Georg-Simon-Ohm Hochschule Nürnberg Fakultät Elektrotechnik
Feinwerktechnik Informationstechnik efi

Projekt

Prüfungsstudienarbeit von
Michael Kao
2108186
B-ME 6

Prüfungsstudienarbeit für das Projekt ReMasterBlaster im Rahmen des Studienprojektes BME-6

Sommersemester 2012

Bestätigung gemäß § 35 (7) RaPO

Michael Kao

Ich bestätige, dass ich die Prüfungsstudienarbeit mit dem Titel:

Prüfungsstudienarbeit für das Projekt 3Dreh Scanner
im Rahmen des Studienprojektes BME-5

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum:

02.08.2012

Unterschrift:

Abstract

Gemäß Paragraph 21, Absatz Eins der Rahmenprüfungsordnung, kurz RaPo, sind Prüfungsstudienarbeiten Prüfungsleistungen mit überwiegend zeichnerischem, gestalterischem oder sonstigem komplexen Inhalt und offenem Lösungsweg zum Nachweis kreativer Fähigkeiten, die sich wegen der umfassenden Aufgabenstellung und der Art der Aufführung in der Regel über einen längeren Zeitraum erstrecken.

Die einzelnen Prüfungsstudienarbeiten des Projektteams, bestehend aus Sebastian Adam, Sergej Bjakow, Michael Kao, Pavlina Pavlova und Maximilian Seyfert, welches sich mit der Realisierung eines Remakes des AMIGA Klassikers MasterBlaster mit neuen Webtechnologien (HTML5, CSS, JavaScript) befasst hat, sollen einen individuellen Einblick auf die jeweiligen Anforderung, die es mit Lösungsstrategien zu bewerkstelligen galt, gewährleisten.

Diese, meine eigene Studienprüfungsarbeit, will Auskunft über die Bedeutung der Rolle meiner Person in der gemeinsamen Teamarbeit verdeutlichen.

Projektroadmap

Tätigkeit	Dokument	Beteiligte
Erstellung des Lastenhefts	02_documents/02_requirements/ Lastenheft1_1.pdf	Sebastian Adam
Erstellung des Pflichtenhefts	02_documents/02_requirements/ Pflichtenheft_v1_1.pdf	Sebastian Adam
Erstellung des Zeitplans	02_documents/03_timeline/ Remasterblaster.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_01.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_02.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_03.pdf	Sergej Bjakow
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_04.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_05.pdf	Maximilian Seyfert
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_06.pdf	Michael Kao
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_07.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_08.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_09.pdf	Maximilian Seyfert
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_10.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_11.pdf	Sergej Bjakow
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_12.pdf	Sebastian Adam
Erstellung des Protokolls	02_documents/01_protocols/ ...Protokoll_13.pdf	Sebastian Adam
Erstellung der Testbench mit der Sprite – Engine	01_development/01_testbench/*	Pavlina Pavlova
Erstellung der ersten Testumgebung	01_development/02_testenvironment/*	Michael Kao Pavlina Pavlova
Programmierung der Spielelogik	01_development/03_final/game.js	Michael Kao
Erstellung der Sprite Zuordnung	01_development/03_final/sprites.js	Michael Kao

Programmierung des Graphical User Interface	01_development/03_final/gui.js	Pavlina Pavlova
Erstellung der Sprites – Quelldatei	04_material/01_rawfiles/ 01_sprites_psd/*	Michael Kao Sergej Bjakow
Modifikation der Sounds zu mp3 Dateien	01_development/03_final/sounds/*	Sergej Bjakow Maximilian Seyfert
Capturing der einzelnen Elemente (Sprites)	01_development/03_final/img/*	Sergej Bjakow
Zusammenfügen der Sprites	04_material/01_rawfiles/ 01_sprites_ psd/*	Michael Kao
Vorbereiten der einzelnen Szenenelemente	01_development/03_final/img/*	Pavlina Pavlova Sergej Bjakow Michael Kao
Erstellung der Zwischenpräsentation	03_presentations/ 2012-05-09_ReMasterBlaster.pdf	Sebastian Adam
Erstellung der Abschlusspräsentation	03_presentations/ 2012-07-20_ReMasterBlaster.pdf	Sebastian Adam Michael Kao Pavlina Pavlova
Erstellung eines Plakats für die Abschluss-präsentation	04_material/04_poster/ extras_plakat.pdf	Sebastian Adam
Erstellung eines Plakats für die Abschluss-präsentationen	04_material/04_poster/ project_presentation.pdf	Michael Kao
Erstellung des Wikis	http://schorsch.efi.fh-nuernberg.de/mewiki/index.php/Project-BME-2012-07-RemasterBlaster/Project-BME-2012-07-RemasterBlaster	Sebastian Adam
Erstellung der Gesamtdokumentation	02_documents/04_documentation/ 02_overalldocumentation/ Gesamtdokumentation.pdf	Sebastian Adam

Inhaltsverzeichnis

1. Projektidee

Die Idee für das Projekt stammt ursprünglich von Prof. Dr. Hopf. Das Thema wurde auf Anfrage von Sebastian Adam nach einem Projekt im HTML5 sowie JavaScript-Bereich gestellt. In der Intension für das Masterblaster Remake waren vor allem folgende Punkte ausschlaggebend:

- Reverse-Engineering
- Untersuchung der existierenden Sprite Engines
- Implementieren des Spiels als HTML5 Applikation in JavaScript

In einem einführendem Dokument von Prof. Dr. Hopf wurden unter anderem diese Punkte, sowie weitere instruierende Angaben, welche für einen gelungen Projektstart wichtig waren, zur Verfügung gestellt.

