

# Validierung

## Super KIT Fighter

Praxis der Software-Entwicklung Gruppe 6

Philipp Schleier, Lucas Bechberger, Holger Caesar, Patrick  
Klinowski, Valerij Wittenbeck

14. Februar 2011

---

Referent: Prof. Dr.-Ing. Uwe D. Hanebeck

Betreuer: Dipl.-Inform. Henning Eberhardt  
Dipl.-Inform. Ferdinand Packi

---

SUPER



FIGHTER

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
<b>2</b>	<b>Umsetzung der Qualitätsanforderungen und Änderungen</b>	<b>2</b>
2.1	Intuitive Holodeck-Steuerung . . . . .	2
2.2	Einfache 3D-Modelle . . . . .	3
2.3	Balancing bezüglich der verschiedenen Eingabemethoden . . . . .	3
2.4	Ausgeglichene Highscoreberechnung . . . . .	3
2.5	Balancierte Waffen . . . . .	4
2.6	Zusammenfassung . . . . .	4
<b>3</b>	<b>Testszenarien</b>	<b>5</b>
3.1	Ändern von Einstellungen im Menü . . . . .	5
3.2	Holodeck vs. Tastatur/Maus ohne Sieger . . . . .	6
3.3	Holodeck vs. Tastatur/Maus mit Sieger . . . . .	7
3.4	Spiel starten & beitreten . . . . .	8
3.5	Vollbildspiel ausführen & Rematch . . . . .	9
<b>4</b>	<b>Überdeckungs-Tests</b>	<b>10</b>
4.1	Line Coverage . . . . .	11
4.1.1	Controller . . . . .	11
4.1.2	Model . . . . .	12
4.1.3	View . . . . .	12
4.2	Branch Coverage . . . . .	13
4.2.1	Controller . . . . .	13

4.2.2	Model . . . . .	13
4.2.3	View . . . . .	14
<b>5</b>	<b>Auslastungs- und Stresstests</b>	<b>15</b>
5.1	Verbundene Clients . . . . .	15
5.2	Anzahl konsekutiver Spiele . . . . .	15
5.3	Tastatur- und Mauseingaben . . . . .	16
5.4	Netzwerk . . . . .	17
<b>6</b>	<b>Fehler</b>	<b>18</b>
6.1	Abhängigkeit vom Tracking . . . . .	18
6.2	Anzeige im Head-Mounted-Display . . . . .	18
6.3	Start-Verzögerung . . . . .	18
6.4	Kodierung der Tastatur-Belegung . . . . .	19
<b>7</b>	<b>Glossar</b>	<b>20</b>
<b>A</b>	<b>Verwendete Testsysteme</b>	<b>21</b>
A.1	Targa Visionary 2000+ . . . . .	21
A.2	MacBook Pro 13" 2.26 GHz . . . . .	21
A.3	MacBook Pro 13" 2.53 GHz . . . . .	22
A.4	AMD Desktop PC . . . . .	22
A.5	Intel Desktop PC . . . . .	22
A.6	Medion MD8080XL . . . . .	22
A.7	Toshiba Satellite A210-103 . . . . .	23

# Einführung

Die hier zusammengefasste Testphase (auch “Validierung” oder “Qualitätssicherung”) schließt sich direkt an die *Implementierungsphase* an und dient der Überprüfung der zuvor programmierten Software.

Überprüft wird hierbei zum einen die Funktionalität des Programms, also die korrekte Implementierung der zuvor im Pflichtenheft festgelegten funktionellen Anforderungen.

Zum anderen wird aber auch in sogenannten *Stresstests* erprobt, wie sich das Programm unter extremen Bedingungen verhält, um zu sehen, ob ein normaler Betrieb auch bei ungewöhnlicher Belastung möglich ist. Ist dies der Fall, so ist auch die stabile Ausführung im normalen Gebrauch gesichert.

Ein weiterer Aspekt besteht in der Überprüfung des eigentlichen Codes hinsichtlich seiner Quantität, im Gegensatz zur den bisher erläuterten reinen qualitativen Merkmalen. Typischerweise wird hierbei mit gewisser Drittsoftware getestet, wie viel des geschriebenen Codes tatsächlich ausgeführt wird und wie viel unnötiger oder auch einfach normalerweise unbenutzter Code produziert wurde.

