses wird der 'DOMContentLoaded' Eventlistener verwendet, um sicherzustellen, dass sämtliche Inhalte der Webseite geladen sind, bevor die Datenabfrage gestartet wird. Dies gewährleistet eine stabile und zuverlässige Performance des Widgets auf dem Smart Mirror. Die API Anfrage erfolgt unter Verwendung eines spezifischen API Schlüssels, der die Authentifizierung gegenüber der Tankerkoenig API ermöglicht. Der Standortbezug erfolgt für die Stadt Regensburg mit definierten geografischen Koordinaten.

Die Datenabfrage wird asynchron durchgeführt, um eine reibungslose Interaktion mit der API zu gewährleisten. Nachdem die Daten abgerufen wurden, erfolgt eine Überprüfung auf erfolgreiche Antwort und die Verfügbarkeit von Tankstelleninformationen. Falls die API Daten erfolgreich zurückgegeben werden und Tankstelleninformationen vorhanden sind, wird die günstigste Tankstelle ermittelt. Dies geschieht durch einen Vergleich der Kraftstoffpreise der abgerufenen Tankstellen, wobei die preisgünstigste Option ausgewählt und deren Informationen weiterverarbeitet werden.

Die Darstellung der Tankstelleninformationen auf dem Smart Mirror erfolgt in einer klar strukturierten Form. Dies umfasst den Namen der Tankstelle, die vollständige Adresse inklusive Straße, Hausnummer, Postleitzahl und Ort sowie den aktuellen Preis pro Liter Kraftstoff. Diese Informationen sind leicht zugänglich und ermöglichen es dem Nutzer, schnell die wichtigsten Details zu erfassen und eine informierte Entscheidung zu treffen.

Das Implementiert Widget wurde im Anschluss auf ihre Funktionalitäten getestet. Auf diesen Bereich wird im Kapitel 3.3 genauer für alle Widget eingegangen.

Potenzielle Erweiterung

Ein bedeutendes Potenzial für die Weiterentwicklung des Tankstellen Widgets liegt in der dynamischen Anpassung der Widget Inhalte basierend auf Eingaben vom Smartphone der Nutzer. Diese Erweiterung würde eine noch individuellere und nutzerzentrierte Erfahrung ermöglichen, indem sowohl der bevorzugte Kraftstofftyp als auch der Standort der Nutzer flexibel konfiguriert werden können.

3.3 Test Verfahren

Erarbeitet von: Leon Kranner und Marcel Wagner

Für die implementierten Widgets auf dem Smart Mirror wurden umfangreiche Testverfahren angewendet, die sowohl die Funktionalität als auch die Benutzererfahrung der einzelnen Widgets sicherstellen sollen. Diese unterschiedlichen Testverfahren werden nun im Folgenden genauer beschrieben.

Funktionalitätstests: Dieser Test konzentrierte sich auf die grundlegenden Aufgaben jedes Widgets. Das Uhrzeitwidget wurde auf seine Fähigkeit getestet, die aktuelle Uhrzeit präzise anzuzeigen. Außerdem wurde sichergestellt, dass die Darstellung formatiert und korrekt aktualisiert wird. Beim News Widget lag der Fokus auf der korrekten Abrufung und Darstellung aktueller Nachrichten, wobei sichergestellt wurde, dass die Informationen stets aktuell und relevant sind. Das Tankstellenwidget durchlief API Integrationstests, um sicherzustellen, dass die Kraftstoffpreise korrekt von der Tankerkoenig API abgerufen und in einem klaren Format angezeigt werden. Das Verkehrsinformations Widget wurde auf seine Fähigkeit geprüft, Verkehrsinformationen zeitnah abzurufen und zuverlässig darzustellen, um Nutzer vor aktuellen Verkehrsbehinderungen zu warnen. Beim Termine Widget wurde getestet, ob die neusten 3 Termine angezeigt werden und keine alten Termine zu sehen sind. Außerdem wurde getestet, ob das Widget aktualisiert, wenn ein Termin vorbei ist und einen neuen Termin anzeigt. Beim Kalender Widget wurde gecheckt, ob immer der richtige Tag markiert wird und ob die Tage immer unter den richtigen Wochentagen angezeigt werden. Bei der Wettervorhersage wurde wie beim News Widget sichergestellt, dass die aktuellen Daten vom Wetter korrekt sind und jederzeit abgerufen werden können. Außerdem wurde beim Notizen Widget getestet, ob auch die Notizen im Raster korrekt angezeigt werden.

Benutzererfahrungstests: Diese waren entscheidend, um sicherzustellen, dass die Widgets intuitiv sind. Hierbei halfen Usability Tests, diese bewerteten die Widgets auf Benutzerfreundlichkeit der Benutzeroberfläche. Dabei wurde besonders darauf geachtet, dass die Widgets übersichtlich gestaltet sind und Nutzer schnell die benötigten Informationen finden können.

Performance- und Lasttests: Diese Testverfahren waren ebenfalls Teil der Teststrategie, um sicherzustellen, dass die Widgets unter verschiedenen Bedingungen effizient arbeiten. Ladezeittests wurden durchgeführt, um sicherzustellen, dass die Widgets schnell genug reagieren und Daten effizient verarbeiten. Skalierbarkeitstests wurden genutzt, um sicherzustellen, dass die Widgets auch bei erhöhtem Datenverkehr stabil bleiben und keine übermäßigen Ressourcen verbrauchen, was besonders wichtig für die Langzeitnutzung ist.

Integrationstest: Dabei wurden Kompatibilitätstests durchgeführt, um sicherzustellen, dass die Widgets reibungslos mit anderen Komponenten des Smart Mirrors interagieren. Systemtests prüften die Gesamtfunktionalität des Smart Mirrors unter verschiedenen Betriebsbedingungen, um sicherzustellen, dass alle Widgets harmonisch zusammenarbeiten und die Gesamtleistung des Systems nicht beeinträchtigen werden.

Diese umfassenden Testverfahren stellen sicher, dass die implementierten Widgets nicht nur funktional sind, sondern auch eine qualitativ hochwertige Benutzererfahrung bieten und unter allen Bedingungen zuverlässig arbeiten.

4 | Spiegel AI Remote

Erarbeitet von David Vollmer.

Im folgenden wird die **Spiegel AI Remote** App - auch **Remote App** genannt - beschrieben. Es handelt sich dabei um eine mobile Anwendung, dessen Hauptaufgabe die Fernsteuerung des Smart Mirrors ist.

4.1 Die Flutter™ SDK

Für die Entwicklung einer mobilen Applikation gibt es heutzutage viele Tool-Kits, die verwendet werden können. Laut einer von JetBrains durchgeführten Umfrage war Flutter im Jahr 2023 das am häufigsten verwendete mobile plattformübergreifende Framework.

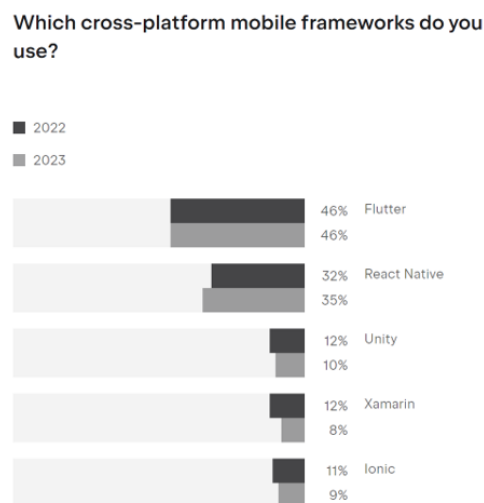


Abbildung 4.1: JetBrains Entwicklerumfrage 2023 **jetbrains_survey**

Zusätzlich zu ihrer Popularität ist die von Google entwickelte Flutter SDK in der Lage, mittels AOT-Compiler Programme direkt für die Zielplattform zu kompilieren. Dabei wird die Programmiersprache Dart verwendet. **dart platform** Flutter unterstützt, unter anderem, die Entwicklung auf den Plattformen Android SDK, iOS, Windows, macOS und Web. **flutter_supported_platforms** Für die Umsetzung der Fernsteuerungs-App wurde insbesondere mit den Plattformen Android und iOS entwickelt und getestet.

4.2 Funktionen

Um das Display des Spiegel Als fernzusteuern, müssen einige Hauptfunktionalitäten vorhanden sein. Die Remote App muss in der Lage sein, mit dem Spiegel zu kommunizieren, die Anzeige der Widgets auf dem Display zu ändern, verfügbare Widgets auszuwählen und Profile zu verwalten. Mit Ausnahme der ersten Anforderung, welche in der **Implementierung** und im Kapitel **Schnittstelle** näher beschrieben wird, werden all diese Punkte in sogenannten Ansichten (englisch: views) behandelt. Diese kann der Nutzer mithilfe einer Navigationsleiste auswählen.

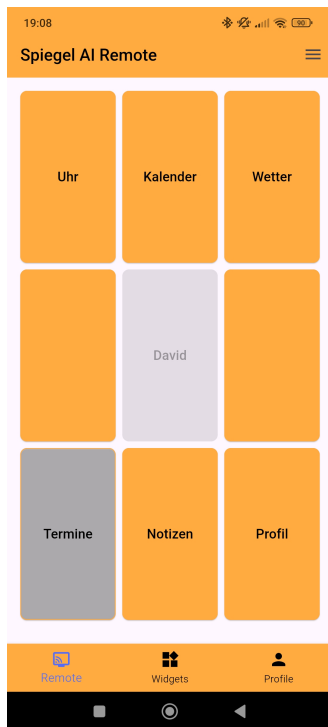


Abbildung 4.2: Remote Ansicht

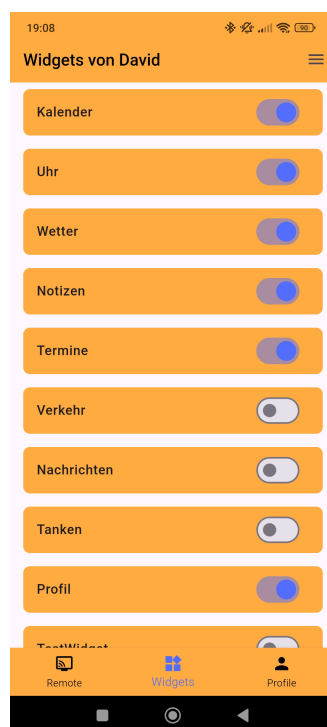


Abbildung 4.3: Widgets Ansicht

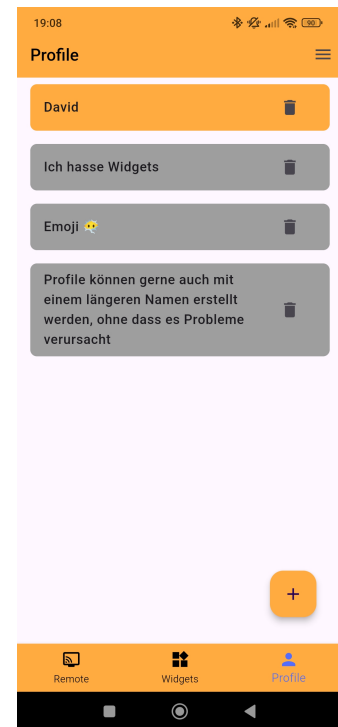


Abbildung 4.4: Profile Ansicht

4.2.1 Remote View

Die Remote View, welche die Standardansicht nach Öffnen der App ist, bietet die Möglichkeit, den Status der Displayanzeige am Spiegel zu ändern. In der Mitte wird der Name des gerade ausgewählten Profils angezeigt. Dieses Feld lässt keinerlei Interaktion zu, da das zentrale Feld der Spiegelanzeige frei bleibt. Das bedeutet, dass bis zu acht Widgets angezeigt und geändert werden können. Mit einem Klick auf einen Button wird das jeweilige Widget aus- oder eingeblendet. Ein Feld in grauer Farbe bedeutet, dass das Widget vom Spiegel AI Display nicht angezeigt wird. Zieht man ein Widget über ein anderes, werden ihre Positionen getauscht. Die Felder, die keinen Text enthalten, haben die selben Interaktionsmöglichkeiten wie die anderen. Sie sind Platzhalter für Widgets, die hinzugefügt werden können. Falls kein Profil ausgewählt ist, wird im Remote View eine Standardeinstellung angezeigt und jegliche Interaktion der Buttons ist ausgestellt. Bei einem Versuch, ohne Profilselektion eine Änderung