2. Projekt Team und Rollen

Das Projektteam formte sich in Zuge der Bekanntgabe des Themas bei der Besprechung der Projektthemen mit den Professoren Anfang des Sommersemesters 2012. Für mich war vor allem für die Wahl dieser Projektgruppe ausschlaggebend, mich ein ganzes Semester mit den Webtechnologien HTML5 und JavaScript zu beschäftigen. Mir war klar, eine tragende Rolle im Entwicklungsbereich der Gruppe anzustreben, da mir die entwickelnde Tätigkeit bereits in der Projektarbeit im vorigen Semester sehr zusagte. Da ich im Wintersemesterprojekt die Rolle des Projektleiters inne hatte, musste ich mich neben den Programmierarbeiten viel um organisatorisch Angelegenheiten kümmern und weniger um die technische Entwicklung unserer Software. Diese Rolle hat mir viel Freude bereitet, war mir aber auch an einigen Stellen zu untechnisch. Nach ersten Gesprächen im Team übernahm Sebastian Adam die Rolle des Projektleiters.

3. Projektstart

Der Startschuss fiel mit der Besprechung mit den Professoren am 26. März 2012. Hier wurden diverse Themen vorgestellt, sowie Vorschläge der Studenten vorgetragen. Eine Kennenlernphase war kaum von Nöten, da sich fast alle Beteiligten untereinander als Kommilitonen kannten, außer Pavlina, welche zwei Semester unter dem Rest der Gruppe studierte und ihre Projektarbeit vorzog. Die zwei Semester Unterschied zum Rest der Gruppe machten sich im Verlauf des Projekts zu keiner Zeit bemerkbar.

4. Einarbeitung in JavaScript

Während sich Teile des Teams mit der Auswahl der Sprite Engine beschäftigten, begann ich mich früh intensiv mit JavaScript zu beschäftigen, da in jedem Fall JavaScript die Sprache, der zu wählenden Sprite Engine sein würde. Durch das Studium hatten wir noch keine Berührung mit der wichtigsten Skriptsprache des Web. Parallel zum Projekt fand die Lehrveranstaltung Multimediaapplikationen statt, welche zur Hälfte von Prof. Dr. Hopf gehalten wurde und einen Grundstein für das Arbeiten mit clientseitigen Webtechnologien legte. Der Web-Teil der Veranstaltung vermittelte ein breites Wissen über die gängigen Auszeichnungs- sowie Interpretersprachen, allen voran JavaScript. Jedoch war mir bewusst, dass für einen Projekterfolgt das frühzeitige Einarbeiten und Vorweggreifen der Lehrveranstaltungsinhalte unabdingbar waren. Neben Skripten und Informationen aus dem Internet legte ich mir das viel renommierte Buch: JavaScript, The Definitive Guide, von David Flangan zu. Dieses Buch half mir besonders ein Gefühl für die Feinheiten der Sprache zu erlangen, als auch Anfängerfehler zu vermeiden, welche nach Aussage von Flanagan viele Einsteiger machen, indem diese beispielsweise JavaScript wie C benutzen. Die Syntax von JavaScript ist der C-Syntax zwar sehr ähnlich, jedoch können zum Beispiel durch das optionale Semikolon unerwartete Fehler auftreten. Neben der Literatur, sowie die

Lehrveranstaltung Multimediaapplikationen, waren uns unsere Betreuer stets bei schwierigen Fragen bezüglich der Programmierung zur Seite gestanden.

5. Einstieg in Crafty

Nachdem ein Großteil der vorhandenen Engines durchgesichtet wurde, kristallisierte sich eine Sprite Engine nach und nach heraus. Hierbei handelte es sich um die Crafty Engine, eine reine JavaScript Bibliothek, welche mit einem Entity – Component System arbeitet. Ein Entity Component System ist eine Art der Implementierung von Spieleinhalten, sodass deren Funktionalität durch Komposition statt objektorientierter Vererbung aufgebaut werden kann.

Durch einen Leistungstest der Engine wurde festgestellt, wie viel Animationen diese simultan erledigen kann. Dabei stellte sich heraus, dass die Performance im Google Chrome deutlich besser war als in anderen Browsern. Dadurch stand der Browser fest, in dem wir das Spiel letztendlich lauffähig haben möchten und während der Programmierung testen.

Crafty ist wie erwähnt eine reine JavaScript Bibliothek. Was auch ausschlaggeben für diese Engine war, ist die gute und vollständige Dokumentation, welche auf alle Funktionen der Bibliothek kurz eingeht. Neben der Dokumentation bietet der Entwickler des Weiteren ein kurzes Einsteigertutorial an, welches an vielen Stellen durch Beispiel-Code komplementiert wird.

Neben den erwähnten Ressourcen der Informationsbeschaffung besitzt die Webseite der Crafty Engine auch ein Forum. In diesem Forum kann von Nutzern nach Rat gefragt werden. Jedoch ist das Forum noch nicht sehr groß, wodurch nur bedingt Lösungen für eigene Probleme gefunden werden konnten. An dieser Stelle sei auch erwähnt, dass die Entwicklung

des Spiels ohne Erstellung eines Threads im Crafty-Forum und somit ohne externe Wissensakquirierung durchgeführt wurde.