Selbstverständlich ist das Ziel dieser Phase der Entwicklung eines Softwareprojekts, dass die ausgeführten Tests nur positive Ergebnisse hervorbringen. Dafür ist es aber auch nötig, eventuell angetroffene Fehler im Programm zu beheben bzw. überflüssigen Code zu entfernen. Dies gehört ebenfalls zu den Aufgaben während dieser Phase.

Am Ende steht demnach ein im Rahmen der Spezifikationen möglichst voll funktionsfähiges Produkt. Nicht behobene oder nicht behebbare Fehler im Programm, die möglicherweise auf andere eingesetzte Komponenten zurückzuführen sind, zählen ebenfalls zum Inhalt dieses Dokuments.

# Umsetzung der Qualitätsanforderungen und Änderungen

Im Folgenden soll überprüft werden, inwieweit die Qualitätsanforderungen, die im Pflichtenheft zur Sicherstellung einer hohen Softwarequalität formuliert wurden, im fertigen Produkt umgesetzt werden konnten und inwiefern von diesen abgewichen wurde.

Es wird zunächst die jeweilige Anforderung aus dem Pflichtenheft zitiert und dann auf die Umsetzung eingegangen.

## 2.1 Intuitive Holodeck-Steuerung

*“Die Steuerung mithilfe des Holodecks sollte möglichst intuitiv gestaltet werden, um einfaches und problemloses Spielen zu ermöglichen und keine künstliche Hemmschwelle aufzubauen.”*

Durch die direkte Abbildung der vom Kopftracking aufgezeichneten Bewegungen und Rotationen auf die Spielfigur ist eine intuitive Bewegung im virtuellen Raum gewährleistet. Da unser Spiel auch mit deaktivierter Convolution-Funktion spielbar ist, dürften auch Neulinge im Holodeck keine Probleme mit der Bewegung der Spielfigur haben, da dann die ungewohnte Pfadkompression wegfällt und Bewegungen und Rotationen 1:1 in die Spielumgebung übertragen werden.

Da es nur zwei verwendete Gesten gibt und beide Gesten in ihrer Ausführung sehr einfach sind (horizontaler Schlag nach vorne als Attacke und Hand vor den Kopf halten als Parade), dürften sich auch hier keine Probleme bezüglich der Steuerung des Spiels ergeben. Eine intuitive Steuerung ist also folglich gewährleistet.

## 2.2 Einfache 3D-Modelle

*“[Es] sollten keine allzu komplexen 3D-Modelle verwendet werden, um auch den Betrieb auf leistungsärmeren Rechnern zu ermöglichen.”*

Das 3D-Modell mit der höchsten Anzahl Polygone (Dreiecksflächen) ist die Spielfigur “Lego Man” mit 2565 Polygonen, gefolgt von der Arena “The Cage” mit 462 Polygonen. Bezüglich der Performance lässt sich sagen, dass pro Match ungefähr 5000 Polygone mit einer Frequenz von 60Hz gerendert werden, was weit unter dem heutigen Standard liegt. Jede OpenGL2-kompatible Grafikkarte ist in der Lage die Szenerie ohne Ruckeln darzustellen, keine Auslastung durch andere Prozesse vorausgesetzt.

## 2.3 Balancing bezüglich der verschiedenen Eingabemethoden

*“Ein gutes Balancing bezüglich der unterschiedlichen Steuerungsmöglichkeiten [...], um die Fairness innerhalb des Spiels zu wahren”*

Es wurde versucht, die Steuerungsvarianten möglichst gut gegeneinander zu gewichten, allerdings bewegen sich Menschen, die das Holodeck noch nicht oft benutzt haben oder gar zum ersten Mal benutzen naturgemäß viel langsamer als versierte Anwender. Aufgrund dessen ist ein zu 100% faires Balancing fast unmöglich. Durch die Implementierung einfacher Skalierungsfaktoren kann dies allerdings recht schnell angepasst werden, sollte es zu Komplikationen kommen.

## 2.4 Ausgeglichene Highscoreberechnung

*“Schließlich ist eine möglichst ausgeglichene Punktevergabe wünschenswert, um sicherzustellen, dass die erreichten Punktzahlen in der Highscoreliste vergleichbar sind.”*

Der erreichte Highscore berechnet sich wie folgt:

$$100 \cdot \left(\frac{Rest}{Max}\right) + 100 \cdot \left(1 - \left(\frac{GegnerRest}{GegnerMax}\right)\right) + 100 \cdot \left(\frac{Restzeit}{Rundenzeit}\right)$$

Dabei bezeichnen Rest und Max bzw. GegnerRest und GegnerMax jeweils die Rest-Lebensenergie und Maximale Lebensenergie der eigenen Spielfigur bzw. des Gegners.