vorzunehmen, wird eine Snackbar angezeigt, welche darauf verweist, dass ein Profil geladen sein muss.

4.2.2 Widgets View

In der Widgets Ansicht können für das ausgewählte Profil Widgets ausgewählt werden. Diese View bietet die Widgets Kalender, Uhr, Wetter, Notizen, Termine, Verkehr, Nachrichten, Tanken, Profil und TestWidget an. Bei letzterem handelt es sich um einen Platzhalter, welcher zum Testen der Widgetfunktionalitäten verwendet wurde, aber auch zukünftig mit einem neuen Widget ersetzt werden kann. Die Anwendungen der restlichen Widgets sind im Kapitel **Display** beschrieben. Mithilfe eines Toggle-Buttons werden bis zu acht Widgets selektiert. Beim Versuch, ein neuntes Widget auszuwählen, schlägt dies fehl und eine Snackbar benachrichtigt über die Obergrenze erlaubter Widgets. Auf die Änderung eines Widgets, ohne ein Profil geladen zu haben, folgt ebenfalls eine dementsprechende Fehlermeldung. Wird ansonsten ein Widget ausgeschaltet, dann wird das im Toggle-Button signalisiert und in der Remote View wird der Name des Widgets mit einem leeren Feld ersetzt. Wenn ein ausgeschaltetes Widget ausgewählt wird, aktualisiert sich auch da der Toggle-Schalter und in der Remote Ansicht wird das erste Feld ohne Textinhalt mit dem Namen des Widgets versehen.

4.2.3 Profile View

Die letzte navigierbare Ansicht ist die Profile View. Hier findet die Verwaltung der gespeicherten Profile statt. Die Profile werden aufgelistet und können mit einem Klick ausgewählt werden. Hält man ein Profil für kurze Zeit gedrückt, kann man diese in ihrer Position in der Auflistung ändern, indem man sie an die gewünschte Stelle zieht. Löschen kann man einen Eintrag, indem auf das Mülleimer-Icon geklickt wird. Darauf öffnet sich ein sogenanntes Alert-Dialog, welches das Abbrechen oder Bestätigen der Löschung durchführt. Ein neues Profil kann erstellt werden, indem auf den Button, welcher sich in der Ansicht rechts unten befindet und mit einem '+'-Symbol gekennzeichnet ist, gedrückt wird. Es erscheint ebenfalls ein Alert-Dialog, welches mithilfe eines Texteingabefeldes einen Profilnamen geben kann. Dieser Prozess kann auch abgebrochen oder bestätigt werden. Falls bei Bestätigung der Name des Profils leer oder schon vergeben ist, wird unterhalb des Textfeldes eine entsprechende Fehlermeldung ausgegeben. Wenn das Erstellen des Profils erfolgreich ist, wird das neue Profil direkt ausgewählt und bekommt die ersten acht Widgets in der Widgets View zugeordnet. Sie werden dementsprechend in der Remote Ansicht angezeigt. Alle Anpassungen, die in diesen beiden Ansichten getätigt werden, werden in den jeweilig ausgewählten Profilen gespeichert.

4.3 Implementierung

Der Dart-Code, welcher die Codebase für die Kompilierung des Programms darstellt, befindet sich in einem Flutter-Projekt im Verzeichnis mit dem Namen `lib`. Der Websocket wird in der `websocket_manager.dart` verwaltet. Die Funktionalitäten der drei Views sind in den Dateien `remote_content.dart`, `widgets_content.dart` und `profile_content.dart` implementiert.

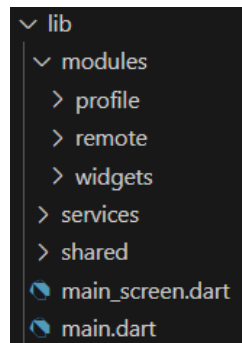


Abbildung 4.5: Verzeichnisstruktur der `lib` im Spiegel AI Remote Projekt

4.3.1 Websocket

Um die Kommunikation zwischen der Remote App und Spiegel AI sicherzustellen, muss der Websocket in der App richtig verwaltet werden. Der sogenannte `WebSocketManager` bewältigt dies mithilfe von Bibliotheken, die die Flutter-Umgebung zur Verfügung stellt. Er ermöglicht, dass eine Verbindung zum Websocket-Server hergestellt und abgebrochen werden kann und erlaubt das Empfangen und Senden von Daten. Die App empfängt Daten als Strings vom Webserver, jedoch werden nur jene mit bestimmten Eigenschaften auch verarbeitet. Der zu empfangende String muss in ein JSON-Format dekodiert werden können. Dann wird geprüft, ob der Wert des ersten Schlüssels mit der Bezeichnung `sender` den Wert `mirror` hat. Dies prüft, ob die Nachricht des Servers ursprünglich vom Spiegel AI gesendet wurde. In diesem Fall werden die Werte des Keys mit der Bezeichnung `profiles` lokal in der App gespeichert. In einem ähnlichen Stil werden Nachrichten versendet. Der einzige Unterschied ist hierbei, dass dabei der `sender` den Wert `remote` bekommt, um zu signalisieren, dass die Quelle der Nachricht die mobile Applikation ist. Gesendet werden diese immer, nachdem eine Änderung der Profile stattfindet. Es gibt jedoch eine weitere Nachricht, die die Remote App versendet. Jedes mal, nachdem eine Verbindung mit dem Websocket aufgebaut wurde, sendet sie einen String mit dem Inhalt `fetch`. Auf die Nachricht folgt, dass der Spiegel AI seinen aktuellen Stand der Profile sendet. Dies ist wichtig, damit die lokalen Profildaten der App synchronisiert werden, bevor sie in der Lage ist, Änderungen vorzunehmen. Ansonsten kann es dazu führen, dass Profildaten des Smart Mirrors mit veralteten Daten der App überschrieben werden. Um sich mit dem Websocket-Server zu verbinden, muss die IP-Adresse und der Port des Servers im Format `ws://<server-ip>:<server:port>` angegeben werden. Der Websocket schließt, sobald die App entweder geschlossen oder in den Hintergrund laufen gelassen wird. Sobald die App wieder geöffnet wird, wird auch die Verbindung zum Websocket hergestellt und sendet den `fetch`-String an den Server.

Ein Problem, das bei der Implementierung des Websockets besteht, ist dass die IP-Adresse des Servers im Code festgelegt ist. Das sorgt dafür, dass bei Änderung der Server-IP der Sourcecode der App umgeändert und neu kompiliert werden muss. In der Entwicklungsphase war dies noch unproblematisch, da die IP-Adresse mithilfe des Debug-Modus des Flutter Tool-Kits in kurzer Zeit zu ändern war. Beim tatsächlichen Betrieb kann der Nutzer die Remote App dadurch jedoch nicht verwenden. Ein Lösungsansatz hierfür ist, dass die App die Eingabe einer IP-Adresse zulässt. Dies kann beispielsweise im sogenannten Drawer, einer Menüansicht, implementiert werden. Hierbei kann eine "Verbinden"-Option gewählt werden, welche den Nutzer dazu

auffordert, eine gültige IP-Adresse einzutippen. Daraufhin versucht die Spiegel AI Remote eine Verbindung zum Websocket herzustellen. Schlägt dies fehl, wird der Nutzer erneut zur Eingabe aufgefordert. Gelingt die Verbindung zum Server, wird eine entsprechende Nachricht angezeigt. Generell ist das Implementieren von Fehlerhandling bei Verbindungsabbrüchen sinnvoll, damit der Nutzer der App sich einen besseren Überblick der Kommunikation verschaffen kann. Um herauszufinden, mit welcher IP-Adresse man sich verbinden muss, kann diese auf dem Spiegel Display angezeigt werden.

4.3.2 Remote

Die Remote App ist zuständig für die Änderung einer Profileigenschaft mit dem Namen `state`. Jedes Profil hat einen State, welcher die Positionierung (oder `index`), ID und Anzeigestatus aller ausgewählten Widgets angibt. Mithilfe dieses Status wird die Anzeige des Spiegels festgelegt. Die Änderungen des States in der Remote Ansicht sind nur am ausgewählten Profil möglich. Ist der Wert der Angegebenen ID -1, dann handelt es sich hierbei um ein Feld ohne Widget. Das heißt, es wird an der Stelle kein Name angezeigt und am Spiegel erscheint an dieser Position kein Widget.

```
{
  "index": 3,
  "id": 8,
  "enabled": true
},
```

Abbildung 4.6: Beispiel eines State-Eintrags im JSON-Format

Die Änderung des Anzeigestatus verläuft so, dass nach dem Klick auf einen Button der Wert des Keys `enable` negiert wird. Ist der Wert `true`, so wird das visualisiert, indem die Farbe des Feldes orange ist. Ist er `false`, dass erscheint es grau. Wird ein Widget über ein anderes gezogen, so werden die Werte ihrer `index`-Schlüssel getauscht. Dies hat zur Folge, dass sowohl in der Remote Ansicht, als auch im Spiegeldisplay die Positionen geändert werden.