Als erstes arbeitete ich das Tutorial durch, welches schon erste Züge im Umgang mit der Engine erkennen ließ. Das Tutorial war gut und ausführlich beschrieben, es führte Schritt für Schritt durch die wichtigsten Elemente der Engine, welche für die Erstellung eines Spieles unabdingbar sind. Von den zentralen Einheiten im Spiel, den Entities, den Komponenten, welche den Entities Funktionalität liefern über das Laden von Sprites und deren Animation, dem Empfangen von Userevents, bis hin zur Verpackung der Spielinhalten in Szenen. Jedoch wirkte die Syntax noch sehr ungewohnt, als auch die Anwendung eines Entity Component Systems war sehr neu für mich.

6. Versionsverwaltung mit git

Auf Anraten der Professoren beschäftigte sich das Team mit git, einer freien Software zur Versionsverwaltung. Mit Versionierungssystemen hatte ich bis dato ausschließlich Kontakt mit Subversion. Das Arbeiten mit Verwaltungssystemen ist für die Softwareentwicklung ungemein wichtig, was ich durchaus im Projekt zu spüren bekam. Besonders wenn im Team gearbeitet wird, ist es wichtig eine Möglichkeit zu haben, die bearbeiteten Programmteile zusammen zu führen und bei möglichen Fehlern in den Versionen zurück zu gehen.

Im Unterschied zu traditionellen Versionskontrollsystemen hat git diverse Eigenschaften, welche anders gelöst sind. Um einen etwas geführteren Einstieg in git zu ermöglichen, entschieden wir uns dafür, ein öffentliches git Repository auf www.github.com anzulegen. Dadurch konnte auf eine Vielzahl von Anleitungen zurückgegriffen werden. Von der Erstellung des ersten Repositories bis hin zum „*commiten*“ und „*pushen*“ wurde auf github detailliert eingegangen. Diese Hilfestellungen waren sehr nützlich,

jedoch war eine zeitaufwendige Einarbeitung in git bis zum Erlangen einer gewissen Routine von Nöten.

7. Programmierung der Spielelogik

Im Folgenden werden meine Tätigkeiten während der Umsetzung der Spielelogik genauer erläutert. Da die Programmierarbeit nicht linear stattfand und sich einzelne Programmteile immer wieder änderten, gehe ich nun im Folgenden auf die Implementierung der ersten Spieleumgebung ein, sprich die Erstellung der Masterblaster Welt, sowie Spielern mit rudimentären Funktionalitäten, um dann im zweiten Teil dieses Abschnitts auf das Endresultat der Spielelogik detaillierter einzugehen.

7.1 Erstellung der ersten Spieleumgebung

Wie bereits erwähnt, lieferte das Crafty Tutorial eine gute Basis, um mit der Entwicklung mit Crafty beginnen zu können. Ähnlich wie im Original Masterblaster waren die Inhalte des Tutorials Spielfiguren, eine rechteckige Welt und Gegenstände, die auf der freien Fläche liegen. Wir entschieden uns dafür, als erstes einen lauffähigen Nachbau der Masterblaster Welt zu implementieren. Dies beinhaltete unter anderem die undurchdringbaren Wände *„wall"*, den grünen Untergrund, sowie die zufällig verteilten Blöcke *„brick"*. Um mit ersten Sprites arbeiten zu können, nahm ich einen Screenshot aus dem Emulator, und baute die *„wall"* und den *„brick"* in Photoshop nach. Da der Emulator die Spielefläche in einer Größe von 608 * 480 Pixeln darstelle, ergab sich eine Sprite Größe von 32 *32 Pixeln, welche fortan als das Maß für die quadratischen Sprites verwendet wurde. Dies erleichterte das Extrahieren der Sprites und das Weiterverarbeiten ohne skalieren. Um die Bilder einer Ressource zu laden, wurden alle Sprites, außer die Spielersprites, fortan in der Datei sprites.png aufgeführt. Beim Laden der Sprites durch Crafty ist es möglich die Größe anzugeben, in unserem Fall *„32"*. Um nun die

jeweiligen Sprites benutzten zu können, wurde ein Objekt übergeben, mit den Namen des jeweiligen Sprites, sowie relativer Position in der Spritesmap. Abbildung 1 zeigt wie die Sprites aus der Spritesmap mittels Crafty.sprite definiert wurden.

```
Crafty.sprite(32, "img/sprites.png", {
      wall: [0, 0],
      brick: [0, 1],
});
```

Abbildung 1: Laden der Sprites

Um die Sprites auf das Spielfeld zu bringen, musste Crafty als erstes mit der „*Crafty.init*" Funktion gestartet werden. Diese erhielt als Parameter die x- sowie y-Länge des Spielfeldes. Mit Crafty.scene wurden zwei Szenen definiert, eine Loading Szene, welche solange „*Loading*" anzeigt bis alle Spritemaps in den Cache geladen wurden, sowie die Hauptszene, in welcher die Spielewelt definiert wurde. Die Hauptszene startete die „*generateWorld*" Funktion, welche die Blöcke und Wände als Entities generierte. Die Entities sind die zentralen Elemente, welche wichtige Interaktionen ermöglichen. Diese bekommen jeweils als Attribut den Namen des jeweiligen Sprites, sowie x-, y- und z-Position. Die z-Position gibt hierbei den z-Bufferwert an. Die „*generateWorld*" Funktion hat zwei Schleifen, welche über die x- und y-Richtung des Spielfeldes laufen und an den jeweiligen Stellen die richtigen Entities setzt. Das Resultat ist in Abbildung 2 zu sehen.

