Fachhochschule der Wirtschaft

FHDW

Mettmann

**Seminararbeit**

Praxis im Software Engineering

Projekt Space War

**Prüfer:**

Dr.-Ing. Alexander Brändle

**Verfasser:**

Moritz Nebel

Am Mühlenteich 16

40822 Mettmann

**Abgabetermin:**

23.04.2015

# Projekt

Für das Modul „Praxis im Software Engineering“ sollte ein Spiel als Ergebnis erstellt werden. Das Spiel soll so ähnlich aufgebaut sein, wie das „Space Taxi“. Das Spiel soll unterschiedliche Levels beinhalten, indem eine Spielfigur auf einem Spielfeld von Ort A nach Ort B kommen soll, um in den nächsten Level zu gelangen. Gegnerische Spielobjekte sollen eingebaut werden, die dieses Vorhaben verhindern. Des Weiteren sollen Hindernisse eingebaut werden, die einen Durchgang der Spielobjekte verhindern sollen.

Da zunächst noch keine großen Anforderungen zu dem Spiel genannt wurden, versuchte ich in den ersten Schritten das Spiel webbasiert mit HTML5 und JavaScript zu gestalten. Als uns dann aber mitgeteilt wurde, dass das Spiel auch unter der Spiel-Engine Unity erstellt werden kann, versuchte ich nun das Spiel mit Unity zu erstellen, da es doch recht einfacher erschien als webbasiert. Video-Tutorials und Informationen von der Website: <http://docs.unity3d.com/Manual/index.html> halfen mir dabei, ein funktionstüchtiges Spiel mit Unity zu entwickeln. Mein Spiel lautet „Spice Driver.“

# Spiel

Das Spiel Space War enthält 6 verschiedene nacheinander ablaufende Levels, die zu absolvieren sind. Das Spielobjekt in Form eines rot-weißen Raumschiffes soll pro Level alle gelben Sterne einsammeln, um am Ende den goldenen Stern einzusammeln, um in den nächsten Level zu gelangen. Pro Level gibt es jedoch Hindernisse und Gegnerobjekte, die aber durch Einsammeln der Sterne oder Schießen beseitigt werden können. Zu Beginn besitzt das Spielobjekt 10 Leben. Wenn man ein Leben verliert oder die Zeit abgelaufen ist, beginnt der Level von vorne. Die Zeit ist pro Level unterschiedlich und ein Leben verliert man, wenn ein Gegner das Spielobjekt berührt oder abschießt. Sind alle Leben verspielt worden, beginnt es nochmal bei Level 1 mit 5 Leben. Wenn man alle 6 Levels durchgespielt hat, hat man die Möglichkeit, die jeweiligen Level noch einmal durchspielen. Man kann sich dabei aussuchen, welches der Level man nun spielen möchte. Hier ist die Anzahl der Leben nun ausgeschaltet, da man Space War durchgespielt hat.

# Spielobjekte

### Spielobjekt: PlayerShip

PlayerShip ist das Objekt des Spielers. Es kann sich mit den Pfeiltasten nach vorne und hinten bewegen und nach links und rechts drehen. In Unity wurde die Spielfigur mit einem selbsterstellten Bild aus Paint versehen, welches ein rot-weißes Raumschiff darstellt. Die Position der Figur ist je nach Level variabel. Damit das Spielobjekt per Pfeiltasten bewegbar ist, wurde dem Spielobjekt ein C#-Skript (**PlayerMovement.cs**) als Komponenten hinzugefügt. Folgender Code ist ausschlaggebend dafür:

*public float rotSpeed = 180f;*

*…*

*Quaternion rot = transform.rotation;*

*float z = rot.eulerAngles.z;*

*z -= Input.GetAxis("Horizontal") \* rotSpeed;*

*rot = Quaternion.Euler(0, 0, z);*

*transform.rotation = rot;*

Die z-Achse der Drehung wird angesprochen, und ihr Wert bei Verwendung der Pfeiltasten Rechts und Links verändert. Die float-Variable rotSpeed dient als Schnelligkeitsfaktor für die Wertänderung für z, also der Drehung. Mit den Pfeiltasten Oben und Unten ist es möglich, das Spielobjekt nach vorne und hinten zu beordern. Folgender Code aus dem Skript **PlayerMovement.cs** ist ausschlaggebend dafür:

*public float maxSpeed = 5f;*

*…*

*Vector3 pos = transform.position;*

*Vector3 velocity = new Vector3(0, Input.GetAxis("Vertical") \* maxSpeed \* Time.deltaTime, 0);*

*pos += rot \* velocity;*

Beim Betätigen dieser Tasten wird zunächst erstmal, nur die x-Achse der Position verändert. Durch die letzte Zeile, wird es dem Spielfigur auch ermöglicht, aus oder bei einer Drehung nach vorne und hinten zu gelangen. Somit ändert sich auch je nach Ausrichtung die x-Achse der Position. Die float-Variable maxSpeed dient als Schnelligkeitsfaktor für die Wertänderung.

