Validierungsbericht

Phase	Verantwortlicher	E-Mail
Pflichtenheft	Florian Lorenz	lorenz@fim.uni-passau.de
Entwurf	Andreas Wilhelm	wilhelma@fim.uni-passau.de
Spezifikation	Andreas Poxrucker	poxrucke@fim.uni-passau.de
Implementierung	Martin Freund	freund@fim.uni-passau.de
Validierung	Florian Bürchner	buerchne@fim.uni-passau.de
Präsentation	Max Binder	binder@fim.uni-passau.de

21. Januar 2011

Inhaltsverzeichnis

Zusammenfassung

Dieses Dokument soll Aufschluss über alle durchgeführten Testfälle geben. Das Projekt wurde zum einen durch *Unit*-Tests als auch durch *Black-Box*-Tests geprüft. Der C-Code wurde mithilfe eines Terminals getestet. Alle hier aufgeührten Testfälle sind verbindlich und wurden unter größter Sorgfalt durchgeführt.

1 Globale Testszenarien und Testfälle

1.1 Pflichtenheft Testfälle

1.1.1 /T50/ Kalibrierung

Nach nicht ordnungsgemäßer Kalibrierung beginnen die Außen-LEDs nacheinander zu leuchten (/F180/).

1.1.2 /T60/ Linienerkennung

Der e-puck wird im Betrieb entlang einer Linie aufgestellt und folgt dieser in Richtung der Liniensensoren (/F100/).

1.1.3 /T70/ Bluetooth-Scan

Mehrere e-puck-Roboter werden auf dem Spielfeld platziert und eingeschaltet. Nach der Initialisierungsphase sollen sämtliche Teilnehmer auf dem Smartphone erkannt werden (/F200/).

1.1.4 /T80/ Broadcast-Test

Gemäß /T70/ werden alle verbindungsbereiten e-puck Roboter auf dem Suchdialog des Smartphones angezeigt. Es wird ein Roboter ausgewählt, alle im Netzwerk befindlichen Teilnehmer werden im Dropdown-Steuerelement aufgelistet (/F50/, /F60/, /F65/, /F70/, /F205/, /F210/).

1.1.5 /T90/ Knotenanalyse und manuelle Steuerung per On-Screen-Joystick

Der Roboter wird gemäß /T60/ aufgestellt. Durch manuelle Steuerung per On-Screen-Joystick muss der Benutzer auf allen Knotentypen die möglichen Fahrtrichtungen testen. Nur gültige Richtungen dürfen vom Roboter befahren werden (/F80/, /F110/, /F220/, /F270/).

1.1.6 /T100/ Steuerung der Fahrtgeschwindigkeit

Dieser Test konnte aufgrund der geänderten Funktionalitäten nicht durchgeführt werden.

1.1.7 /T110/ Steuerung per Beschleunigungssensor

Der Test wird analog zu /T80/ durchgeführt, wobei als Steuerungsmethode der Beschleunigungssensor verwendet wird (/F280/).

1.1.8 /T120/ Erkundungstest

Mehrere e-puck Roboter werden auf entsprechende Startpositionen innerhalb des Spielfelds gesetzt und eingeschaltet. Ziel des Testszenarios ist die vollständige Erkundung durch effiziente Zusammenarbeit der Teilnehmer. Falls die Roboter nach Abschluss auf die Startpositionen zurückkehren und die Karte auf dem Smartphone dargestellt wird, ist der Testfall erfolgreich (/F90/, /F120/, /F130/, /F135/, /F140/, /F150/, /F155/, /F160/, /F190W/, /F230/, /F320/, /F330/).

1.1.9 /T130/ Erweiterter Steuerungstest

Der Benutzer wählt nach der Lokalisierung von mindestens zwei Roboter einen per Dropdown-Steuerelement aus. Anschließend wird die Steuerung über beide Steuerungsarten (/F270/, /F280/) durchgeführt. Im nächsten Schritt wird eine anderer Roboter gewählt und die Steuerung mit diesem geprüft (/F290/, /F300/, /F310/).

1.1.10 /T140W/ Speichern der Kartendaten

Nachdem eine Kartenansicht auf dem Smartphone verfügbar ist, speichert der Benutzer die Kartendaten ab (/F250W/).

1.1.11 /T150W/ Laden der Kartendaten

Nachdem Kartendaten auf dem Smartphone gespeichert wurden (/T120W/), lädt der Benutzer diese. Daraufhin wird die entsprechende Karte angezeigt (/F260W/).

1.1.12 /T160W/ Test von globaler Lokalisierung

Dieser Test konnte aufgrund der geänderten Funktionalitäten nicht durchgeführt werden.

1.1.13 /T170W/ Test der Zoomfunktionalität

Dieser Test konnte aufgrund der geänderten Funktionalitäten nicht durchgeführt werden.