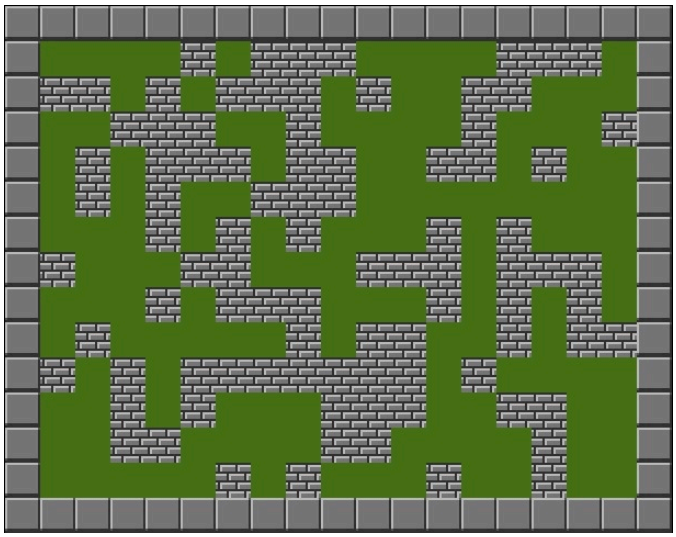

Abbildung 2: Erstellung der Welt

Die Erstellung einer spielbaren Figur erfolgt ähnlich der von Blöcken. Jedoch muss eine Spielerfigur um erheblich mehr Funktionalität erweitert werden. Zu einem muss die Figur steuerbar sein, sprich in unserem Fall auf Tastaturevents reagieren, als auch in Folge dieser Events animieren. Um zwei steuerbare Spieler zu implementieren, griff ich auf Teile des Bespielcodes aus dem Tutorial zurück und passte diesen auf unsere Gegebenheiten an. Zur Animation der Bewegung musste der Name der Animation definiert und die dazugehörigen Koordinaten der Animationsschritte aus der Spritesmap *„player_sprites.png*" angegeben werden. Damit die Spieler Entity auf die definierten Animationen zugreifen kann, sind diese in einer Komponente verpackt. Weiter gibt es eine Komponente, welche die Tastenevents definiert. Diese Komponenten erledigten im Prinzip immer die gleichen Aufgaben, waren jedoch durch kleine Unterschiede nicht für alle Spieler Entities anwendbar. Dadurch zeichnete sich bereits zu Beginn ein wachsender redundanter Code ab, was unter anderem zeigte, dass die anfängliche Implementierung noch nicht gut gelungen war.

7.2 Entwicklung der Spielelogik

Es kristallisierte sich heraus, dass die implementierte Steuerung der Spielfiguren wenig Flexibilität besaß. Ein anderer wichtiger Punkt war die Kollisionserkennung. Spieler dürfen beispielsweise nicht durch Blöcke laufen, aber durch andere Spieler. Die durch Crafty gelieferte Kollisionserkennung bot hier nur wenig Möglichkeit, auf spezielle Anforderungen wie wir sie hatten, genügend einzugehen.

Aus den genannten Gründen entschied ich mich dazu, weniger vorgefertigte Komponenten wie die Kollisionserkennung, oder die Spielersteuerung von Crafty zu verwenden, um stattdessen diese Komponenten selbst zu schreiben. Dadurch musste natürlich erheblich mehr Code geschrieben werden, was aber ermöglichte dem Verhalten des Originalspiels erheblich näher zu kommen. Fortan arbeitete ich an einer Lösung, welche es ermöglichte, dass Spielverhalten genauer zu definieren. Im Folgenden gehe ich auf die letztendlichen Ergebnisse dieser Arbeit ein.

7.2.1 Übergabe des Spielstandes

Das eigentliche Spiel wird aus der von Pavlina geschriebenen GUI geladen. Um dies zu ermöglichen packte ich den gesamten Code in eine Wrapperfunktion, welche durch die GUI aufgerufen werden kann. Diese neue Funktion mit dem Namen „*startGame*“ bekommt zwei Parameter:

Als ersten Parameter erhält diese das Objekt „*gameState*“, also den aktuellen Spielestand. Dies war von großer Bedeutung, da durch die Realisierung des Shops die Anforderung für eine flexible Aktualisierung der Spielereigenschaften gestiegen war. Der zweite Parameter ist eine Referenz zur aufrufenden Umgebung, womit es möglich ist, aus dem Spiel eine Funktion der GUI aufzurufen. In unserem Fall, um zu signalisieren, dass die Runde zu Ende ist. Um diese Schnittstellen von GUI und Spiel zu gestalten, arbeitete ich eng mit Pavlina zusammen.