Ein Problem der Implementierung dieser Ansicht ist das mittlere Feld. Dieses war ursprünglich ein Widget, das Interaktionen zuließ. Diese Interaktionen sind nun wie gewollt nicht mehr möglich, jedoch handelt es sich hier immer noch um ein Widget, welches auch über den State versendet wird. Dies führt jedoch dazu, dass es beim Handling der Widgets viele Ausnahmesituationen gibt, in denen das mittlere Feld berücksichtigt werden muss. Ein Beispiel dafür ist das Setzen des States, wenn ein neues Profil default-Werte zugewiesen bekommt. Die IDs der Widgets, die hier ausgewählt sind, sind eine Liste der Zahlen von 0 bis 7. Um den State aber richtig zu befüllen, werden die IDs mit den Werten 0, 1, 2, 3, -1, 4, 5, 6, 7 initialisiert, da das Widget an der Stelle 4 „nicht ausgewählt“ ist. Um dieses Problem zu beheben, muss das mittlere Feld aus der State genommen und das Handling der Ausnahmefälle angepasst werden.

4.3.3 Widgets

Die Widgets View verwaltet die sogenannten `selectedWidgets`. In der Liste der zehn Widgets können diese per Toggle ab- oder ausgewählt werden. Die Funktionalität der Änderung des States nachdem ein Widget selektiert oder deselektiert wurde, ist in der `_updateProfileState(int index)` implementiert. Diese prüft zuerst, ob ein Widget ausgeschaltet wurde. In diesem Fall wird über das den State des ausgewählten Profils iteriert, bis das Widget mit der ID des `index` gefunden wurde. Die ID wird mit dem Wert `-1` überschrieben, was zur Folge hat, dass an dieser Stelle des States kein Widget mehr gewählt ist. Soll wiederum ein Widget hinzugefügt werden, wird ebenfalls durch den State iteriert, bis die ID `-1` gefunden ist. Eine Ausnahme ist hier das Widget mit an vierter Stelle, welches ignoriert werden soll. Die ID des Widgets wird daraufhin mit dem `index` des Widgets ersetzt, sodass der erste Eintrag ohne Widget nun vom selektierten Widget dargestellt wird. Hier ist auch zu beachten, dass dieser `index` nicht mit dem des States zu verwechseln ist. Es handelt sich nämlich um den `index` der Liste der Widgets, welche im State als ID dargestellt ist.

Ein Problem, das sich hier zeigt ist die limitierte Auswahl der Widgets. Es ist tatsächlich sehr einfach, weitere Widgets hinzuzufügen, indem man neue Einträge in der `widgetNames`-Liste anfügt. Aber die Funktionalität der Widgets muss dann in der Displayapplikation vorhanden sein.

4.3.4 Profile

Die Profile Ansicht erlaubt das Selektieren, Löschen und Hinzufügen von Profilen. Die Selektion wird durch einen Button-Klick-Listener realisiert. Das angeklickte Profil wird ausgewählt. Falls es sich jedoch um das geladene Profil handelt, wird dieses abgewählt. In diesem Fall gibt es kein sogenanntes `selectedProfile` und somit keinen State, der angezeigt werden soll. Deswegen wird hier die Anzeige im Remote und im Spiegel AI auf die gleichen default-Werte gesetzt. Eine weitere Option ist das Löschen von Profilen. Nachdem die Löschung des Profils vom Nutzer bestätigt ist, wird das Profil aus der Liste der Profile entfernt und in der Profilansicht nicht mehr angezeigt. War das gelöschte Profil das `selectedProfile`, dann hat das zur Folge, dass kein Profil ausgewählt sein wird. Wird das Anlegen eines neuen Profils vom Nutzer bestätigt, so wird zunächst geprüft, ob der angegebene Name leer oder ein Duplikat ist. In den Fällen wird unterhalb des Texteingabefeldes eine jeweilige Fehlermeldung ausgegeben. Ansonsten wird das neue Profil mit dem gegebenen Namen und einer ID, welche den Millisekunden seit dem Unix-Epoch entspricht, in der Liste angelegt und selektiert.

In dieser View gab es öfter das Problem, dass bei Änderungen die Fehlermeldungen für den Eintrag des Namens nicht angezeigt wurden. Die genaue Ursache konnte ich nicht feststellen, jedoch schien es nicht aufzutreten, wenn der Dialog zum Hinzufügen neuer Profile unverändert blieb.

4.3.5 Sonstige Implementierungen

Eine wichtige Funktion ist das korrekte Speichern der Profile. Hierbei ist es nicht nur wichtig, dass die Profile lokal gespeichert werden, sondern dass diese auch nach jeder Speicheranweisung an den Spiegel AI gesendet werden. Falls dies nicht geschieht, können alte Profildaten des Spiegels die Änderungen in der Remote App überschreiben. Deswegen ist es ebenfalls wichtig, dass nach jeder Profiländerung die Daten

gespeichert werden. Eine weitere Funktion ist, dass die Ansichten beim Laden von Profilen aktualisiert werden. Das heißt, wenn Profile vom Websocket gesendet werden, sollen diese Änderungen in der Remote App widergespiegelt werden, ohne dass man dafür in eine andere Sicht navigieren muss. Hierbei gab es jedoch eine Schwierigkeit. Es funktionierte problemlos in der Remote und Widgets Anzeige, aber nicht im Profile View. Hier war das Problem, dass das Textfeld, welches zum Namenseintrag der Profile verwendet wurde, nicht korrekt geschlossen werden konnte, wenn eine Aktualisierung der Ansicht stattfand. Dieser Fehler war besonders frustrierend, da ich den Textfeld-Controller tatsächlich mit dem `dispose()`-Befehl schließe. Trotz Debugging und Recherche kam ich zu keiner Lösung dieses Problems und entschied mich, die Profilanzeige nicht zu aktualisieren, wenn der Spiegel Profile sendet.

5 | Gesichtserkennung

Erarbeitet von Marco Kuner.

Diese Dokumentation beschreibt die Entwicklung einer Gesichtserkennungslösung für einen Smart Mirror. Das Projekt wurde im Rahmen eines Vier-Personen-Teams durchgeführt und beinhaltet die Recherche, Implementierung und Optimierung verschiedener Gesichtserkennungstechnologien. Besonderes Augenmerk liegt auf der genauen Protokollierung der Entscheidungsprozesse und der technischen Herausforderungen.

5.1 Einleitung

5.1.1 Projektziel

Das Hauptziel dieses Projekts war die Entwicklung einer fortschrittlichen Gesichtserkennungslösung für einen Smart Mirror. Dieser Smart Mirror sollte in der Lage sein, Benutzer anhand ihrer Gesichter zu erkennen und personalisierte Informationen anzuzeigen. Die Gesichtserkennung sollte zuverlässig unter verschiedenen Bedingungen wie wechselnden Lichtverhältnissen und unterschiedlichen Gesichtswinkeln funktionieren.

5.1.2 Bedeutung der Gesichtserkennung

Gesichtserkennungstechnologien haben in den letzten Jahren erheblich an Bedeutung gewonnen. Sie finden Anwendungen in zahlreichen Bereichen wie Sicherheit, wo sie zur Zugangskontrolle und Überwachung eingesetzt werden, in der Personalisierung, wo sie individuelle Benutzererlebnisse ermöglichen, und in der Benutzerfreundlichkeit, da sie eine nahtlose Interaktion mit technischen Geräten bieten. Die Entwicklung einer zuverlässigen Gesichtserkennung für den Smart Mirror ist also zwingend erforderlich für eine reibungslose Kundenerfahrung.

5.2 Recherche und Anfangsphase

5.2.1 Grundlagen der Gesichtserkennung

Gesichtserkennung ist ein Bereich der Computer Vision, der sich mit der Identifikation von Individuen anhand ihrer Gesichtszüge beschäftigt. Dies geschieht durch den Einsatz von Algorithmen, die charakteristische Merkmale eines Gesichts extrahieren und analysieren. Traditionelle Methoden der Gesichtserkennung umfassen Ansätze

wie die Verwendung von Haar-Cascades, während moderne Methoden oft auf tiefen neuronalen Netzwerken basieren.

5.2.2 Vergleich von Methoden

In der Anfangsphase des Projekts wurde zunächst die Möglichkeit in Betracht gezogen, ein eigenes Deep-Learning-Modell für die Gesichtserkennung zu trainieren. Nach weiterer Recherche und intensiver Absprache mit Kommilitonen aus dem KI-Studiengang wurde jedoch festgestellt, dass das Training eines eigenen Modells aufgrund des hohen Zeitaufwands und der benötigten Rechenressourcen nicht praktikabel wäre.

Daraufhin wurde eine umfassende Recherche zu verschiedenen existierenden Methoden der Gesichtserkennung durchgeführt. Dabei wurden herkömmliche Ansätze wie Haar-Cascades und moderne Ansätze wie Deep Learning verglichen. Haar-Cascades, die auf der Viola-Jones-Methode basieren, bieten den Vorteil einer schnellen Berechnung und einfachen Implementierung, sind jedoch in ihrer Genauigkeit und Robustheit begrenzt. Im Gegensatz dazu bieten moderne Deep-Learning-Ansätze, wie sie in der Dlib-Bibliothek verwendet werden, eine höhere Genauigkeit und Robustheit, erfordern jedoch mehr Rechenleistung und sind komplexer in der Implementierung.

5.3 Erste Implementierung mit Haar-Cascades

5.3.1 Haar-Cascade-Ansatz

Aufgrund des begrenzten Speichers unseres Raspberry Pi wurde zunächst der Haar-Cascade-Ansatz gewählt, da dieser deutlich weniger komplex aufgebaut ist und weniger Rechenleistung erfordert. Haar-Cascades basieren auf der Viola-Jones-Methode, die einen robusten Algorithmus zur Gesichtserkennung darstellt.

Die Haar-Cascade-Methode verwendet eine Kaskade von sogenannten Haar-ähnlichen Merkmalen **haar_quelle**. Eine Kaskade in diesem Kontext bedeutet eine Abfolge von Klassifikatoren, die nacheinander angewendet werden, um die Erkennungsgenauigkeit zu erhöhen und gleichzeitig die Rechenleistung zu optimieren. Diese Merkmale sind einfache Muster, die in unterschiedlichen Größen und Positionen auf das Bild angewendet werden, um Kontraste zu erkennen, die typisch für Gesichtszüge sind.

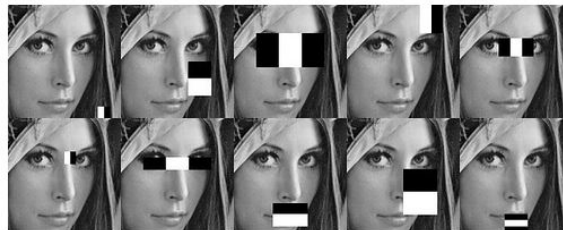


Abbildung 5.1: Beispiel verschiedener Haar-ähnlicher Merkmale
haar_cascade_example

Ein integrales Bild wird verwendet, um diese Merkmale effizient zu berechnen. Die Viola-Jones-Methode besteht aus mehreren Hauptkomponenten:

Merkmalerkennung: Haar-ähnliche Merkmale bestehen aus einfachen rechteckigen Bereichen, die Intensitätsunterschiede innerhalb des Bildes messen. Es gibt drei Arten von Haar-ähnlichen Merkmalen: Kantenmerkmale, Linienmerkmale und vierrechteckige Merkmale. Diese Merkmale helfen dabei, grundlegende Strukturen wie Kanten, Linien und Ecken zu erfassen, die in Gesichtern häufig vorkommen.