Da die maximale Lebensenergie aller Beteiligten mit der Dauer des Matches mitskaliert und in unserer Berechnungsformel sowohl die eigene Rest-Lebensenergie als auch die gegnerische Rest-Lebensenergie und auch die verbleibende Zeit mit eingehen, ist eine ausgeglichene Punktevergabe gewährleistet, bei der auch Highscores, die bei verschiedenen Rundenlängen erzielt wurden, vergleichbar bleiben.

## 2.5 Balancierte Waffen

*“Für den Fall, dass noch optionale Funktionen, wie z.B. verschiedene Waffen, implementiert werden, ist auch hier auf ein gutes Balancing zu achten, z.B. bezüglich Schlaggeschwindigkeit und verursachtem Schaden.”*

In unserem Spiel gibt es vier verschiedene Waffen. Diese unterscheiden sich in den verursachten Schadenspunkten sowie in der Reichweite. Somit muss man bei der Wahl der Waffe abwägen, ob man lieber mehr Schaden pro Treffer verursachen möchte oder lieber eine größere Reichweite beim Schlag hat. Hinsichtlich der Schlaggeschwindigkeit ergeben sich keine Unterschiede, da diese nicht von der verwendeten Waffe sondern von der verwendeten Spielfigur abhängig ist.

## 2.6 Zusammenfassung

Insgesamt lässt sich erkennen, dass die im Pflichtenheft formulierten Qualitätsanforderungen zum Großteil erfüllt werden konnten. Zwar gibt es in einigen Punkten noch Verbesserungsbedarf, in der Summe aber dürfte die gewünschte Qualität unseres Softwareprodukts gewährleistet sein.

## KAPITEL 3

# Testszzenarien

Testszzenarien dienen der Sicherstellung der Funktionalität eines Softwareprodukts gemäß den Spezifikationen in einem zuvor angefertigten Dokument.

Hier wird also überprüft, ob die im Pflichtenheft festgelegten Produkthanforderungen entsprechend umgesetzt wurden und ein fehlerfreies Benutzen der Software möglich ist.

### 3.1 Ändern von Einstellungen im Menü

Dieses Szenario dient dazu, zu überprüfen, ob im Menü vorgenommene Änderungen an den gespeicherten Einstellungen nach einem Neustart des Programms ordnungsgemäß übernommen werden.

Testfall	Getestete Funktion	Bestanden
/T000/	Starten des Programms	✓
/T010/	Navigation im Menü (alle Möglichkeiten durchprobieren)	✓
/T020/	Änderung der Grafikeinstellungen	✓
/T030/	Konfigurieren der Tastatursteuerung	✓
/T035/	Soundeinstellungen ändern	✓
/T190/	Rücksetzen der Highscore-Liste	✓
/T200/	Beenden des Programms	✓
/T000/	Starten des Programms	✓
/T020/	Überprüfen der Änderungen an Grafikeinstellungen	✓
/T030/	Überprüfen der Konfiguration der Tastatursteuerung	✓
/T200/	Beenden des Programms	✓

### 3.2 Holodeck vs. Tastatur/Maus ohne Sieger

In diesem Testszenario wird das Holodeck mit Kopf- und Handtracking eingesetzt und im Spiel gegen einen Gegner mit Tastatur/Maus-Steuerung getestet. Dabei wird gleichzeitig das Verhalten des Spiels bei Erzielen eines Unentschieden getestet.

Testfall	Getestete Funktion	Bestanden
/T000/	Starten des Programms	✓
/T040/	Starten eines 1-gegen-1 Spiels	✓
/T050/	Korrekte und flüssige Anzeige des Spiels mit HMD	✓
/T060/	Korrekte und flüssige Anzeige des Spiels auf PC-Bildschirm	✓
/T080/	Korrekte Reaktion des Programms auf Eingabesignal via Holodeck-Tracking (Helm und 1 Handschuh)	✓
/T090/	Korrekte Reaktion des Programms auf Eingabesignal via Maus/Tastatur	✓
/T100/	Kollision eines Avatar mit der Spielfeldbegrenzung	✓
/T110/	Kollision zweier Avatare miteinander	✓
/T120/	Kollision zweier Waffen	✓
	Ausgabe von Soundeffekten	✓
/T140/	Matchende durch Ablauf der Zeit	✓
/T200/	Beenden des Programms	✓

### 3.3 Holodeck vs. Tastatur/Maus mit Sieger

In diesem Testszenario wird das Holodeck mit **Computerunterstützung** (hier AI genannt) eingesetzt. Es wird also kein Hand-Tracking verwendet, sondern die AI. Diese entscheidet, ob der Gegner angegriffen wird, wenn er in der Nähe ist, oder ob ein potentieller Angriff abgewehrt wird.