7.2.2 Übergabe der Parameter an die Spieler

Damit der Spieler mit Leben befüllt werden kann, entwickelte ich nach und nach die Konstruktorfunktion „*Gamelogic*" aus. Mit der Erzeugung eines neuen Objektes als Komponente in Crafty werden fortan alle wichtigen Events empfangen bzw. Events ausgelöst, welche den jeweiligen Spieler betreffen. Das Gute an dieser Herangehensweise ist die Möglichkeit, die „*Gamelogic*" als Konstruktorfunktion in JavaScript zu verpacken. Dadurch kann für jeden Spieler ein neues Objekt dieser „*Gamelogic*" angelegt werden, worauf nur dieser Zugriff hat. Der Weg zu dieser eleganten Lösung wurde erst in den letzten Wochen der Entwicklungsphase intensiv beschritten, was vermehrt mit dem Vorhaben zu tun hatte, mehr als zwei Spieler im Spiel zu unterstützen.

Die „*Gamelogic*" Konstruktorfunktion besitzt eine Funktion mit dem Namen „*gamelogic*", welche an ihr aufgerufen werden kann, was bei der Erstellung der Spieler Entities geschieht. Mit dem Aufruf werden die initialen Spielerstände an den betreffenden Spieler weitergegeben. Wie oben bereits erwähnt, wird das Objekt, welches alle wichtigen Eigenschaften für den Spieler enthält, wie z.B. Spielername, Geschwindigkeit, Anzahl der legbaren Bomben, Reichweite des Feuer, usw. übergeben. In diesem Objekt stehen des Weiteren die Tastaturbefehle, welche intern für die Steuerung des Spielers benötigt werden. Diese fünf Befehle werden für alle vier Laufrichtungen, sowie für das Legen der Bombe als ASCII Wert übergeben.

7.2.3 Tastatur Eventhandling

Durch Crafty kann über Events auf Tastatureingaben reagiert werden. Da jedoch bei der Ansteuerung von zwei Spielern in der alten Implementierung, die Tastaturbefehle den anderen Spieler beeinflussten und dieser somit immer zum Stillstand gelang, entschloss ich mich kurzer Hand dazu eine andere Ressource zu nutzten. Die Bibliothek jQuery,

welche mir aus dem Multimediaapplikationen Unterricht geläufig war, bot eine simple Möglichkeit, die Tastaturbefehle abzufangen. Das Handling ist so aufgebaut, das Tastenab und Tastenauf Events separat an alle Spieler gesendet werden. Durch die Übergabe der initialen ASCII Werte zur Steuerung der Spielfiguren ist es nun möglich, dass nur diejenigen Spieler auf die Events reagieren, welche auch auf den ASCII Wert initialisiert wurde. So muss auch keine Vorauswahl der Tastatureingaben getroffen werden. An der Gamelogic Komponente sind drei Events gebunden, was bedeute, wird ein Event an einem Spieler „*getriggert*", reagiert dieser darauf. Im Fall der Tasten Events können Spieler somit auf „*keyup*" und „*keydown*" Events reagieren.

7.2.4 Spielerbewegung

Durch die „*keydown*" Events wird in der „*Gamelogic*" ein Objekt mit den booleschen Variablen „*left*", „*right*", „*up*" und „*down*" auf wahr gesetzt. Gegenteiliges passiert mit dem Event „*keyup*", welches die jeweilige Richtung auf falsch setzt. Wie oben erwähnt sind drei Events an die Komponente gebunden. Das dritte Event ist das „*enterframe*" Event. Dieses Event wird durch Crafty für jeden neuen Frame aufgerufen. In der Callback Funktion des Events wird das Objekt, welches die aktuelle gedrückte Richtung enthält, jede Bildwiederholung neu abgefragt. Ist eine der Richtungen auf wahr gesetzt, werden folgende Punkte abgehandelt:

1. Die Animation zur Bewegung gestartet
2. Geprüft, ob sich vor dem Spieler etwas undurchdringbares befindet
3. Wenn nein, wird die Position des Spielers entsprechend verändert

Das setzten der neuen Position geschieht dadurch, dass auf die aktuelle Position die Geschwindigkeitsvariable „*speed*" in x- oder y-Richtung addiert bzw. subtrahiert wird.

7.2.5 Umpositionierung der Spieler

Im Original Masterblaster ist das Laufverhalten von Spielern an eine Art Gitter gebunden. Dadurch läuft der Spieler immer auf einer vordefinierten Bahn, welche er bei Richtungswechsel verlassen kann, jedoch geschmeidig wieder auf die naheliegendste Bahn zurück geschoben wird. Wichtig für die Implementierung dieses Verhaltens war somit die Ermittlung, wie entschieden wird, dass der Spieler auf der ihm naheliegendsten Bahn läuft. Die Lösung hierfür waren Modulo Operationen auf die Position des Spielers. Durch den Rest konnte entschieden werden, ob und in welche Richtung der Spieler geschoben werden musste. Um den Effekt des geschmeidigen Laufens zu erzielen, wurde je nach Restergebnis der Modulo Operationen *„1“* auf x bzw. y addiert oder subtrahiert. Dieses Laufverhalten macht sich besonders bemerkbar, wenn sich die Spielerfigur vor einem Hindernis befindet. Bei der Berechnung wird die Mitte des Spielersprites betrachtet, womit entschieden wird ob sich der Spieler bei Tastendruck gegen das Hindernis vorbei bewegen soll. Abbildung 3 soll diesen Vorgang verdeutlichen. Die Konzeption der *„PlayerRelocator“* Funktionen geschah zum Teil in Zusammenarbeit mit Maximilian.