Damit die Spielfigur sich nur innerhalb der Kamerasicht bewegen darf, hilft folgender Code aus dem Skript **PlayerMovement.cs**:

*float shipBoundaryRadius = 0.5f;*

*…*

*if (pos.y + shipBoundaryRadius > Camera.main.orthographicSize)*

*{*

*pos.y = Camera.main.orthographicSize - shipBoundaryRadius;*

*}*

*if (pos.y - shipBoundaryRadius < -Camera.main.orthographicSize)*

*{*

*pos.y = -Camera.main.orthographicSize + shipBoundaryRadius;*

*}*

*float screenRation = (float)Screen.width / (float)Screen.height;*

*float widthOrtho = Camera.main.orthographicSize \* screenRation;*

*if (pos.x + shipBoundaryRadius > widthOrtho)*

*{*

*pos.x = widthOrtho - shipBoundaryRadius;*

*}*

*if (pos.x - shipBoundaryRadius < -widthOrtho)*

*{*

*pos.x = -widthOrtho + shipBoundaryRadius;*

*}…*

Die ersten beiden if-Abfragen überprüfen, ob das Spielobjekt mit seinem Radius, über die obere bzw. untere Kameragrenze hinausschreiten will. Wenn dies der Fall ist, verhindern die If-Abfragen dieses Vorhaben, in dem die y-Achse der Position des Objekts zurückgesetzt wird. Die letzten beiden If-Abfragen haben dieselbe Funktion für die linken und rechten Grenzen der Kamerasicht. Jedoch mussten die x-Achsenwerte der Länge durch das Ansichtsverhältnis und die orthographische Kameragröße berechnet werden, um diese in den if-Abfragen verwenden zu können.

Des Weiteren ist das Spielobjekt wie fast jedes andere in der Entwicklung verwendete Objekt mit den Komponenten Rigidbody2D und einem Collider 2D versehen. Im Rigidbody2D ist „Is Kinematic“ ausgestellt sowie „Gravity Scale“ auf 0 gesetzt, damit es unter anderem nicht runterfällt. Im Collider ist ein Haken bei „Is Trigger“ gesetzt, damit es bei Berührung mit anderen Spielobjekten reaktionsfähig ist.

Des Weiteren beinhaltet das Spielerobjekt ein Skript als Komponenten: **TimeDamageHandler.cs**, indem sich die Programmierung für Ansicht und Einstellungen von Zeit und Leben in Bezug auf den Spieler befindet:

*void OnTriggerEnter2D(Collider2D Player)*

*{*

*if (Player.tag == "Enemy")*

*{*

*health--;*

*}…*

Hier wird eine in der Spielentwicklung des Spiels häufig verwendete If-Abfrage verwendet, die ein mögliches Szenario liefert, wenn die Spielerfigur mit der Gegnerfigur in Kontakt gerät. In vielen Skripten wird diese Abfrage benutzt: dazu gehören: DamageHandler.cs für die spielende Spielfigur, EnemyHandler.cs für die Gegner, SelfDestruct.cs für das Schussobjekt, GoalHandler für die einzusammelnden Objekte (Sterne). In diesem Fall bekommt

der Spieler ein Leben weniger. Health bezieht sich auf das aktuelle Leben, welches immer nur 1 ist.