1.2 Weitere Testfälle

1.2.1 Test der Autoskalierung

Es werden nacheinander zwei unterschiedlich große Karten mittels Import-Funktion (/F260W/) geladen. Beide Karten werden nun autoskaliert und vollständig angezeigt.

1.2.2 Test der Kartenüberprüfung auf Gültigkeit

Es werden ungültige Karten erstellt und nacheinander geladen. Es wird eine Meldung ausgegeben, dass die Karte nicht gültig ist.

2 Unit Tests

2.1 Android Anwendung

2.1.1 GridMap

- `testInsertNode()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird die GridMap in eine Liste umgewandelt und deren Größe abgefragt, um sicherzustellen, dass der Knoten eingefügt wurde.
- `testFrontierNodeRightT()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob drei Frontierknoten eingefügt wurden.
- `testFrontierNodeLeftT()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob drei Frontierknoten eingefügt wurden.
- `testFrontierNodeTopT()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob drei Frontierknoten eingefügt wurden.
- `testFrontierNodeBottomT()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob drei Frontierknoten eingefügt wurden.
- `testFrontierNodeCross()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob vier Frontierknoten eingefügt wurden.
- `testFrontierNodeBottomRightEdge()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob zwei Frontierknoten eingefügt wurden.
- `testFrontierNodeBottomLeftEdge()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob zwei Frontierknoten eingefügt wurden.
- `testFrontierNodeTopRightEdge()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob zwei Frontierknoten eingefügt wurden.
- `testFrontierNodeTopLeftEdge()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird durch die GridMap eine Liste mit Frontierknoten erstellt und überprüft, ob zwei Frontierknoten eingefügt wurden.

- `testUpdateNode()` Es werden zwei neue Knoten erstellt und die GridMap eingefügt. Der zuletzt Eingefügte wird abgerufen und sein Knotentyp überprüft. Außerdem wird durch die GridMap erneut eine Liste mit Frontierknoten erstellt und deren Größe, die abhängig von der Wahl der Knotentypen ist, überprüft.

- `testMapBorders()` Es werden vier neue Knoten erstellt und in die GridMap eingefügt. Je nach Wahl der Koordinaten dieser Knoten wird hier die Größe des Spielfeldes überprüft. Es werden vier Werte verglichen, der minimale x Wert, der maximale x Wert, der minimale y Wert und der maximale y Wert.

- `testSerialieMapInString()` Ein neuer Knoten wird erstellt und in die GridMap eingefügt. Anschließend wird die serialisiert und in einem String Array gespeichert. Dieses Array beinhaltet nun den x, den y Wert und den Knotentyp der zuvor eingefügten Knoten, in genau dieser Reihenfolge. Es wird überprüft, ob sich diese Werte entsprechen.

2.1.2 Behaviour

- `testExploreBehaviour()`

2.1.3 ComManager

- `testAddClientAndSend()` Für diesen Testfall muss ein Testclient erstellt werden, der anschließend dem ComManager hinzugefügt wird. Über den ComManager wird nun an den Testclient über `broadcast()` eine HelloRequest Nachricht gesendet. Diese wird vom Testclient ausgelesen und mit der vorher erzeugten HelloRequest Nachricht verglichen.

- `testRemoveClient()` Hier wird dem ComManager ebenfalls ein Testclient hinzugefügt, aber anschließend auch gleich wieder entfernt. Dann wird wie im obigen Testfall ein Broadcast mit einer HelloRequest Nachricht versendet. Am Testclient wird nun versucht die Nachricht auszulesen. Dies ist nicht erfolgreich, da andernfalls das Entfernen vom Testclient nicht funktioniert hätte.

2.1.4 AStarPathFinder

- `testFindPuckMapNodeMapNodeArray()` Für diesen Testfall wird ebenfalls eine vollständig, vorgegebene Karte erstellt. Außerdem wird ein Start- sowie Zielknoten festgelegt. Der A-Stern Algorithmus wird anhand diesen Parameterwerten gestartet und nach Abschluss geprüft, ob der gefundene Pfad (inklusive exakter Wegkosten) dem erwarteten Ergebnis entspricht.

2.1.5 Handler

- `testSimTurnHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf REQUEST_TURN gesetzt. Anschließend wird mit dem Array ein neuer VirtualPuckRequest erzeugt und an

den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testSimStatusHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf REQUEST_STATUS gesetzt. Anschließend wird mit dem Array ein neuer VirtualPuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testSimSpeedHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf REQUEST_SPEED gesetzt. Anschließend wird mit dem Array ein neuer VirtualPuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testSimResetHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf REQUEST_RESET gesetzt. Anschließend wird mit dem Array ein neuer VirtualPuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testSimMoveHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf REQUEST_MOVE gesetzt. Anschließend wird mit dem Array ein neuer VirtualPuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testSimLEDHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf REQUEST_LED gesetzt. Anschließend wird mit dem Array ein neuer VirtualPuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testPuckStatusHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf RES_STATUS gesetzt. Anschließend wird mit dem Array ein neuer PuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testPuckRejectHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf RES_REJECT gesetzt. Anschließend wird mit dem Array ein neuer PuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.