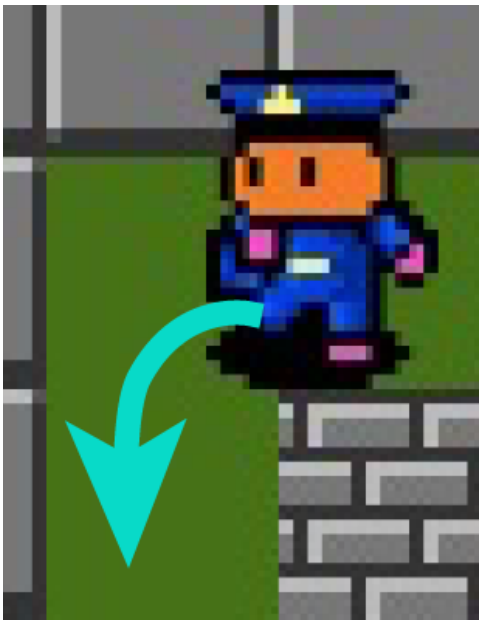

Abbildung 3: Bewegung

7.2.6 Animation der Spieler

Die Animation der Spielerfiguren geschieht in Folge des „*enterframe*“ Event Aufrufs. Wie erwähnt wird in deren Callback Funktion die Bewegungsrichtung des Spielers abgefragt. Trifft eine Richtung zu, wird die jeweilige Animation gestartet. Da der Spieler durch das Goody „*Invincible*“ seine Animation auch grundlegend ändern kann, ist es wichtig andere Sprites zur Animation angeben zu können. Ich entschied mich dazu, die Definition der Animationen somit in weiteren Komponenten auszulagern. Dies hat den Vorteil, dass die Komponenten je nach Gebrauch an der Spieler Entity hinzugefügt und auch wieder entfernt werden können. Um diese Komponenten jedoch für alle Spieler nutzbar zu machen, wird an die Komponente der Spielername übergeben, womit diese dann mit einer weiteren Funktion „*getPlayerCord*“ die Koordinaten aus der Spritemap für den jeweiligen Spieler geliefert bekommt. Probleme bereitete das Beenden der Animation. Erfolgt das Keyup Event unterbrach die Animation mit dem letzten Sprite der Animationskette. Dies hatte zur Folge das der Spieler manchmal einen bewegten Sprite angezeigt hat, obwohl die Spielfigur eigentlich still stand. Die Lösung war es, innerhalb des Keyup Events die laufende Animation zu stoppen und darauf hin eine Animation für eine stehende Spielerfigur zu starten. Zur Veranschaulichung ist in Abbildung 4 die benötigte Anordnung der Spielersprites gezeigt.

Abbildung 4: Spielersprite

7.2.7 Solid Test

Für die Spieler sind zwei Gegenstände undurchdringbar. Diese Beiden sind Wände und Blöcke. Zur Definition wo sich eine solide Einheit befindet, wird zur Initiierung der Entities „*brick*“ und „*wall*“ gleichzeitig ein Array namens „*brick_array*“ mit Number Werten befüllt, welche die jeweiligen Einheiten repräsentieren. Im Falle der Blöcke gibt es mehrere Zustände, welche hier berücksichtigt werden mussten. Um auf diese Entities später wieder zugreifen zu können, wurden die Entities ebenfalls in einem Array angelegt.

Um nun zu überprüfen, ob sich vor der Spielerfigur eine solide Einheit befindet, wird die Position des Spielers vor dem Verändern durch vier Funktionen überprüft, jede für die entsprechende Laufrichtung. Leider musste ich vier verschiedene Funktionen schreiben, da jede Richtung ihre besonderen Feinheiten mit sich brachte. Diese Funktionen überprüfen, ob die Position des Spielers relativ mit einer Einheit im oben erwähntem Array übereinstimmt. Befindet sich also eine „*2*“ im Array, teilt die Funktion der Gamelogic mit, dass der Spieler hier nicht weiterlaufen kann.

7.2.8 Bomben

Die zentrale Fähigkeit, welche der Spieler von Anfang an besitzt, ist das Legen von Bomben. Der Spieler kann zu Beginn lediglich eine Bombe platzieren, wenn diese dann explodiert kann er eine weitere legen. Diese Anzahl der legbaren Bomben kann durch das „*bombup*“ Goody erhöht werden. Um nun einen Bombe zu legen, muss zu Beginn nach Erhalten des Tastendrucks die Position des Spielers zur relativen Position im Gitter umgerechnet werden. Dadurch wird gewährleistet, dass eine Bombe immer an der richtigen Stelle gelegt wird. Anschließend wird überprüft, ob die Anzahl der gelegten Bomben kleiner als die Anzahl der maximal