Integralbild: Das Integralbild ist eine Datenstruktur, die verwendet wird, um die Berechnung von Rechteckmerkmalen in konstanter Zeit zu ermöglichen. Dies wird erreicht, indem für jedes Pixel die Summe aller Pixelwerte oben und links davon berechnet wird. Dadurch kann jedes Rechteckmerkmal durch wenige Zugriffe auf das Integralbild effizient berechnet werden.

Adaboost-Training: Um die Merkmale zu einem starken Klassifikator zu kombinieren, wird der Adaboost-Algorithmus verwendet. Adaboost ist eine Methode des maschinellen Lernens, die eine große Anzahl schwacher Klassifikatoren zu einem starken Klassifikator kombiniert. Während des Trainingsprozesses werden die wichtigsten Merkmale ausgewählt und gewichtet, um die Erkennungsrate zu maximieren und gleichzeitig die Fehlerrate zu minimieren.

Kaskadenklassifikation: Die Klassifikatoren werden in einer Kaskade organisiert, wobei jeder Klassifikator die Aufgabe hat, ein Fenster entweder als Gesicht oder Nicht-Gesicht zu klassifizieren. Ein Fenster, das von einem Klassifikator als Nicht-Gesicht klassifiziert wird, wird sofort verworfen, was die Berechnungen erheblich beschleunigt. Nur Fenster, die von allen Klassifikatoren in der Kaskade als Gesicht erkannt werden, werden letztendlich als Gesicht klassifiziert.

Die ersten Versuche konzentrierten sich darauf, Gesichter in verschiedenen Beleuchtungssituationen und Winkeln zu erkennen, um die Robustheit des Ansatzes zu testen.

5.3.2 Vorteile und Nachteile

Nach der Implementierung des Haar-Cascade-Ansatzes in OpenCV wurde die Methode intensiv getestet, um ihre Vor- und Nachteile zu ermitteln:

Vorteile:

- **Schnelle Berechnung:** Die Methode ist sehr effizient in der Berechnung und kann in Echtzeit auf Geräten mit begrenzten Ressourcen wie dem Raspberry Pi ausgeführt werden.
- **Einfache Implementierung:** OpenCV bietet vorgefertigte Haar-Cascade-Modelle, die leicht zu integrieren sind.

Nachteile:

- **Begrenzte Genauigkeit:** Die Genauigkeit der Erkennung ist begrenzt, insbesondere bei schwierigen Lichtverhältnissen oder seitlich aufgenommenen Gesichtern.

5.3.3 Ergebnisse der Tests

Der Problem lag in der mangelhaften Genauigkeit, insbesondere bei schwierigen Lichtverhältnissen oder seitlich aufgenommenen Gesichtern. Diese Einschränkung lässt sich dadurch erklären, dass Haar-Cascades stark auf Kontraste und einfache geometrische Merkmale angewiesen sind. Bei wechselnden Lichtverhältnissen ändern sich die Intensitätsunterschiede im Bild, was dazu führt, dass die Merkmale, die für die Erkennung verwendet werden, weniger zuverlässig sind. Dies beeinträchtigt die Genauigkeit der Erkennung erheblich, da die Algorithmen Schwierigkeiten haben, die relevanten Merkmale konsistent zu identifizieren.

Zusammenfassend zeigte sich, dass die Haar-Cascade-Methode zwar effizient in der Berechnung ist und schnell auf Geräten mit begrenzten Ressourcen wie dem Raspberry Pi ausgeführt werden kann, jedoch nicht die erforderliche Präzision und Robustheit für die Gesichtserkennung in einem Smart Mirror bietet. Diese Erkenntnisse führten zur Entscheidung, nach präziseren Methoden für die Gesichtserkennung zu suchen.

5.4 Umstieg auf Dlib für höhere Präzision

5.4.1 Wechsel zu Dlib

Nach dem begrenzten Erfolg mit Haar-Cascades wurde entschieden, auf die Dlib-Bibliothek umzusteigen, um eine robustere Gesichtserkennung zu erreichen. Dlib ist eine freie Software-Bibliothek, die Algorithmen für maschinelles Lernen, Bildverarbeitung und maschinelles Sehen bereitstellt. Für die Gesichtserkennung in diesem Projekt wurde insbesondere der Histogram of Oriented Gradients (HOG)-Algorithmus und das 68-Facial-Landmarks-Modell zur präzisen Merkmalsextraktion verwendet.

5.4.2 Technologien: HOG und 68-Facial-Landmarks

Histogram of Oriented Gradients (HOG): Der HOG-Algorithmus ist eine Methode zur Merkmalerkennung in Bildern, die darauf basiert, das lokale Auftreten von Gradientenorientierungen zu zählen. Der HOG-Algorithmus bildet die Basis des vortrainierten 68-facial-landmarks-Modells. Diese Methode funktioniert folgendermaßen:

- **Gradientenberechnung:** Für jedes Pixel im Bild wird der Gradient berechnet. Der Gradient eines Pixels gibt die Richtung und die Stärke der größten Helligkeitsänderung an **hog_quelle**. Dies geschieht durch die Anwendung von Sobel-Filtern in horizontaler und vertikaler Richtung, wodurch zwei Bilder entstehen, die die Helligkeitsänderungen in x- und y-Richtung darstellen. Der Gradient kann dann durch die Kombination dieser beiden Bilder berechnet werden.
- **Zellaufteilung:** Das Bild wird in kleine Zellen unterteilt, typischerweise von 8x8 Pixeln. Diese Zellen sind klein genug, um lokale Details zu erfassen, aber groß genug, um signifikante Informationen zu enthalten. Innerhalb jeder Zelle werden die Gradientenorientierungen der Pixel gesammelt und analysiert.
- **Orientierungshistogramme:** Für jede Zelle wird ein Histogramm der Gradientenorientierungen erstellt. Die Gradienten innerhalb jeder Zelle werden in Bins sortiert, die verschiedene Richtungen repräsentieren, üblicherweise in 9 Bins,

die Winkelbereiche von 0 bis 180 Grad abdecken. Jeder Bin enthält die Summe der Gradientenstärken, die in seine Richtung fallen, was eine robuste Darstellung der Orientierungsmuster innerhalb der Zelle ermöglicht.

- **Normierung:** Um die Beleuchtungsunterschiede zu kompensieren, werden die Histogramme in Blöcken normalisiert. Ein Block besteht aus mehreren benachbarten Zellen, typischerweise 2x2. Die Normierung erfolgt durch Berechnung der Quadratwurzel der Summe der quadrierten Bin-Werte, was eine gleichmäßige Darstellung der Merkmale ermöglicht, unabhängig von lokalen Beleuchtungsunterschieden.
- **Merkmalsvektor:** Die normalisierten Histogramme aller Blöcke werden zu einem einzigen Merkmalsvektor zusammengefügt, der das Bild repräsentiert. Dieser Merkmalsvektor ist hochdimensional und enthält eine detaillierte Beschreibung der Gradientenorientierungen im gesamten Bild. Er dient als Eingabe für maschinelle Lernalgorithmen, die darauf trainiert sind, Gesichter von Nicht-Gesichtern zu unterscheiden.

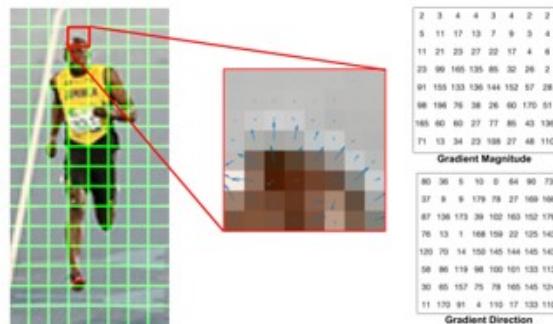


Abbildung 5.2: Beispiel einer HOG-Analyse
hoganalysis

5.4.3 Implementierung und Herausforderungen

Die Implementierung von Dlib für die Gesichtserkennung begann mit der Integration der Bibliothek und dem Einbinden der vortrainierten Modelle für HOG und Facial Landmarks. Dies stellte sich jedoch als Herausforderung heraus, da Dlib zwei große Dateien (68-Landmarks-Modell: ca. 100MB; ResNet-Modell: ca. 21MB) benötigt, die die Modelle zur Gesichtserkennung und Merkmalsextraktion enthalten. Auf dem Raspberry Pi führte dies aufgrund des begrenzten RAMs von einem GB zu erheblichen Leistungsproblemen.

Die Implementierung der Gesichtserkennung erfolgte durch die 'main.py', welche die folgenden Hauptschritte umfasst:

- **Initialisierung und Laden der Modelle:** Zunächst wurden die notwendigen Modelle geladen, darunter der Frontalgesicht-Detektor, das 68-Landmarks-Modell und das ResNet-Modell zur Gesichtserkennung und -extraktion. Diese Modelle wurden verwendet, um Gesichter im Videostream zu erkennen und deren Merkmale zu extrahieren.

- **Starten der WebSocket-Verbindung:** Eine WebSocket-Verbindung wurde initialisiert, um die Profile zwischen dem Smart Mirror und der mobilen Kontroll-App zu synchronisieren.
- **Erfassung und Verarbeitung des Videostreams:** Der Videostream wurde von der Kamera erfasst und in Graustufen umgewandelt, um die Gesichtserkennung zu erleichtern. Die Erkennung der Gesichter erfolgte durch den Frontalgesicht-Detektor.
- **Erkennung und Markierung der Gesichtspunkte:** Sobald ein Gesicht erkannt wurde, wurden die 68 Gesichtspunkte (Landmarks) ermittelt. Diese Landmarks dienten als Grundlage für die weitere Merkmalsextraktion.
- **Merkmalsextraktion mit dem ResNet-Modell:** Basierend auf den ermittelten Landmarks wurde das Face Embedding mit dem ResNet-Modell berechnet. Dieses Modell erzeugte einen 128-dimensionalen Vektor, der die einzigartigen Merkmale des Gesichts repräsentiert.
- **Gesichtserkennung und Profilsynchronisation:** Das erzeugte Face Embedding wurde verwendet, um das Gesicht zu erkennen oder ein neues Profil zu erstellen. Diese Informationen wurden anschließend gespeichert und über die WebSocket-Verbindung synchronisiert (die Logik für die Feature Extraction, das Abgleichen der Face Embeddings und die Profil- und Synchronisationslogik wird in späteren Kapiteln ausführlicher erklärt).
- **Anzeige der Ergebnisse:** Das erkannte Gesicht und die entsprechenden Landmarks wurden im Videostream hervorgehoben und mit dem erkannten Profilnamen versehen.