Der Gegner verbindet sich über das Netzwerk und benutzt zur Steuerung Tastatur und Maus. Dabei wird gleichzeitig das Verhalten des Spiels für den Fall, dass ein Spieler gewinnt, getestet.

Testfall	Getestete Funktion	Bestanden
/T000/	Starten des Programms	✓
/T040/	Starten eines 1-gegen-1 Spiels	✓
/T050/	Korrekte und flüssige Anzeige des Spiels mit HMD	✓
/T060/	Korrekte und flüssige Anzeige des Spiels auf PC-Bildschirm	✓
/T070/	Korrekte Reaktion des Programms auf Eingabesignal via Holodeck-Tracking (nur Helm)	✓
/T090/	Korrekte Reaktion des Programms auf Eingabesignal via Maus/Tastatur	✓
/T100/	Kollision eines Avatar mit der Spielfeldbegrenzung	✓
/T110/	Kollision zweier Avatare miteinander	✓
/T120/	Kollision zweier Waffen	✓
	Ausgabe von Soundeffekten	✓
/T130/	Kollision von Waffe und Avatar & Schadensberechnung	✓
/T150/	Matchende durch Sieg, korrekte Highscore-Berechnung & Anzeigen des After-Match-Bildschirms	✓
/T170/	Eintragen in die Highscore-Liste	✓
	Überprüfen des Highscore-Eintrags	✓
/T200/	Beenden des Programms	✓

### 3.4 Spiel starten & beitreten

Getestet wird das Starten eines Servers auf dem ersten Client und das Herstellen einer Verbindung zu diesem Server von einem zweiten Client.

Testfall	Getestete Funktion	Bestanden
/T000/	Starten des Programms	✓
	Starten eines Spiels über den Menüpunkt "Start Game"	✓
	Korrekte Verarbeitung der Einstellungen für die Skins (Arena, Waffe, Spieler), Rundendauer und Spielernamen	✓
	Beitreten des Spiels durch einen zweiten Client über den Menüpunkt "Join Game"	✓
	Aufbauen einer Netzwerkverbindung zum gestarteten Spiel	✓
	Korrekte Verarbeitung der Einstellungen für den Waffen- und Spieler-Skin und des Spielernamen	✓
/T050/	Korrekte und flüssige Anzeige des Spiels mit HMD	✓
/T060/	Korrekte und flüssige Anzeige des Spiels auf PC-Bildschirm	✓
/T070/	Korrekte Reaktion des Programms auf Eingabesignal via Holodeck-Tracking (nur Helm)	✓
/T090/	Korrekte Reaktion des Programms auf Eingabesignal via Maus/Tastatur	✓
/T100/	Kollision eines Avatar mit der Spielfeldbegrenzung	✓
/T110/	Kollision zweier Avatare miteinander	✓
/T120/	Kollision zweier Waffen	✓
	Ausgabe von Soundeffekten	✓
/T130/	Kollision von Waffe und Avatar & Schadensberechnung	✓
/T150/	Matchende durch Sieg, korrekte Highscore-Berechnung & Anzeigen des After-Match-Bildschirms	✓
/T170/	Eintragen in die Highscore-Liste	✓
	Überprüfen des Highscore-Eintrags	✓
/T200/	Beenden des Programms	✓

### 3.5 Vollbildspiel ausführen & Rematch

Dieses Testszenario wurde zu Gunsten der Übersichtlichkeit leicht verkürzt. Dabei wurde auf die Erwähnung bereits getesteter Fälle, wie die Kollisionserkennung, korrekte Darstellung oder korrekte Verarbeitung der Einstellungen beim Starten eines bzw. Beitreten zu einem Spiel, verzichtet. Diese wurden natürlich dennoch getestet, die Ergebnisse sind den vorhergehenden Testszenarien zu entnehmen.