legbaren Bomben des Spielers ist. Ist dies gegeben, wird überprüft, ob sich an der Stelle, wo eine Bombe platziert werden soll, sich bereits eine Bombe befindet. Die Abfrage geschieht hier auch wieder über ein Array, in dem eine Bombe durch eine Zahl repräsentiert wird. Wenn nun diese Abfragen erfolgreich durchlaufen werden, wird ein neues „*bomb*" Entity erstellt, welches zugleich in ein Bombenarray geschrieben wird. Dies hat den Grund, dass das Feuer einer Bombe eine andere noch tickende Bombe zur Detonation bringen kann. Um also dies zu realisieren ist ein Bombenarray mit allen „*bomb*" Entities von Nöten, um diese referenzieren zu können. Ist die Entity gesetzt, wird diese animiert. An dem bomb Entity ist des Weiteren das Event „*explode*" gebunden. Dadurch ist es möglich die Bombe gezielt zu zerstören (Detonation durch Explosion benachbarter Bomben). Die Zerstörung der Bombe wird im Normalfall nach einer Zeit von drei Sekunden ausgeführt. Diese Zeit kann zusätzlich durch ein entsprechendes Goody verringert werden. Wird das Event „*explode*" der Bombe aufgerufen, sorgt eine weitere Funktion für das Setzten der Feuer Entities in die entsprechenden vier Richtungen.

7.2.9 Feuer

Das Feuer wird in der Funktion „*bombExplosion*" erstellt. Diese erhält die x- sowie y-Position der Bombe, sowie eine Referenz an den Aufrufenden Spieler. Die Explosion einer Bombe verursacht im Normalfall einen Feuerradius von 2 Einheiten. Das Besondere ist die Ausbreitung des Feuers, denn diese geschieht zeitversetzt. Anfänglich versuchte ich die Ausbreitung über eine for-Schleife, welche bei jedem Durchlauf kurz pausieren werden sollte, zu realisieren. Dadurch kam es jedoch zu merkwürdigen Fehlern. Im Gespräch konnte mir Prof. Dr. Hopf weiterhelfen, indem er mir nahelag die Feuer Entities neue Feuer Entities aufrufen zu lassen. Für diese mussten dann lediglich einen Richtungsvektor angegeben werden, um zu bestimmen in welche Richtung

diese weitere Feuer Entities setzten sollen und wie viele Male dieser Vorgang noch wiederholt werden sollte.

Wurde ein Feuer gesetzt, handelt deren Komponente „*SetFire*“ noch einige wichtige Abfragen ab. Zu aller erst wird überprüft, ob sich an der Stelle etwas befindet, was durch das Feuer zerstört wird. Ist dies der Fall wird die Ausbreitung des Feuers nicht weiter geführt. Um herauszufinden, ob sich ein Spieler an der vom Feuer getroffenen Stelle befindet, wird über alle Spieler Entities gelooped und deren Position mit der des Feuers verglichen. Somit ist auch sicher gestellt, dass mehrere Spieler auf dem gleichen Feld durch das Feuer getroffen werden, was in früheren Versionen des Remakes nicht funktioniert hat. Wurde also ein Spieler getroffen, wird die Sterbeposition des Spielers auf die des Feuers gesetzt. Dies hat den Sinn, dass die Sterbeanimation auch da angezeigt wird, wo der Spieler getroffen wurde.

7.2.10 Goodys

Goodys sind Gegenstände, welche beim Sprengen von Blöcken erscheinen können. Diese können von den Spielern aufgenommen werden, und deren Attribute, Fähigkeiten bzw. Erscheinung ändern. Im Spiel gibt es 10 Goodys, die der Spieler aufnehmen kann. Neben diesen Goodys gibt es noch einen weiteren Gegenstand, den Totenkopf. Dieser führt beim übertreten zum Tod der Spielerfigur. Von den zehn Goodys gelang es mir sechs zu implementieren. Aufgrund der begrenzten Zeit waren leider nicht mehr Goody Implementierungen möglich.

Die Entwicklung der Goodys begann mit den für uns am wichtigsten erscheinenden Goodys. Das waren unter anderem das „*bomb up*“, „*speed up*“ sowie „*fire up*“ Goody. Nachdem diese Goodys implementiert waren erstellte ich mit Sebastian eine priorisierte Liste mit den noch offenen Goodys. Wichtig war vor allem heraus zu finden, welche Goodys schneller zu implementieren sind.

Das Setzten der Goodys geschieht nach der Zerstörung eines Blocks. Wenn dieser vom Feuerradius einer Bombe getroffen wurde, wird eine neue Entities erstellt, welche die Animation eines verbrennenden Blocks darstellt. Bevor diese Entity zerstört wird, ruft diese noch die Funktion „*rollTheDiceForGoody*" auf. Diese Funktion entscheidet durch das Ziehen einer zufälligen Zahl, ob ein Goody generiert wird, oder nicht. Die Goodys im original Spiel haben Auftrittswahrscheinlichkeiten, welche wir durch den Verzicht der 4 Goodys so genau wie möglich eingebunden haben. Wird nun durch die Zufallszahl entschieden, dass ein Goody erstellt werden soll, wird die Funktion „*generateGoody*" aufgerufen. Um die Goodys im Verlauf des Spiels erkennen und ansprechen zu können, werden diese in das „*brick_array*", sowie in das „*goody_array*" für die Speicherung der Entities geschrieben.