In seltenen Fällen trat ein Segmentation Fault auf, vermutlich weil der Stack im RAM aufgrund der umfassenden Modelle zu groß wurde. Es trat nur sporadisch auf, sodass es in der weiteren Entwicklung hintenangestellt wurde.

5.4.4 Verbesserung der Performance

Die initiale Implementierung mit Dlib ergab in qualifizierten Tests eine Performance von maximal einer Ausführung pro Sekunde. Um die Performance zu verbessern, wurden verschiedene Techniken implementiert:

- **Reduzierung der Bildgröße:** Durch die Reduzierung der Bildgröße vor der Verarbeitung soll die Berechnungszeit verringert werden.
- **Frame Skipping:** Nicht jeder Frame wurde überprüft, um die Verarbeitungslast zu reduzieren.
- **Multithreading:** Implementierung von Multithreading zur gleichzeitigen Verarbeitung mehrerer Aufgaben.

Trotz Implementierung all dieser Techniken konnte die Performance auf nur maximal zwei Ausführungen pro Sekunde angehoben werden. Die Lösung kam letztendlich durch einen Rat von Professor Metzner: "[...]Es ist egal, dass es so langsam läuft,

eine Gesichtsabfrage 1 mal pro Sekunde ist völlig ausreichend.[...]Diese pragmatische Einstellung ermöglichte es mir, mich auf die Feature Extraction zu konzentrieren, anstatt auf die Geschwindigkeit der Gesichtserkennung.

5.5 Feature Extraction und Matching

5.5.1 Feature Extraction

Um zu überprüfen, ob ein Gesicht neu oder bereits im System bekannt ist, wurde Dlib's Deep Metric Learning Ansatz verwendet. Dieser Ansatz dient der Feature Extraction und nutzt das ResNet-Modell, das speziell für die Gesichtserkennung trainiert wurde.

Vorgehensweise:

- **Landmark-Detektion:** Nach der Gesichtserkennung werden relevante Gesichtspunkte mittels des vorher bereits erwähnten Modells bestimmt. Dieses vortrainierte ML-Modell erkennt 68 charakteristische Punkte im Gesicht, wie Augen, Nase, Mund und Kieferlinie. Diese Punkte werden als Landmarks bezeichnet und helfen dabei, die Position und Ausrichtung des Gesichts zu bestimmen und dienen als Eingabe in das ResNet-Modells.
- **Berechnung des Gesichtsembeddings:** Mit Hilfe des Dlib-ResNet-Modells wird aus den extrahierten Landmarks ein Gesichtseembedding berechnet. Das ResNet-Modell verwendet die Informationen der 68 Landmarks, um einen 128-dimensionalen Vektor zu erzeugen. Dieser Vektor, das sogenannte Gesichtseembedding, repräsentiert die einzigartigen Merkmale eines Gesichts in einem hochdimensionalen Raum. Die Werte des Vektors sind als Floating Points zwischen 0 und 1 skaliert.

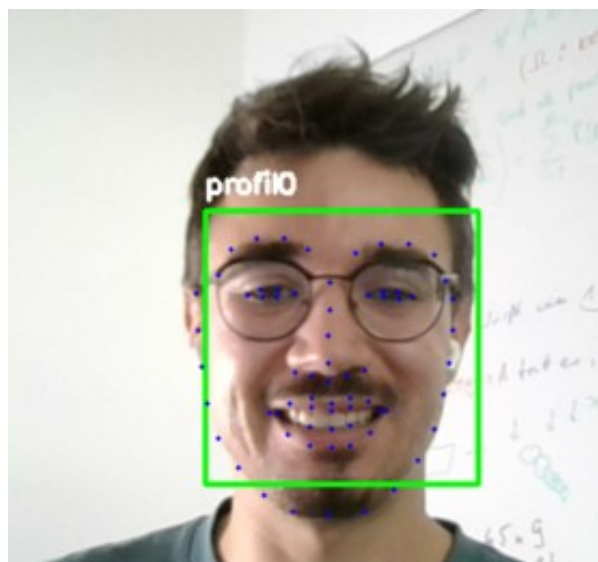


Abbildung 5.3: Praxisbeispiel der 68-landmarks
68landmarks

5.5.2 Vergleich der Gesichtsempeddings

Die erstellten Embeddings werden zur Identifikation von Personen genutzt, indem sie mit bereits gespeicherten Embeddings verglichen werden.

Vergleichsmethode: Zur Identifikation wird der euklidische Abstand zwischen den Embeddings berechnet. Der euklidische Abstand ist eine Maßzahl für die Distanz zwischen zwei Punkten in einem n-dimensionalen Raum, die durch die Wurzel der Summe der quadrierten Differenzen ihrer Koordinaten berechnet wird. Wenn der Abstand gering ist (unter einem bestimmten Schwellenwert, z.B. 0.6), wird angenommen, dass es sich um dieselbe Person handelt. Der Schwellenwert von 0,6 wurde durch eine Reihe von Tests empirisch bestimmt, um eine Balance zwischen False Positives und False Negatives zu erreichen, was eine zuverlässige Unterscheidung zwischen verschiedenen Personen ermöglichen soll.

5.6 Speichern und Verwalten der Profile

An diesem Zeitpunkt im Projekt war der Kern der Gesichtserkennung, also das Erkennen eines Gesichts, das Erstellen eines Face Embeddings und die Logik für den Vergleich mit bereits erkannten Gesichtern, fertiggestellt. Es folgte also nun die Integration in unser bestehendes System. Die Kommunikation mit dem Display-Webserver lief reibungslos, da dieser auf dem gleichen Gerät gehostet wird und somit nur der Ablagepfad der Profildaten geändert werden musste. Ein größeres Problem war jedoch die Kommunikation mit der Remote-App. Da diese ebenfalls Änderungen an den Profilen vornehmen kann, mussten wir sicherstellen, dass die Profile samt Einstellungen auf beiden Seiten synchron sind, um Informationsverlust oder Schlimmeres zu verhindern. Dafür gab es einige Besprechungen mit dem Entwickler der App, David Vollmer. Wir einigten uns auf ein genormtes Format der Profildatenspeicherung (profiles.json), welches dann über einen Websocket vom Spiegel an die Remote und vice versa gesendet werden kann.

5.6.1 Speichern der Profile

Wenn ein unbekanntes Gesichtsempedding identifiziert wird, wird ein neues Profil mit Standardwerten in der JSON-Datei erstellt. Diese Datei enthält die Informationen darüber, welche Profile existieren, welches Profil gerade aktiv ist und für jedes Profil, welche Widgets (IDs) an welcher Stelle (index) angezeigt werden sollen. Wird ein Gesichtsempedding wiedererkannt, wird lediglich das dazugehörige 'isSelected' Flag gesetzt und das vorherige gecleared.

5.6.2 Verbindung zum Websocket

Um die Profile mit der Android-Remote zu synchronisieren wurde eine Websocket-Verbindung implementiert. Diese Verbindung ermöglicht die Echtzeitsynchronisation von Profiländerungen zwischen dem Smart Mirror und der Android-App.

Vorgehensweise:

- **Websocket-Listener:** Sowohl die Remote-App als auch der Smart Mirror haben jeweils einen Listener am Websocket, um Nachrichten zu empfangen.

- **Synchronisation beim App-Start:** Beim Starten der Android-App sendet sie einen 'fetch'-Befehl über den Websocket, um sicherzustellen, dass sie sofort mit den aktuellen Profildaten synchronisiert wird. Der Smart Mirror antwortet daraufhin mit der aktuellen profiles.json-Datei.
- **Automatische Updates bei Profiländerungen:** Die profiles.json wird jedes Mal von dem Spiegel an den Websocket gesendet, wenn:
 - Ein neues Profil angelegt wurde (das neue Profil erhält automatisch das 'isSelected'-Flag).
 - Ein neues, aber bereits bekanntes Gesicht erkannt wird und das 'isSelected'-Flag entsprechend gesetzt wurde.
- **Updates von der Remote-App:** Wenn die Remote-App eine Änderung an einem Profil vornimmt, sendet sie die profiles.json an den Websocket. Der Smart Mirror empfängt diese Nachricht und ersetzt die bestehende profiles.json mit den neuen Daten.

5.7 Schlussfolgerung

5.7.1 Ausblick

Die Entwicklung der Gesichtserkennungslösung hat eine solide Grundlage geschaffen, auf der zukünftige Erweiterungen und Verbesserungen aufbauen können. Obwohl einige vielversprechende Ideen aufgrund der Stabilität des Systems nicht umgesetzt wurden, bieten sie spannende Möglichkeiten für zukünftige Arbeiten.

- **Age- und Gender-Classifiser:** Die Dlib-Bibliothek bietet auch Age- und Gender-Classifiser, die in Zusammenarbeit mit dem Widget-Team als eigenständiges Widget hätten angeboten werden können. Lokale Tests hatten dazu bereits funktioniert, und eine vollständige Implementierung könnte die Personalisierung des Smart Mirrors weiter verbessern.
- **Änderung des Profilnamens:** Mit mehr Zeit wäre eine Möglichkeit implementiert worden, den Profilnamen zu ändern. Derzeit ist der Profilname der einzigartige Bezeichner für die Face Embeddings. Um dies zu ermöglichen, müsste ein zusätzlicher eindeutiger Bezeichner eingeführt und die Profil-Logik sowohl auf dem Spiegel als auch in der App angepasst werden.
- **Flusskontrolle:** Eine weitere wichtige Verbesserung wäre die Einführung einer Flusskontrolle. Derzeit senden Spiegel und App Änderungen sofort, wenn bestimmte Ereignisse auftreten. Dies kann zu Konflikten und inkonsistenten Daten führen, wenn beide Seiten gleichzeitig senden. Eine Flusskontrolle könnte sicherstellen, dass nur eine Seite senden kann, während die andere Seite blockiert ist. Gesendete Änderungen während der Blockierphase würden in einem Puffer gespeichert und nach der Blockierung sofort gesendet.
- **Optimierung des Schwellenwerts:** Mit mehr Zeit hätten intensivere Tests zur Einstellung des euklidischen Abstands durchgeführt werden können, um Fehler

wie bei der Produktpräsentation zu vermeiden. Ein genauer kalibrierter Schwellenwert könnte die Erkennungsgenauigkeit weiter verbessern und sicherstellen, dass neue Gesichter nicht fälschlicherweise als bekannte Profile erkannt werden.

- **Datenbanksynchronisation:** Wenn wir vorher gewusst hätten, wie aufwändig die Synchronisation der Profile sein würde, hätten wir definitiv eine Datenbank verwendet, die die Synchronisation eigenständig durchführt. Eine Datenbank könnte viele der aktuellen Herausforderungen bei der Datenkonsistenz und -synchronisation lösen.