Testfall	Getestete Funktion	Bestanden
/T000/	Starten des Programms	✓
/T020/	Änderung der Grafikeinstellungen (Vollbild mit entsprechender Auflösung)	✓
/T200/	Beenden des Programms	✓
/T000/	Starten des Programms	✓
	Starten eines Spiels über den Menüpunkt "Start Game"	✓
	Beitreten des Spiels durch einen zweiten Client über den Menüpunkt "Join Game"	✓
/T140/	Matchende durch Ablauf der Zeit	✓
/T180/	Starten eines Revanche-Matches	✓
/T150/	Matchende durch Sieg, korrekte Highscore-Berechnung & Anzeigen des After-Match-Bildschirms	✓
/T170/	Eintragen in die Highscore-Liste	✓
/T200/	Beenden des Programms	✓

*Das Programm hat somit alle Testszenarien erfolgreich absolviert.*

## Überdeckungs-Tests

Überdeckungstests (sogenannte “Coverage-Tests”) dienen dazu, festzustellen, wieviel Code eines Programms tatsächlich in den Testfällen ausgeführt wird. Es geht darum, herauszufinden, ob es Stellen im Code gibt, die von den Testfällen nicht erreicht werden. Darunter fallen natürlich auch solche Stellen im Code, die auf Erweiterbarkeit des Programms ausgelegt sind, aber aufgrund vorerst nur einfacher Implementierungen oder noch nicht vorhandener Erweiterungen für diesen Programmteil natürlich auch von Testfällen nicht erreicht werden. Die Werte von Coverage Tests geben demnach Auskunft darüber, wie viel Code letztlich im regulären Gebrauch einer Software ständig benutzt wird, aber auch darüber, wie viel möglicherweise unnötigen Code ein Programm enthält. Ein möglichst hoher Anteil an abgedecktem Code ist daher wünschenswert. Zur Überprüfung dieses Produkts wurde die freie Software *Cobertura v1.9.4.1* benutzt. Diese ermöglicht es recht einfach Line Coverage und Branch Coverage Tests durchzuführen.

78		<code>public void update(float tpf) {</code>
79	86691	<code>    hudNode.updateGeometricState();</code>
80		
81	86692	<code>        int duration = match.getRemainingTime();</code>
82	86689	<code>        int seconds = duration % 60;</code>
83	86689	<code>        int minutes = (duration - seconds) / 60;</code>
84	86691	<code>        timeLeft.setText(String.format("%02d:%02d", minutes, seconds));</code>
85	86689	<code>        timeLeft2.setText(String.format("%02d:%02d", minutes, seconds));</code>
86		
87	86691	<code>        updateHealthBars(tpf, healthBars, healthBarOverlays, camera, hudNode, false);</code>
88	86689	<code>        updateHealthBars(tpf, healthBars2, healthBarOverlays2, camera, hudNode, true);</code>
89	86691	<code>    }</code>
90		
91		<code>private void initHUD() {</code>
92		
93	5	<code>    ViewPort hudViewPort = rm.getPostView("HUD ViewPort");</code>
94	5	<code>    if( hudViewPort == null ) {</code>
95	4	<code>        ViewPort guiViewPort = rm.getPostView("Gui Default");</code>
96	4	<code>        hudViewPort = app.getRenderManager().createPostView("HUD ViewPort",</code>
97	4	<code>        hudViewPort.setClearEnabled(false);</code>
98	4	<code>        hudViewPort.getCamera().setLocation(new Vector3f(0, 0, -10));</code>
99		<code>    } else {</code>
100	1	<code>        List&lt;Spatial&gt; scenes = hudViewPort.getScenes();</code>
101	3	<code>        for ( Spatial scene : scenes ) {</code>
102	1	<code>            hudViewPort.detachScene(scene);</code>
103		<code>        }</code>
104		<code>    }</code>
105	4	<code>    hudNode = new Node("HUD Node");</code>
106	4	<code>    hudNode.setQueueBucket(Bucket.Gui);</code>
107	4	<code>    hudNode.setCullHint(CullHint.Never);</code>
108		

Abbildung: Detaillierte Ergebnis-Ansicht eines Coverage-Tests

## 4.1 Line Coverage

Dies ist die simpelste Form des Überdeckungstests: Sie misst lediglich wieviel Zeilen Code des Programms tatsächlich ausgeführt werden. Das Ergebnis besteht dann entsprechend aus der Anzahl der ausgeführten Zeilen, sowie dem prozentualen Anteil der ausgeführten Zeilen gemessen an der jeweiligen Klasse oder am jeweiligen Paket.