**TimeDamageHandler.cs**

*public Text Leben;*

*public float health = 1;*

*public static float live = 10;*

*public GameObject Blocks;*

*public float timeLeft = 5;*

*public Text Zeit;*

*…*

*void Update()*

*{*

*if (live <= 10 && live > 0)*

*{*

*timeLeft -= Time.deltaTime;*

*Leben.text = "Leben: " + live;*

*Zeit.text = "Zeit: " + timeLeft.ToString("0.0");*

*if (health <= 0 || timeLeft < 0.1)*

*{*

*Destroy(gameObject);*

*Destroy(Blocks);*

*live--;*

*Application.LoadLevel(Application.loadedLevel);*

*if (live == 0)*

*{*

*Destroy(GameObject.FindWithTag("Music"));*

*Application.LoadLevel("Main");*

*live = 5;*

*}*

}

Level 6:

**MenuLoader.cs**

*if (PlayerShip.tag == "FinalGoal")*

*{*

*if (GameObject.FindGameObjectsWithTag("Goal").Length == 0)*

*{*

*Application.LoadLevel("Menu");*

*DamageHandler.live = 0;*

*}*

*}*

**TimeDamageHandler.cs**

*…*

*if (live < 0)*

*{*

*timeLeft -= Time.deltaTime;*

*Leben.text = "";*

*Zeit.text = "Zeit: " + timeLeft.ToString("0.0");*

*if (health <= 0 || timeLeft < 0.1)*

*{*

*Destroy(gameObject);*

*Destroy(Blocks);*

*Application.LoadLevel("Menu");*

*}*

*}*

*}*

Dieser Code demonstriert die Anzeige für die vorhandenen Leben und die aktuell verbleibende Zeit für einen Level. Die Ansicht für Leben und Zeit befindet sich meist mittig in der oberen Spielhälfte. Die Zeit ist je nach Level unterschiedlich. Man startet ein Spiel mit 10 Leben. Ist die Zeit abgelaufen oder hat man das aktuelle Leben verloren, werden alle einzusammelnden Sterne und das Spielobjekt gelöscht. Die Anzahl der Gesamtleben minimiert sich um eins und der aktuelle Level wird neu geladen, es sei denn die Anzahl der Leben wurde schon verspielt, dann gelangt man in das Startmenü und fängt wieder mit 5 Leben von vorne an. Die aktuell laufende Musik wird hierbei auch zerstört, damit diese nicht doppelt abgespielt wird. Sollte man im letzten Level das Ziel erreicht haben, gelangt man ins Levelmenü, wo man die Wahl über alle Level hat, die man spielen möchte. In diesem Fall wird die Anzahl der Leben nicht mehr angezeigt und auf 0 gesetzt. Wenn man nun verliert, gelangt man wieder ins Levelmenü.

Des Weiteren beinhaltet der PlayerShip das Skript **PlayerShooting.cs,** das es dem Spieler ermöglicht, Gegner abzuschießen:

*public Vector3 bulletOffset = new Vector3(0, 0.7f, 0);*

*public GameObject bulletPrefab;*

*public float fireDelay = 0.25f;*

*float cooldownTimer = 0;*

*void Update () {*

*cooldownTimer -= -= 0.01f;*

*if(Input.GetButton("Fire1") && cooldownTimer <= 0)*

*{*

*cooldownTimer = fireDelay;*

*Vector3 offset = transform.rotation \* bulletOffset;*

*GameObject bulletGO = (GameObject)Instantiate(bulletPrefab, transform.position + offset, transform.rotation);*

*bulletGO.layer = gameObject.layer;*

*}*

Dem Skript wird ein Schussobjekt in Form eines blauen Streifens hinzugefügt haben. Positioniert wird es über dem Spielobjekt. Je nach Rotierungsposition wird das Schussobjekt nach vorne geschossen. Für diese Berechnung dient *Vector3 offset = transform.rotation \* bulletOffset;.* Hier multiplizieren sich die Werte der Ausgangssituation mit den normalen Positionswerten des Objekts. Per Abfrage wird festgestellt, ob ein Button zum Schießen gedrückt wurde (Strg-Taste). Die cooldownTimer-Variable dient dazu, dass nicht zu viele Schüsse nacheinander geschossen werden können. Die Variable FireDelay gibt einen bestimmten Zeitwert für eine maximale Anzahl von Schüssen pro Zeittakt an.

Unabhängig vom Spielerobjekt gibt es außerdem ein weiteres Skript, welches dem PlayerShip als Komponente hinzugefügt ist: **LoadNextLevel.cs**:

*if (PlayerShip.tag == "FinalGoal")*

*if (GameObject.FindGameObjectsWithTag ("Goal").Length == 0) { …*

*Application.LoadLevel ("Game2." + (currentLevel + 1));*

*…*

Hier wird bei Erreichen des goldenen Sterns der jeweilige nächste Level geladen. Durch public-Deklaration ist es recht einfach, die Zahl des jeweiligen nächsten Levels separat anzugeben. Das Ziel wird erst erreicht wenn alle Objekte mit Tag „Goal“ (alle Sterne) eingesammelt worden sind *(if (GameObject.FindGameObjectsWithTag("Goal").Length == 0))*.