5.7.2 Zusammenfassung

Im Verlauf dieses Projekts wurde eine fortschrittliche Gesichtserkennungslösung für einen Smart Mirror entwickelt. Der Übergang von herkömmlichen Haar-Cascades zu modernen Methoden wie Dlib und HOG führte zu einer signifikanten Verbesserung der Erkennungsgenauigkeit. Trotz der Performance-Probleme konnte eine zufriedenstellende Verarbeitungsgeschwindigkeit erreicht werden.

Ein wichtiger Teil des Projekts war die Speicherung und Verwaltung der Profildaten sowie die Synchronisation mit einer Android-Remote-App über eine Websocket-Verbindung. Die Trennung der Face Embeddings von der profiles.json-Datei und die Implementierung eines robusten Synchronisationsmechanismus stellten sicher, dass die Daten auf beiden Seiten konsistent und aktuell blieben.

Insgesamt konnte eine zuverlässige Gesichtserkennung und Profilverwaltung implementiert werden.

6 | Schnittstellen

Erarbeitet von David Vollmer

In diesem Kapitel werden die Schnittstellen des **Spiegel AI** Projekts beschrieben. Im Verlauf der Entwicklung gab es mehrere Iterationen, welche das Handling der Kommunikation änderten.

6.1 Websocket zwischen Remote App und Display

Die erste Version war die Kommunikationsschnittstelle zwischen der Spiegel AI Remote und der Anzeige des Spiegel AI. Sie wurde vor der Realisierung von Profilen implementiert. Dieser Schritt diente vor allem zum praktischen Test, ob Websockets sinnvoll anzuwenden sind und ob der Datentransfer problemlos funktioniert. Um eine Verbindung mit einem Websocket-Server aufzubauen, musste ich diesen zuerst implementieren. Dafür schrieb ich eine `websocket.py`, welche für die IP-Adresse des Hosts am Port 8000 Nachrichten empfängt. Da wir hier eine State-Änderung empfangen sollten, prüften wir zunächst, ob die Nachricht in ein JSON-Format konvertiert werden konnte. Falls dies erfolgreich war, wurden die empfangenen Daten als String an alle am Websocket verbundenen Clients versendet. Um zu testen, ob die Implementierung auch funktionierte, verwendete ich das Kommandozeilen-Tool `wscat`, welches erlaubt, sich in kurzer Zeit mit einem Websocket-Server zu verbinden und Nachrichten zu versenden. Nachdem die Funktionalität bestätigt wurde, schrieb ich den `WebsocketManager`, welche für das Verwalten des Websockets in der Remote App zuständig war. Da zu dem Zeitpunkt noch keine zu empfangenden Nachrichten erwartet waren, wurden hier nur die Websocket-Verbindung und das Senden der State-Änderungen realisiert. Bei jeder Änderung, welche in der Remote View stattfand, wurde eine `action` gesendet. Für die `action` gab es die Werte `swap` und `enable`. Bei `swap` wurden die `index` der zwei getauschten Widgets mitgesendet. In der `enable`-Aktion war nur der `index` des ein- oder auszublendenden Widgets angegeben. Damit diese Nachricht in der Anzeige-App ankommt, schrieb ich einen kleinen Websocket-Handler, welcher die Verbindung mit dem Websocket-Server aufbaut und auf neue Nachrichten wartet. Wird eine Nachricht empfangen, so wird diese an eine ältere Iteration der `updateState()`-Funktion weitergegeben, in der die Bildschirmanzeige entsprechen angepasst wird.

Bei der Implementierung des Websockets gab es viele kleine Probleme. Eins, das heraussticht ist der Versuch, einen Secure Websocket zu realisieren. Hierbei erstellte ich ein sogenanntes self-signed certificate, welches zur Verschlüsselung der Kommunikation diente. Das Senden der Daten mit der Remote App lief hierbei tatsächlich problemlos, jedoch erlaubte der Chromium Browser, der als Web Browser am Raspberry Pi verwendet wird, keine Nachrichten mit selbstsignierten Zertifikaten zu empfangen. Da ich keine Möglichkeit hatte, ein Zertifikat einer „Certificate Authority“ zu besorgen,

entschloss ich mich dazu, die Kommunikation unverschlüsselt zu lassen.

6.2 Senden und Empfangen von Profilen

Der nächste Schritt der Schnittstellenimplementierung war es, gespeicherte Profile zu senden und empfangen. Das war erst möglich, nachdem die Profilverwaltung, insbesondere das Speichern der Profile, in der Remote App abgeschlossen war. Hier wurde das Senden von Nachrichten so angepasst, dass bei jeder Änderung der Profile die Nachrichten geliefert werden und nicht nur bei der Änderung des States. Der Inhalt der Nachricht war diesmal die Liste der Profile als String. Dafür mussten auch der Server und die Display-App angepasst werden. Dies führte auch zur aktuellen Implementierung der `updateState()`-Funktion.

6.3 Synchronisation der Profile

Erarbeitet von Marco Kuner und David Vollmer

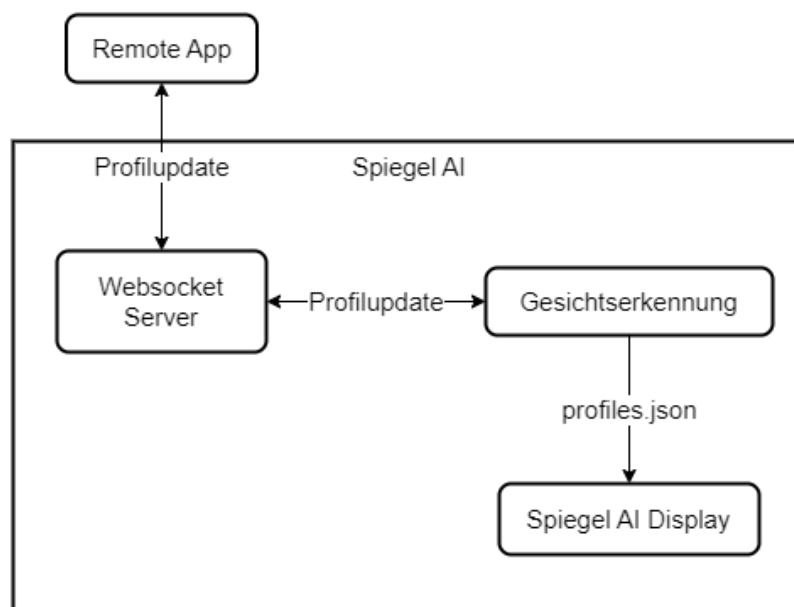


Abbildung 6.1: Darstellung der aktuellen Schnittstellen

Die finale Version der Schnittstellen wurde nach korrekter Profilverwaltung der Gesichtserkennung realisiert. Das Ziel war hier, die Websocket-Kommunikation zur Synchronisation der Profile zu nutzen, sodass das Display nicht mehr mit dem Server verbunden wird, sondern die Daten aus der lokalen `profiles.json`-Datei geladen werden. Um zu verstehen, wie eine Synchronisation erfolgreich implementiert werden kann, müssen einige Dinge berücksichtigt werden. Die Gesichtserkennung läuft dauerhaft und ist kontinuierlich mit dem Websocket verbunden. Die Remote App läuft auf einem Smartphone, welches die Verbindung beim Öffnen der App aufbaut, jedoch auch wieder trennt, wenn sie im Hintergrund läuft oder geschlossen wird. Das heißt, dass Änderungen des Profils am Spiegel AI stattfinden können, während die Remote

App noch alte Profildaten enthält. Aus diesem Grund sendet die Spiegel AI Remote eine `fetch`-Nachricht an den Server, sobald eine Verbindung aufgebaut wird. Diese Nachricht wird von der Gesichtserkennung aufgenommen und löst das Senden der aktuellen Profile des Smart Mirrors aus. Somit läuft man nicht in Gefahr, dass die alten Profildaten im Smartphone die aktuellen Daten des Spiegels überschreiben. Eine weitere Änderung im Vergleich zur vorherigen Iteration war das Angeben eines `sender`. Dies stellt sicher, dass nur Nachrichten, welche nicht die eigens gesendet wurden, verarbeitet werden.

6.4 Dynamische Widget Anordnung

Erarbeitet von: Marcel Wagner

Die Entwicklung eines Smart Mirrors mit dynamisch anpassbarer Display Anordnung stellt eine technische Herausforderung dar. Das Ziel dieses Bereiches war es, eine benutzerfreundliche und flexible Oberfläche zu schaffen, die sich den individuellen Präferenzen der Benutzer anpasst. Dieser Abschnitt beschreibt ausführlich die Methodik der Implementierung der dynamischen Display Anordnung sowie die dabei aufgetretenen Probleme und deren Lösungen.

Ein essenzieller Bestandteil des Systems war die kontinuierliche Überwachung der 'profiles.json' Datei. Diese Datei wird von der Gesichtserkennung erstellt. Auf diesen Bereich wurde im vorherigen Kapitel bereits eingegangen. Hierzu wurde ein periodischer Abrufmechanismus implementiert, der alle 200 Millisekunden die Datei abfragt. Bei jeder Abfrage wurde der aktuelle Inhalt der Datei mit dem vorherigen Zustand verglichen, um Änderungen zu erkennen. Diese Methode stellte sicher, dass Anpassungen in den Benutzerprofilen zeitnah detektiert und umgesetzt wurden.

Aus dieser Datei wird beginnend nach dem aktuellen ausgewählten Profil gesucht. Zur Identifikation des aktuell ausgewählten Profils wurde eine spezielle Funktion entwickelt. Diese Funktion durchsuchte die Liste der Profile nach dem als ausgewählt markierten Profil und gibt diese zurück. Ist kein Profil ausgewählt, wird null zurückgegeben, was die Anwendung des Standardzustands bedeutet. Die Fähigkeit, das aktive Profil zu identifizieren, war entscheidend für die Anpassung der Display Anordnung und stellte sicher, dass die Benutzereinstellungen korrekt umgesetzt wurden.

Als nächster Schritt war ein weiterer zentraler Aspekt der Implementierung eine Funktion für die Ermittlung des aktuellen Zustands der Widgets. Hierbei wird das vorher ausgelesene Profil genutzt um hierfür den Zustand und Positionierung der Widgets auszulesen. Diese flexible Handhabung ermöglichte es, die Anzeige dynamisch an die individuellen Präferenzen der Benutzer anzupassen.

Abschließend mussten noch auf Basis dieser Daten eine Dynamische Anpassung der HTML Seite vorgenommen werden. Basierend auf dem aktuellen Zustand der Widgets wurden die entsprechenden HTML Elemente ein- oder ausgeblendet und in der gewünschten Reihenfolge angeordnet. Diese Anpassungen wurden durch die

Funktion 'updateState' gesteuert, die die Widgets gemäß den Benutzereinstellungen neu positionierte. Die Funktion arbeitete folgendermaßen: Zunächst wird der Container, der die Widgets enthielt, geleert. Anschließend werden die Widgets gemäß der im Profil definierten Reihenfolge wieder hinzugefügt. Dabei wird auch die Sichtbarkeit jedes Widgets entsprechend dem enabled Status beachtet. Widgets, die nicht aktiviert waren, werden ausgeblendet, während aktivierte Widgets sichtbar blieben. Diese dynamische Anpassung ermöglichte es, die Widgets je nach Benutzerprofil in der gewünschten Anordnung und Sichtbarkeit darzustellen.