Der zweite wichtige Teil der Implementierung der Goodys ist die Erkennung, ob ein Spieler über einen Gegenstand läuft. Hierfür bediente ich mich der Solid-Prüffunktionen. Läuft eine Spielfigur in eine Richtung, wird überprüft, ob sich ein Goody auf dessen Position befindet. Dazu wird in den Solid-Prüffunktionen die Funktion „*checkForGoodys*" aufgerufen. Erkennt diese über das „*brick_array*" einen Gegenstand, entscheidet diese welche Einflüsse auf den Spieler genommen werden müssen und entfernt das Goody danach.

7.2.11 Shrinking

Das Shrinking ist ein Mechanismus des Spiels, lang andauernde Runden zu beenden. Dieser tritt nach 90 Sekunden auf und bewirkt, dass das Spielfeld nach und nach verkleinert wird. Dazu werden neue Wände in einer Spirale nach innen gesetzt.

Um diesen Mechanismus zu realisieren, entwickelte ich eine Komponente, welche zur Funktionsweise der Feuerkomponente gewisse Ähnlichkeit aufweist. Zu Beginn wird eine Entity erstellt, welche zeitverzögert eine

weitere erstellt usw. Ist eine Einheit an der Wand angelegt, muss diese entsprechend die Richtung ändern. Wie viele Wände noch gesetzt werden müssen, legt die „*wallsLeft*" Variable fest. Befindet sich ein Spieler an der Position an der eine neue Wand entsteht, wird dieser Spieler entsprechend zerstört.

Der Aufruf des Shrinking Mechanismus geschieht über die Standard JavaScript Funktion „*setTimeOut*". Dadurch dass wir keinen Weg gefunden haben eine Szene in Crafty zu zerstört, mussten einige Mechanismen eingebaut werden, welche es erlauben das Menü nach einer Runde in den Vordergrund zu befördern und bei erneutem Beginn einer Runde, Crafty ohne Seiteneffekte von neuem zu initialisieren. Besonders das Setzten einer Timeout Funktion, oder das Abspielen von Sounds machte diverse Schwierigkeiten, welche aber rechtzeitig behoben werden konnten. Neben dem Shrinking wird auch noch der Hintergrund um das Spielfeld animiert, welcher alternierend zwischen schwarz und rot wechselt. Diese Implementierung wurde von Sergej geliefert.

7.2.12 Ende der Runde

Ist nur noch eine Spielfigur am Leben, hat dieser Spieler gewonnen. Um dies festzustellen muss nach jedem „Tod" eines Spieler überprüft werden, wie viele Spieler noch leben. Falls nur noch einer am Leben ist, muss die Runde beendet werden. Hier kommt die Referenz auf die GUI Umgebung zum Einsatz, denn an dieser wird die Funktion „*gameFinished*" aufgerufen. Diese übergibt zusätzlich den modifizierten „*gameState*" zurück, worin am Gewinnerspieler ein Tag gesetzt wird, sowie das gesammelte Geld der jeweiligen Spieler vermerkt wird, um im Nachhinein damit im Shop Goodys für zukünftige Runden kaufen zu können.

8. Reflexion und Resümee

Nachdem die Entwicklungsphase der Projektarbeit des sechsten Semesters nun abgeschlossen ist, kann ich etwas objektiver an meine Entwicklungsarbeit an der Spielelogik heran treten. Durch das gesammelte Wissen während der Programmierarbeiten, als auch durch die erlernten Fähigkeiten aus meinem Modul Multimediaapplikationen würde ich jetzt einige Stellen der Programmierung etwas anders realisieren, was nicht heißen soll, dass die bewerkstelligte Implementierung schlecht ist. Jedoch könnte ich mit dem erlangten Wissen das ein oder andere Codefragment eleganter formulieren. So glaube ich, dass eben die Erlangung dieser Erkenntnis von großer Bedeutung ist. Gerade durch diverse Fehler die man während der Entwicklung macht, vor allem bei der für mich neuen Sprache habe ich enorm viel fachlich, als auch über mich gelernt. Ich freue mich auch darüber, mich dazu entschlossen zu haben, eine tragende Rolle in der Entwicklung des Spiels übernommen zu haben. Gerade dadurch habe ich mich selbst immer wieder angetrieben und mich zu neuen Lösungswege inspirieren können. Der Grundpfeiler unseres Projektes war ein solides Projektmanagement, sowie sehr gute Zusammenarbeit zwischen GUI und Spielelogik Entwicklern.

Ich bin auch davon überzeugt, wäre noch mehr Zeit für die Projektarbeit gewesen, wäre das Remake komplett gelungen. Nicht zu vergessen, dass wir schon weit über unsere anfänglich gesteckten Ziele hinaus schreiten konnten.

Zu Anfang des Projektes hatte ich noch gewisse Abneigung gegenüber JavaScript, welche mittlerweile in wirkliche Begeisterung um geschwappt ist. So hat mir Prof. Dr. Hopf immer wieder aufgezeigt, welche eleganten Lösungen mit dieser Sprache möglich sind. Auch die schnelle Implementierung im Gegensatz zu höheren Sprachen begeistert mich enorm. So kann ich mir durchaus vorstellen in Zukunft mehr mit JavaScript zu machen.

Bedanken möchte ich mich hiermit bei meinen Teamkollegen, besonders Sebastian für sein tolles Projektmanagement. Weiter bedanke ich mich bei unseren Betreuern Prof. Dr. Hopf und Prof. Dr. Röttger für ihre Unterstützung und Kompetenz während der Projektarbeit.

Abbildungsverzeichnis