### Spielobjekt: Hindernisse

Ein Hindernis beinhaltet die Komponenten Rigidbody 2D und einen Circle Collider 2D. Der Trigger ist in diesem Fall ausgestellt und „Is Krimentic“ ist eingehackt. Hindernisse sind in Form von roten Wänden dargestellt (mit Paint erstellt), um Schussobjekte abzuwehren und für das Spielerobjekt und die Gegnerobjekte keinen Durchgang zu bieten. Eventuell werden ein oder mehrere Hindernisse entfernt, wenn man bestimmte Sternobjekte einsammelt.

### Spielobjekt: Kamera

Die Kamera zeigt das Spielfeld. Bis auf das Spielerobjekt dürfen sich andere Objekte wie Gegner, Hindernisse und Schussobjekte aus dem Spielfeld befinden oder rausbewegen.

### Spielobjekt: Gegner (Enemy)

Gegner sind Spielobjekte, die das Spielobjekt verfolgen und unter Umständen (je nach Level) dieses auch abschießen können. Wie das Spielobjekt besitzt es die Komponenten: Ridigbody 2D und einen Collider 2D mit denselben Einstellungen wie beim Spielerobjekt. Es beinhaltet außerdem als Komponente das Skript **EnemyShooting.cs**, welches einen ähnlichen Aufbau wie **Playershooting.cs** hat. Der Unterschied besteht in dem automatisierten Schießen der Gegner ohne Tastendruck (Ein if-Parameter fällt weg). Des Weiteren besitzt es das Skript **MoveForward.cs**, die die programmatische Fortbewegung der Gegner umsetzt:

*public float maxSpeed = 15f;*

*void Update () {*

*Vector3 pos = transform.position;*

*Vector3 velocity = new Vector3(0, maxSpeed \* Time.deltaTime, 0);*

*pos += transform.rotation \* velocity ;*

*transform.position = pos;*

Die automatische Bewegung entsteht durch Time.deltaTime, welches andauernd ansteigt und sich somit ständig mit dem eingestellten Speed (maxSpeed) multipliziert. Es bewegt sich je nach Rotation, diese kommt durch das Skript **FacesPlayer.cs** als Komponente zur Geltung:

*public float rotSpeed = 90f;*

*Transform player;*

*void Update () {*

*if (player == null)*

*{*

*GameObject go = GameObject.Find("PlayerShip");*

*if (go != null)*

*{*

*player = go.transform;*

*}*

*}*

*Vector3 dir = player.position - transform.position;*

*float zAngle = Mathf.Atan2(dir.y, dir.x) \* Mathf.Rad2Deg -90;*

*Quaternion desiredRot = Quaternion.Euler(0,0, zAngle);*

*transform.rotation = Quaternion.RotateTowards(transform.rotation, desiredRot, rotSpeed);*

*…*

Hier wird berechnet, in welchem Winkel das Gegnerobjekt sich fortbewegen soll, um sich in Richtung des das spielenden Spielers fortzubewegen. Hierbei hilft unter anderem die Tangentenberechnung für den Winkel zwischen Spielerobjekt und Gegnerobjekt und die Subtraktion der Matrix Wert von Spieler und Gegner. Die Variable rotSpeed gibt die Geschwindigkeit der Anfangdrehung an.

Sollte ein Gegnerobjekt von einem Schussobjekt des Spielers getroffen worden sein, wird dieses zerstört. Das Skript **DamageEnemy.cs** zeigt dessen Programmierung:

*public float health = 1;*

*void OnTriggerEnter2D(Collider2D Enemy)*

*{*

*if (Enemy.tag == "Bullet")*

*{*

*health--;*

*}*

*if (health <= 0)*

*{*

*Destroy(gameObject);*

*}*

Durch das Berühren mit einem Schussobjekt, verliert der Spieler das aktuelle Leben und wird aufgrund der zweiten Schleifen letzten Endes zerstört.