Aufgetretene Probleme und deren Lösung

Cache Verwaltung: Das am häufigsten aufgetretene Problem war das der Cache Verwaltung des Browsers. Dies stellte eine besondere Herausforderung dar, da durch die regelmäßigen Anfragen an die 'profiles.json' Datei oft veralteter Inhalt aus dem Cache verwendet wurde, anstatt die neuesten Daten abzurufen. Dieses Problem wurde durch gezielte Deaktivierung des Caches für die betreffenden Anfragen gelöst. Der HTTP Header Cache Control wurde entsprechend konfiguriert, um sicherzustellen, dass die Anfragen stets frische Daten zurückliefert. Dies gewährleistete, dass immer die aktuellste Version der profiles.json Datei abgerufen und verarbeitet wurde, was eine zuverlässige Aktualisierung der Anzeige ermöglicht.

CORS Beschränkung: Ein weiteres Problem, welches während der Implementierung aufgetreten ist, war im Zusammenhang mit der Same Origin Policy des Browsers. Diese Sicherheitsrichtlinie verhinderte den Abruf der profiles.json Datei von einem anderen Ursprung, was die Aktualisierung der Profile erschwerte. Um dieses Problem zu umgehen, wurde ein lokaler Server mit Python erstellt, der die Datei ausliefert. Dieser Server ermöglichte es, die Datei lokal zu hosten und somit die CORS Beschränkungen (Cross Origin Resource Sharing) zu umgehen. Durch den Zugriff auf diesen lokalen Server konnte die Datei problemlos und sicher abgerufen werden, was die zuverlässige Aktualisierung der Profile gewährleistet.

Fehler bei der Skript ausführung: Während der Implementierung der Dynamischen Widget Anordnung trat ein weiterer Fehler auf, der verhinderte, dass das Skript in der 'updateState.js' Datei ausgeführt wurde. Um das Problem zu beheben war es für mich notwendig das Skript manuell in den code auszuführen.

Durch diese manuelle Einbindung konnte der Fehler erfolgreich behoben und die korrekte Funktion des Skriptes sichergestellt werden. Diese Fehlerbehebung war notwendig um den weiteren Verlauf der Widget Anordnung zu gewährleisten.

7 | Ergebnisse

In diesem Kapitel fassen wir die Ergebnisse des Projekts **Spiegel AI** zusammen. Wir gehen auf die erreichten Ziele, die Herausforderungen und die zukünftigen Arbeiten ein.

7.1 Zusammenfassung der Ergebnisse

7.1.1 Erreichte Ziele

Unser Projektteam hat den Smart Mirror erfolgreich realisiert und dabei folgende Ziele erreicht:

Der Spiegelprototyp wurde ansehnlich und robust aufgebaut. Er ist nicht nur funktional, sondern fügt sich auch optisch ansprechend in die Umgebung ein. Die Display-Ausgabe funktioniert einwandfrei und ermöglicht eine klare und deutliche Darstellung aller Inhalte.

Der Smart Mirror ist in der Lage, die aktuelle Wetterlage des Ortes sowie Kalendereinträge anzuzeigen und bietet viele weitere informative und nützliche Funktionen. Benutzer des Spiegels haben die Möglichkeit, ihr angezeigtes Profil individuell zu gestalten. Dies erfolgt durch die Auswahl aus neun verschiedenen Widgets, die auf acht unterschiedlichen Positionen platziert werden können. Diese Personalisierung ermöglicht eine hohe Flexibilität und Anpassungsfähigkeit an die individuellen Bedürfnisse und Vorlieben der Benutzer.

Ein weiteres herausragendes Merkmal ist die interaktive Steuerung über eine mobile Kontroll-App. Diese App ermöglicht es den Benutzern, ihr Profil zu ändern oder zu personalisieren, was die Benutzerfreundlichkeit und den Komfort erheblich steigert. Alle Informationen werden in deutscher Sprache angezeigt, was die Bedienung für deutschsprachige Nutzer intuitiv und einfach macht.

Ein neues Gesicht wird erkannt und das jeweilige Profil angezeigt, was eine schnelle und reibungslose Nutzung sicherstellt. Die Benutzeroberfläche wurde in Tests als 'sehr intuitiv' eingestuft, was die hohe Benutzerfreundlichkeit des Systems unterstreicht.

Technisch konnten wir sicherstellen, dass sowohl der Spiegel als auch die App erfolgreich eine Verbindung zum Websocket herstellen können. Beide Komponenten senden nach einer ausgeklügelten Synchronisationslogik jeweils den aktuellen Stand der Profile, wodurch eine konsistente und aktuelle Datenbasis gewährleistet wird. Die

Gesichtserkennung arbeitet äußerst zuverlässig und erkennt Personen zu 95 Prozent korrekt, wodurch die personalisierten Profile geladen und angezeigt werden können.

7.1.2 Herausforderungen

Die Herausforderungen, denen sich das Team während der Entwicklung des Projekts stellen musste, sind in den jeweiligen Dokumentationen der einzelnen Projektteilnehmer ausführlich beschrieben.

7.1.3 Zukünftige Arbeiten

Für zukünftige Arbeiten und mögliche Erweiterungen des Projekts verweisen wir ebenfalls auf die detaillierten Ausführungen in den Dokumentationen der einzelnen Projektteilnehmer.

7.1.4 Betriebssetzung

Zum Verwenden des Spiegel Als sind einige Schritte notwendig. Zuerst benötigen Sie eine Kamera und einen WLAN-Stick, da diese Komponenten Eigentum der Projektteilnehmer waren, welche sie wieder mitnahmen. Beim Verwenden eines WLAN-Sticks ist noch zu beachten, dass gegebenenfalls Treiber installiert werden müssen. Nachdem die Hardware bereit ist, sind die folgenden Schritte zur Betriebssetzung des Smart Mirrors nötig:

1. Schließen Sie alle über USB zu verbindenden Komponenten, inklusive Tastatur und Maus, am Raspberry Pi an.
2. Schalten Sie den Raspberry Pi ein
3. Öffnen Sie ein Terminal-Fenster und navigieren Sie zu `~/Mirror/dt-g5/`
4. Öffnen Sie zwei neue Tabs
5. Navigieren Sie im ersten Tab zu `~/Mirror/dt-g5/websocket_server` und starten Sie den Websocket mit `python websocket.py`
6. Navigieren Sie im zweiten Tab zu `~/Mirror/dt-g5/Display` und starten Sie einen HTTP-Server mit `python -m http.server 8001`
7. Navigieren Sie im dritten Tab zu `~/Mirror/dt-g5/facialRec/facialRecReadability/mitWebsocket` und starten Sie die Gesichtserkennung mit `python main.py`
8. Öffnen Sie den Chromium Browser und rufen Sie `http://localhost:8001` auf
9. Eventuell müssen Sie hier den Cache bereinigen, falls die Anzeige sich nicht ändert
10. Installieren Sie die Android-APK auf einem Smartphone. Sie ist im GitLab-Repository im Verzeichnis `remote_app` hinterlegt
11. Starten Sie die Applikation auf dem Smartphone

Stundenliste

Stundenliste Leon Kranner

Kalenderwoche	Stunden	Aufgabe
12	3	Einführungsveranstaltung
	3	Requirements + Materialliste
13	3	GANNT Diagramm
	2	Teambesprechung
14	2	Planung mit Hardware-Team
	2	Teambesprechung
	1	Requirements überarbeiten
	3	Planung Produkterstellung
15	3	Postererstellung und Hw
	2	Displaymessung + Postervorstellung
	2	Teambesprechung
	1	Baumarkt
	2	Besprechung Spiegelrahmen
16	4	Display-Projekt aufsetzen
	2	Baumarkt Materialien erkunden
	2	Planung/Besprechung Bilderrahmen
	2	Teambesprechung
17	2	Umstrukturierung des Display-Projekts
	2	Neues Widget erstellen
	3	Projektbesprechung und weitere Programmierung
	2	Teambesprechung
18	6	Weitere Widgets und Änderungen an alten Widget
	2	Bugfixing beim Kalender
	4	Holz auf Maß schneiden und hobeln
	3	Vorbohren und Bilderrahmen zurechtlegen
19	3	Einkerbungen fräsen
	1	MDF Platte auf Maß schneiden
	1	Zwischenstücke vorne anschrauben
	1	Teambesprechung

Fortsetzung auf nächster Seite

Tabelle 7.1 – Fortsetzung von vorheriger Seite

Kalenderwoche	Stunden	Aufgabe
	2	Holz zusammenleimen und trocknen lassen
20	2 1	Teambesprechung Umstrukturierung des Stundenplans
21	2 1	Teambesprechung Plexiglas auf Maß schneiden
22	2 7	Teambesprechung Display dynamisch gestalten + Automatische Aktualisierung der Widgets
23	2 2	Automatische Aktualisierung der Widgets Teambesprechung
24	2 2 2	Teambesprechung Besprechung Schnittstellen Bugfixing Kalender Widget (Anzeigen des aktuellen Tages)
25	2 1 2 1 3 3	Teambesprechung Plexiglas überarbeiten Löcher bohren und Plexiglas festschrauben Spiegelfolie aufbringen und Kabel-Loch bohren Fine tuning für Display (Anpassung Widget: Größe, Formatierung, Anordnung, etc.) Aufbau und Testen des Displays
26	2 1 3 1 1 2 2 1 1	Teambesprechung Fehler korrigieren am Spiegel Austausch der Spiegel Folie, Aufbau der Spiegels, Hardware installieren Testen des Displays mit Aufgebauten Spiegels Besprechung Präsentation Vorbereitung Präsentation Bilder und Aufbau Raspberry Pi Gesichtserkennung testen Anpassung des Kalenders: Monate nebeneinander anzeigen statt untereinander Anpassung der Wettervorhersage: nur 3 Tage und nicht untereinander
27	2 1	Powerpoint vorbereiten Text für Präsentation vorbereiten

Fortsetzung auf nächster Seite

Tabelle 7.1 – Fortsetzung von vorheriger Seite

Kalenderwoche	Stunden	Aufgabe
	1	Vorbereitung Präsentation
	3	Final Version von Projekt auf Raspberry spielen + Testen aller Funktionen
	2	Teambesprechung
	1	Stundenliste in Dokumentation eintragen
	2	Informieren über Mögliche Testvarianten für Widget
	1	Einleitung der Dokumentation schreiben
	3	Verschieden Testvarianten bei den Widgets durchführen
28	2	Dokumentation für Hardware schreiben
	3	Dokumentation für Widgets
	3	Dokumentation für Rahmen
	2	Dokumentation für Display-Aufbau schreiben + Kapitel formatieren in Latex
	1	Testverfahren für Doku schreiben