- `testPuckOkHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf RES_OK gesetzt. Anschließend wird mit dem Array ein neuer PuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.
- `testPuckNodeHitHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf RES_HIT_NODE gesetzt. Anschließend wird mit dem Array ein neuer PuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.
- `testPuckCollisionHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf RES_COLLISION gesetzt. Anschließend wird mit dem Array ein neuer PuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.
- `testPuckAbyssHandler()` Es wird eine neue Nachricht im verwendeten Nachrichtenformat (32 Byte Array) erzeugt. Die ersten beiden Bytes der Nachricht (Nachrichtenheader) werden auf RES_ABYSS gesetzt. Anschließend wird mit dem Array ein neuer PuckRequest erzeugt und an den Handler weitergegeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtenheaders erkennt.
- `testFailureRequestHandler()` Es wird eine neue FailureRequest-Nachricht erstellt und an den Handler übergeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtentyps erkennt.
- `testDriveRequestHandler()` Es wird eine neue DriveRequest-Nachricht erstellt und an den Handler übergeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtentyps erkennt.
- `testControlledRequestHandler()` Es wird eine neue ControlledRequest-Nachricht erstellt und an den Handler übergeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtentyps erkennt.
- `testCollisionRequestHandler()` Es wird eine neue CollisionRequest-Nachricht erstellt und an den Handler übergeben. Der Test ist bestanden, wenn der Handler die Nachricht anhand des Nachrichtentyps erkennt.

2.2 e-puck Firmware

2.2.1 Ringpuffer

Auf dem Ringpuffer wurden die üblichen Funktionen, wie `pop()` und `push()` getestet. Zunächst wurde auf den leeren Puffer ein pop ausgeführt. Als Statusmeldung wurde -1 zurückgeliefert, was den Erwartungen entsprach `(Size: 32 Usage: 0 Free: 32)`. Dann wurden die Buchstaben a bis z gepusht `(IsEmpty: False Size: 32 Usage: 26 Free: 6)`. Anschließend wurden die Zahlen 0 bis

5 gepusht, um den Puffer zu füllen (`IsFull: True`). Es wurde versucht ein weiteres Element zu pushen. Dies wurde vom Ringpuffer angelehnt und die Nachricht wurde verworfen. Zum Ende wurden alle Elemente gepopt (`Size: 32 Usage: 0 Free: 32`).

3 Blackbox Tests

3.1 Simulator

Es wird eine Karte erstellt und im Simulator geladen. Anschließend wird eine Erkundung mit mehreren e-puck Robotern durchgeführt. Nach erfolgreicher Erkundung fahren die virtuellen Roboter auf ihre Ausgangsposition zurück.

3.2 Mehrfach überfahrene Knoten

Es wird eine Karte erstellt, die ein Roboter nicht abfahren kann, ohne einen Knoten mehrfach zu befahren. Die Erkundung wird dann mit mehreren Roboter gestartet und geprüft, ob sich ein Knoten, wenn er mehrfach überfahren worden ist verfärbt.

3.3 Steuerung

Nach erfolgreichem Verbindungsaufbau eines e-puck Roboters mit dem Smartphone wird zum Dialog *Steer* gewechselt und der Roboter in alle vier Richtungen gesteuert. Der Vorgang wird mit der zweiten Steuerungsart wiederholt.

3.4 Korrektes Anzeigen aller Schaltflächen und der Karte

Es werden gültige Karten erstellt und nacheinander geladen und geprüft, ob sie korrekt dargestellt werden. Beim Verbinden mit den e-puck Robotern werden nacheinander alle angezeigten Roboter an- und wieder abgeählt. Diese müssen farblich jeweils hervorgehoben werden. Anschließend wird zum Dialog *Steer* gewechselt und jeder Button für die Richtungen nacheinander gedrückt und validiert, ob die jeweilige Auswahl hervorgehoben wird.

3.5 Menüführung

Von jeder Activity wird über das Menü zu allen anderen erreichbaren Dialogen gewechselt.

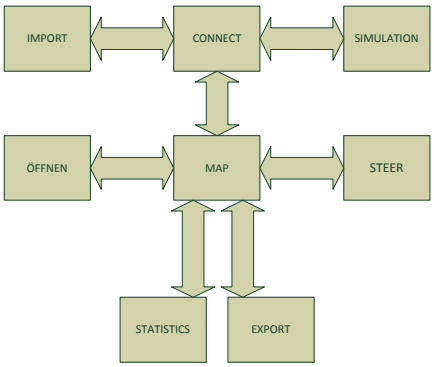

Abbildung 1: Menüführung