Die grünen Bereiche im Diagramm stehen jeweils für den abgedeckten Code, die roten für den nicht abgedeckten.

### 4.1.1 Controller

Der Controller setzt sich aus vier Paketen zusammen, welche im Durchschnitt eine Testabdeckung von 86,93% erreichen.



### 4.1.2 Model

Das Model, also die Datenhaltung und Logik der Software, hat eine durchschnittliche Testüberdeckung von 87,06%.



### 4.1.3 View

Die View besteht aus den drei Anzeigemodi: Spieler im HMD, Spieler an einem normalen Bildschirm und die Zuschauer-Ansicht.

Mit ca. 93% Abdeckung ist die View der am besten durch Testfälle abgedeckte Teil des Spiels, was darauf zurückzuführen ist, dass man natürlich ohne Anzeige schlecht spielen kann.



## 4.2 Branch Coverage

Bei der Branch Coverage wird gemessen, wieviele der möglichen Pfade im Programm während der ausgeführten Tests durchlaufen wurden.

Auch hier gilt wie bei der Line Coverage, dass manche Pfade bei auf Erweiterbarkeit ausgelegter Software noch nicht durchlaufen werden, solange eine Schnittstelle für Erweiterungen noch nicht benutzt wird, da diese Erweiterungen nicht oder noch nicht zur Verfügung stehen. In manchen Fällen etwa wird die Software lediglich erweiterbar gestaltet, um sich die Möglichkeit offen zu halten, später leichter Erweiterungen zu schreiben, nicht etwa weil das konkrete Vorhaben besteht, Erweiterungen für diese Schnittstelle zu programmieren.

### 4.2.1 Controller



### 4.2.2 Model



### 4.2.3 View



## KAPITEL 5

# Auslastungs- und Stresstests

Die folgenden Tests dienen dazu, festzustellen wie sich das Produkt im Einsatz unter extremen Nutzungsbedingungen verhält.

Es geht darum festzustellen, ob die Software in diesen extremen Fällen stabil weiterläuft, oder aufgrund von Überlastung versagt. Im Übrigen werden bei den hier durchgeführten Tests die jeweils geprüften Faktoren größer gewählt als im normalen Einsatz zu erwarten.

### 5.1 Verbundene Clients

Es soll hier getestet werden, ob der Server eine große Anzahl an Clients (Zuschauer) mit Daten beliefern kann, ohne dass die Spielperformance beeinträchtigt wird.

#### Ergebnis

Getestet wurde die Verbindung von insgesamt 16 Clients zum Server, darunter zwei Spieler und 14 Zuschauer. Dabei liefen alle Clients, vor allem aber die der Spieler, problemlos und flüssig. Auch am Server waren keinerlei Auswirkungen sichtbar. ✓

### 5.2 Anzahl konsekutiver Spiele

Sinn dieses Tests ist es, herauszufinden, ob die Performance des Produkts bei vielen aufeinanderfolgenden Spieldurchläufen leidet oder nahezu endloser Spielspaß gewährleistet ist.

Probleme könnten dabei entstehen, wenn im Programm Speicher nicht richtig freigegeben wird und somit die Hardware des zugrundeliegenden Geräts überlastet wird.

Der hier getestete Fall entspricht dem nichtfunktionalen Testfall */NFT000/ Langzeittest am Spiel*.

## Ergebnis

Getestet wurde das Spiel mit knapp über 50 aufeinanderfolgenden Spieldurchläufen. Es konnten keine Unterschiede in der Hardwarebelastung zwischen dem ersten und letzten durchgeführten Spiel festgestellt werden. ✓

## 5.3 Tastatur- und Mauseingaben

Hier wird getestet, ob das Programm wie erwartet reagiert, wenn man beispielsweise mehrere Tasten gleichzeitig drückt oder die Maus sehr schnell bewegt. Ziel ist es herauszufinden, ob möglicherweise die vielen Eingabedaten zu einem Fehler in der Verarbeitung führen, oder diese korrekt umgesetzt werden und trotzdem flüssige Bewegungen entstehen.

Getestet wurden dabei mehrere mögliche Fälle, deren Ergebnisse im folgenden kurz umrissen sind.