Gesamtstunden: 151

Stundenliste Marco Kuner

Kalenderwoche	Stunden	Aufgabe
12	3	Einführungsveranstaltung
13	3	GANNT Diagramm
14	3 2 3	Teileliste / GANNT Diagramm Teambesprechung Inventur
15	10 3 2	Technologie-Recherche + Postererstellung Posterdemütigung ertragen und HW Teambesprechung
16	2 8	Teambesprechung Erster Prototyp mit HAAR Cascades
17	2 8	Teambesprechung Neue Version mit DLIB Bibs geschrieben
18	2 2 4	Teambesprechung Recherche über facial Landmark Storage Neue Iteration mit Storage Technologie
19	10 2	Troubleshoot da extrem langsam Teambesprechung
20	2 4	Teambesprechung Recherche zu geeigneter Schnittstelle und Format der Profilerstellung mit profile landmarks
21	2	Teambesprechung
22	2	Teambesprechung
23	2	Teambesprechung
24	2	Teambesprechung
25	2 6 2	Teambesprechung Implementieren einer Lösung zur automatischen Erkennung eines neuen Gesichts und output der Daten in .json Schnittstellen Thinktank mit David
26	2 5 6	Teambesprechung Ausgabe und automatische Aktualisierung einer genormten profiles.json Implementierung eines neuen Websockets zwischen Raspi und Android in Vorbereitung zur Synchronisation

Fortsetzung auf nächster Seite

Tabelle 7.2 – Fortsetzung von vorheriger Seite

Kalenderwoche	Stunden	Aufgabe
	2	Recherche zu Technologien zur Synchronisation zwischen Raspi und Android (inotify?)
	2	Besprechung mit remote app Spezialist bzgl. Synchronisationsproblemen
27	4	Vor- und Aufbereiten der Präsentation
	2	Verbessern der readability des Algorithmus
	8	Implementation des Websockets mitsamt Logik für andauernder Synchronisation
	4	Troubleshooting: Gesichtserkennung stürzt ab auf Raspi
	3	Testing
	6	Dokumentation Gesichtserkennung
	3	Literatur+Abbildungsverzeichnis
	2	Zusammenfassen der Ergebnisse

Gesamtstunden: 142

Stundenliste David Vollmer

Kalenderwoche	Stunden	Aufgabe
12	3	Einführungsveranstaltung
	4	Setup Gitlab und Drafts
13	4	Erstellung GANNT Diagramm und Lastenheft
	2	Teambesprechung
14	5	Abgabevorbereitung GANNT und Lastenheft
	2	Teambesprechung
	3	Hardwarediskussion und -suche
	2	Überarbeitung GANNT und Lastenheft
15	4	Postererstellung
	3	Vostellung Poster und Hardware suche
	6	Setup Flutter und Frontend dev
	2	Teambesprechung
	2	Frontend dev (Navigation)
16	2	Teambesprechung
	3	Frontend dev
17	2	Setup Raspberry Pi
	2	Teambesprechung
18	2	Teambesprechung
19	2	Teambesprechung
	1	Frontend dev (Widget buttons)
20	2	Teambesprechung
21	2	Teambesprechung
22	2	Teambesprechung
	4	Troubleshooting Android SDK
23	2	Teambesprechung
24	2	Besprechung Schnittstellen
	4	Frontend dev (Widgets final)
	2	Teambesprechung
	2	Konfiguration Raspberry Pi
	8	Konfiguration Schnittstellen (Flutter + Server)
	5	Konfiguration Schnittstellen (Spiegel + Server)
25	2	Teambesprechung
	3	Konfiguration Raspberry Pi wifi
	5	Troubleshooting + Testing Websocket
	2	Anpassung Android und iOS (icon, splash, usw.)
	8	Anpassung Datenspeicher, Profile und Websocket

Fortsetzung auf nächster Seite

Tabelle 7.3 – Fortsetzung von vorheriger Seite

Kalenderwoche	Stunden	Aufgabe
	3	Speichern von Widget- und Remotestatus in Profilen
	3	Code Refactoring und Bugfixing
26	1	Besprechung Schnittstellen Profile
	1	Überarbeitung Poster
	2	Teambesprechung
	1	Überarbeitung Websocket-Message
	2	Troubleshooting selected Widgets
	2	Konfiguration Raspberry Pi
	6	Gesichtserkennung Erstellung Powerpoint
27	2	Besprechung profiles sync
	1	Refactoring File Reader
	2	Implementierung profiles sync
	3	Schreiben des Präsentationsskripts
	2	Übung Präsentation
	5	Testen der Gesichtserkennung am Websocket
	4	Testen und Korrigieren profiles sync
	2	Überarbeitung Dokumentationslayout
	5	Implementierung und Testen selectedWidgets
	1	Kompilieren Remote App Release
	2	Dokumentation Remote App Views
	2	Anpassung des Literaturverzeichnis in Dokumentation
	1	Dokumentation Flutter SDK
	2	Dokumentation Implementierung Websocket und Remote Content
28	2	Dokumentation Raspberry Pi Konfiguration
	3	Dokumentation Implementierung Widgets, Profile und Sonstiges
	2	Dokumentation Schnittstellen
	1	Dokumentation Fertigstellung

Gesamtstunden: 174

Stundenliste Marcel Wagner

Kalenderwoche	Stunden	Aufgabe
12	3	Einführungsveranstaltung
13	3	GANNT Diagramm
	2	Teambesprechung
	2	Vorbereitung Template
14	2	Teambesprechung
	3	Planung und Hardware Suche
	3	Setup CAD und ersten Entwurf zeichnen
15	3	Poster Erstellung
	2	Besprechung Spiegelrahmen
	2	Detaillierung der CAD Datei
	1	Baumarkt
	2	Teambesprechung
16	3	Einführung Display Programmierung und erste Ansätze
	2	Planung und Besprechung für den Bilderrahmen
	2	Materialien im Baumarkt suchen
	2	Teambesprechung
17	4	Weitere Setup für Display Programmierung
	4	Projekt Besprechung und weitere Programmierung
	4	Weitere Widget Programmierung und Bug Fixing
	2	Teambesprechung
18	2	Display Programmierung (Fertigstellung des Verkehrsinformations Widget)
	2	Teambesprechung
	4	Holz auf Maß schneiden und hobeln
	3	Vorbohren und Bilderrahmen zurechtlegen
19	3	Einkerbungen fräsen
	1	MDF Platte auf Maß schneiden
	1	In MDF Platte Löcher bohren für Befestigung
	1	Zwischenstücke vorne anschrauben
	2	Teambesprechung
	2	Holz zusammenleimen und trocknen lassen
20	2	Teambesprechung
	1	Bug Fixing von älteren Widgets
21	2	Teambesprechung

Fortsetzung auf nächster Seite

Tabelle 7.4 – Fortsetzung von vorheriger Seite

Kalenderwoche	Stunden	Aufgabe
	1	Plexiglas auf Maß schneiden
22	2	Teambesprechung
23	3	Erstellung weiter Widgets
	2	Teambesprechung
24	2	Teambesprechung
	2	Besprechung Schnittstellen
	2	Bugfixing für das News Widget
25	2	Teambesprechung
	1	Bugfixing der Widget Ansicht
	1	Plexiglas überarbeiten
	2	Löcher bohren und Plexiglas festschrauben
	1	Spiegelfolie aufbringen und Kabel Loch bohren
	5	Schnittstelle zwischen Gesichtserkennung und Display die Grundlagen auf Seite des Displays aufsetzen
	2	Bugfixing Schnittstellen Problem
26	2	Teambesprechung
	1	Fehler korrigieren am Spiegel
	1	Bugfixing Widget
	3	Austausch der Spiegel Folie, Aufbau der Spiegels, Hardware installieren
	1	Testen des Displays mit Aufgebauten Spiegels
	1	Vorbereitung Präsentation
	2	Vorbereitung Präsentation (Bilder und Aufbau Finalisieren)
	2	Raspberry Pi Gesichtserkennung Testen
	5	Profile aus der Gesichtserkennung auslesen und Speicherin in Raspberry
	2	Bugfixing Browser Problem
27	2	Powerpoint erstellung
	3	HMTL neu anordnen auf Basis von Json Datei
	2	Troubleshooting Cache Probleme
	1	nicht ausgewählte Widgets ausblenden
	2	Vorbereitung Präsentation
	3	Schreiben des Präsentationsskripts
	2	Teambesprechung

Fortsetzung auf nächster Seite

Tabelle 7.4 – Fortsetzung von vorheriger Seite

Kalenderwoche	Stunden	Aufgabe
	1	Studenliste in Dokumentation eintragen
	4	Dokumentation für Widgets schreiben
	1	Dokumentation für Testverfahren schreiben
	2	Informieren über Mögliche Testvarianten für Widget / Schnittstelle
	3	Verschieden Testvarianten bei den Widgets durchführen
	2	Dokumentation der Dynamischen Widget Anordnung / Überarbeitung Layout
	2	Dokumentation für Hardware schreiben
	3	Dokumentation für Rahmen
	1	Quellen aktualisieren und kleine Fehlerbehebungen in der Dokumentation
	3	Final Version von Projekt auf Raspberry spielen und Testen aller Funktionen für die Vorstellung
28	1	Dokumentation der Fehler bei der Implementierung der Dynamischen Widget anordnung
	1	Korrektur der Fertigen Dokumentation

Gesamtstunden: 166

Literaturverzeichnis

Abbildungsverzeichnis

1.1	Raspberry Pi Model B	4
1.2	Logitech Kamera	5
1.3	Dell Monitor	5
2.1	Rahmen wurde geleimt	10
2.2	Fertiggestellter Rahmen	10
2.3	Raspberry Pi am Rahmen befestigt	12
2.4	Spiegel AI	12
3.1	Display Layout	13
3.2	Termine Widget	15
3.3	Kalender Widget	16
3.4	Wettervorhersage Widget	18
3.5	Notizen Widget	19
3.6	Uhrzeit Widget	21
3.7	Verkehrsinformations Widget	22
3.8	News Widget	23
3.9	Tankstellen Widget	24
4.1	JetBrains Entwicklerumfrage 2023 jetbrains_survey	28
4.2	Remote Ansicht	29
4.3	Widgets Ansicht	29
4.4	Profile Ansicht	29
4.5	Verzeichnisstruktur der <code>lib</code> im Spiegel AI Remote Projekt	31
4.6	Beispiel eines State-Eintrags im JSON-Format	32
5.1	Beispiel verschiedener Haar-ähnlicher Merkmale	36
5.2	Beispiel einer HOG-Analyse	39
5.3	Praxisbeispiel der 68-landmarks	41
6.1	Darstellung der aktuellen Schnittstellen	46