### Spielobjekt: Bullet (Schussobjekt)

Ein Schussobjekt wird nach Betätigen der Strg-Taste abgefeuert, wenn es vom Spieler aus abgeschossen wurde. Schüsse vom Gegner werden automatisch abgefeuert. Schussobjekte sehen immer gleich aus: ein blauer Strich. Jedoch gibt es zwei verschiedene Vorlagen (Prefabs), die mit unterschiedlichen Tags verwiesen wurde. Eine Vorlage für Schüsse des Spielers (Bullet01) und eine Vorlage für Schüsse der Gegner (Bullet02). Dieser Auseinanderhaltung ist deswegen notwendig, da ein Gegner sonst möglicherweise einen anderen Gegner abschießen kann. Genau wie die Gegner besitzt auch ein Schussobjekt das Skript **MoveForward.cs** als Komponente. Der Unterschied ist aber, dass diese kein **FacesPlayer.cs** besitzen, die das Spielerobjekt verfolgen könnten. Es besitzt als weitere Komponente neben dem Rigidbody 2D und einem Collider 2D ohne Trigger die Skripte **SelfDestruct.cs** (für Bullet01) und **SelfDestruct2.cs** (für Bullet02) als Komponenten:

**Selfdestruct2.cs**

*if (Bullet2.tag == "Hind")*

*Destroy(gameObject);*

*if (Bullet2.tag == "Goal")*

*Destroy(gameObject);*

*if(Bullet2.tag == "Enemy")*

*Destroy(gameObject);*

Ein Hindernis wird zerstört wenn es ein Hindernis, ein Stern oder einen Gegner trifft.

### Spielobjekt: Goal (Sterne)

Sterne müssen von dem Spielobjekt eingesammelt werden. Sind alle Sterne bzw. Teilziele eingesammelt, kann man bei Berührung mit dem goldenen „Fin“-Stern in den nächsten Level gelangen. Die Anzahl der Sterne variiert pro Level. Für das Zusammentreffen von Spieler und Stern ist folgende Programmierung aus dem als Komponenten angehangenes Skript **GoalHandler.cs** verantwortlich:

*public GameObject soundi;*

*public GameObject Hindernis;*

*void OnTriggerEnter2D(Collider2D Goal)*

*{*

*if (Goal.tag == "Player")*

*{*

*Destroy(gameObject);*

*Destroy(Hindernis);*

*GameObject soundy = (GameObject)Instantiate(soundi, transform.position, transform.rotation);*

*soundy.layer = gameObject.layer;*

*}…*

Hier wird das Ereignis beschrieben, wenn der Spieler einen Stern berührt. Der Stern wird aus dem Spielfeld entfernt und falls in Unity angegeben, wird ein Hindernis aus dem Spielfeld entfernt.

Wenn alle Levels erfolgreich beendet wurden, gelangt man über den „Win-Bildschirm“ ins Levelmenü und kann dort einzelne Levels nochmal spielen. Leben spielen, wie oben bereits erwähnt, keine Rolle

mehr und werden dementsprechend in der Spielansicht ausgeblendet.

# Sounds

Das Spiel beinhaltet folgende Sounds: Hintergrundmusik während des ganzen Spiels: Mbius – Space Tuna (lizenzfrei); Schießgeräusch (Selbstaufnahme) und Sterneinsammlung (Selbstaufnahme)

# Bilder

Die verwendeten Bilder für die Spielobjekte und Ansichten wurden mit Paint erstellt. Dazu gehören: Space-Taxi, Gegnerschiff, gelber Stern, goldener „Fin“-Stern, Hauptmenü, Startmenü, „Win“-Ansicht Hindernisse, Schussobjekt, teilw. Schriften.

# Programme

Für die Programmierung der Spielentwicklung wurde Unity und Visual Studio (C# und JavaScript) verwendet. Für die verwendeten Objekte, wurden mit Paint Bilder gezeichnet verwendet.

# Bugliste

-viereckige Figuren mit schwarzem Hintergrund 🡪 Problem hinsichtlich der Berührung von Figuren mit Hindernissen oder Schriften (Timer, Leben, Level)

-kein schönes Layout

-schwierige Formatierung auf allen Ausrichtungen 🡪 Probleme mit Spielrand und Schriftansicht

-Sounds nur in Nähe der Kamera

-Gegner häufig zu sehr an einer Stelle 🡪 Probleme mit der Darstellung

-kein Menü für Sound ein/aus

-keine Möglichkeit zur Spielunterbrechung

-keine Zwischenzeit zwischen den Levels

-erst ab der Skalierung 800 x 600 spielbar