### Ergebnisse

#### *Entgegengesetzte Bewegungen:*

Beim gleichzeitigen Drücken der Tasten für eine Vortwärts- und Rückwärtsbewegung bzw. Links- und Rechtsbewegung bewegt sich die Spielfigur nicht von der Stelle. ✓

#### *Diagonale Bewegungen:*

Beim gleichzeitigen Drücken zweier Richtungstasten, die nicht entgegengesetzte Richtungen steuern, verhält sich die Spielfigur wie erwartet und bewegt sich diagonal. ✓

#### *Schlagen und Blocken:*

Versucht man gleichzeitig zu schlagen und zu blocken, so verhält sich das Spiel korrekt und es wird nur geblockt. ✓

#### *Umschauen:*

Da ein gleichzeitiges Bewegen der Maus in entgegengesetzte Richtungen nicht möglich ist, wurde dieser Test mit reiner Tastatursteuerung durchgeführt. Es wurden dabei die Tasten für das Umschauen in entgegengesetzte Richtungen gleichzeitig gedrückt. Das erwartete Verhalten, nämlich dass sich die Blickrichtung nicht ändert, trat ein. ✓

#### *Mehrfachbelegung einer Taste:*

Beim Belegen einer Taste mit allen Befehlen führte die Spielfigur gelegentlich einen Schlag aus, ansonsten gab es keinerlei Reaktion. ✓

## 5.4 Netzwerk

Hier soll die Reaktion des Spiels bei verschiedenen Problemen in der Netzwerkverbindung getestet werden. Da die Netzwerk-Kommunikation zwischen Server- und Client-Teil der Anwendung stattfindet, wird dabei auch speziell darauf geachtet, wie der jeweils relevante Teil des Spiels reagiert.

### Ergebnisse

#### *Starten mehrerer Server:*

Beim Versuch mehrere Server-Teile des Programms auf einem PC zu starten tritt kein Ausnahmefall auf. Wie erwartet wurde einen Server gestartet, bei den anderen wurde der Startvorgang abgebrochen, weil die benötigten Ports bereits belegt waren. ✓

#### *Beenden eines Clients:*

Beendet man eine Client-Anwendung während des Spiels, so wird das Spiel abgebrochen und der andere Spieler erhält eine entsprechende Nachricht. ✓

#### *Beenden eines MC-Server:*

Wird ein Motion-Compression-Server beendet oder die Verbindung aus anderen Gründen unterbrochen, stürzt der Serverteil des Spiels ab, die Clients frieren ein. ✗

#### *Beenden des Servers:*

Beendet man den Server-Teil während eines Spiels, zeigen die Clients eine entsprechende Meldung und man kann zum Hauptmenü zurückkehren. ✓



Abbildung: Bestandene Tests (grün), nicht bestandene Tests (rot)

# Fehler

### 6.1 Abhängigkeit vom Tracking

Leider fiel während der Tests des Öffterns das Tracking im Holodeck aus. Da das Programm zur korrekten Funktion natürlich auf diese Daten angewiesen ist, führen solche Ausfälle zu willkürlichem Verhalten des Programms.

Dieser Fehler ließ sich leider nicht beheben, da wir keine Möglichkeit finden konnten, die Daten des MC-Server auf Korrektheit zu prüfen.

### 6.2 Anzeige im Head-Mounted-Display

Beim Betrieb im Head-Mounted-Display des Holodecks wurden keine Lebensbalken und fehlerhafte Menüs angezeigt. Startet man das Programm auf einem beliebigen PC im Holodeck-Modus, so treten keinerlei Probleme auf, die Darstellung des Menüs funktioniert korrekt und auch alle Lebensbalken werden angezeigt. Es hat den Anschein als wäre Java nicht in der Lage mehr als ein Bild auf einmal zu laden und würde den Bild-Cache immer mit einem einzigen Bild überschreiben.

Wir konnten für dieses Problem leider keine Lösung finden, es scheint aber in irgendeiner Weise auf die verwendete Java-Version oder Hardware des Holodeck-Rechners zurückzuführen zu sein. Die Anzeige im HMD-Modus funktioniert also überall außer auf dem HMD.

### 6.3 Start-Verzögerung

Das Spiel läuft schon einige Sekunden, bevor die Clients anfangen zu rendern. Dadurch ist das Spiel am Ende minimal kürzer als zuvor eingestellt, allerdings entseht kein Nachteil, da alle Clients diese Verzögerung erleiden.

## 6.4 Kodierung der Tastatur-Belegung

Die Tastatur-Belegung lässt sich zwar im Menü ändern, derzeit aber nur, wenn man die Codes der jMonkey-Engine für die jeweiligen Tasten kennt. Dies ist sehr unkomfortabel, leider gibt es aber keine Umwandlungsmöglichkeit seitens der verwendeten Engine.

## KAPITEL 7

# Glossar

Begriff	Erklärung
After-Match-Bildschirm	Dialog nach Beenden einer Spielrunde, welcher das Starten einer neuen Spielrunde ermöglicht
AI	Kurzform für Artificial Intelligence (dt. künstliche Intelligenz), hier: Computerunterstützung bei fehlendem Eingabegerät
Branch Coverage Test	Messung der beim Programmablauf abgearbeiteten Ausführungspfade im Programm
Holodeck	Das Holodeck ermöglicht es dem Benutzer mit Hilfe eines HMD, dessen Position mit akustischem Tracking erfasst wird, sich telepräsent an einem anderen (in unserem Fall virtuellen) Ort zu befinden. Optional können zusätzlich Geräte wie der Datenhandschuh oder die Haptik verwendet werden, um den Interaktionsgrad zu erhöhen. Eine Besonderheit ist auch die Verwendung von Motion Compression, um den zur Verfügung stehenden Platz im Holodeck besser auszunutzen
HMD	Kurz für: Head-Mounted-Display, ein auf dem Kopf getragenes visuelles Ausgabegerät, das die computergenerierten Bilder auf zwei Displays (eines für jedes Auge) in relativ geringem Abstand zu den Augen darstellt
Line Coverage Test	Messung der beim Programmablauf ausgeführten Zeilen Code
Model-View-Controller	Trennung von Spiellogik, Anzeige und Eingabe

## Verwendete Testsysteme

### A.1 Targa Visionary 2000+

- Prozessor: AMD Athlon XP, 1666 MHz
- Hauptspeicher: 256 MB DDR SDRAM
- Grafik: NVIDIA GeForce4 MX 440 (64 MB)
- Betriebssystem: Microsoft Windows XP Home Edition
- Java Runtime: JRE 1.6

*Wurde zum Testen der Mindestvoraussetzungen genutzt. Spiel startete nicht, da OpenGL 2 nicht unterstützt.*

### A.2 MacBook Pro 13" 2.26 GHz

- Prozessor: Intel Core 2 Duo, 2.26 GHz
- Hauptspeicher: 2 GB DDR3 RAM @ 1067 MHz
- Grafik: NVIDIA GeForce 9400M (256 MB)
- Betriebssystem: Apple Mac OS X 10.6.6
- Java Runtime: Apple Distributed JRE 1.6

### **A.3 MacBook Pro 13" 2.53 GHz**

- Prozessor: Intel Core 2 Duo, 2.53 GHz
- Hauptspeicher: 4 GB DDR3 RAM @ 1067 MHz
- Grafik: NVIDIA GeForce 9400M (256 MB)
- Betriebssystem: Apple Mac OS X 10.6.6
- Java Runtime: Apple Distributed JRE 1.6

### **A.4 AMD Desktop PC**

- Prozessor: AMD Phenom II X4 955, 3.20 GHz
- Hauptspeicher: 4 GB DDR3 RAM @ 1333 MHz
- Grafik: ATI Radeon HD 5770 (1024 MB)
- Betriebssystem: Microsoft Windows 7 Professional
- Java Runtime: JRE 1.6

### **A.5 Intel Desktop PC**

- Prozessor: Intel Core 2 Quad Q6600, 2.40 GHz
- Hauptspeicher: 4 GB DDR2 RAM @ 1066 MHz
- Grafik: ATI Radeon HD 5770 (1024 MB)
- Betriebssystem: Microsoft Windows 7 Professional
- Java Runtime: JRE 1.6

### **A.6 Medion MD8080XL**

- Prozessor: Intel Pentium D 830, 3.0 GHz
- Hauptspeicher: 2 GB DDR2-533 SDRAM
- Grafik: ATI Radeon X1900 GT (256 MB)
- Betriebssystem: Microsoft Windows 7 Professional
- Java Runtime: JRE 1.6

## A.7 Toshiba Satellite A210-103

- Prozessor: AMD Turion 64 X2 TL-58, 1.90 GHz
- Hauptspeicher: 2 GB DDR2 RAM @ 667 MHz
- Grafik: ATI Mobility Radeon HD 2600 (256 MB)
- Betriebssystem: Microsoft Windows 7 Professional, Ubuntu Linux 10.10
- Java Runtime: JRE 1.6