Technology
Arts Sciences
TH Köln

Bachelorarbeit

Konzeption und Evaluation eines Job-basierten Entity-Component-Systems anhand einer Massenpaniksimulation auf Basis von Unity DOTS

TH Köln WS 19/20

vorgelegt von:	Marvin Nicholas Hallweger
Matrikel-Nr.:	11117481
Adresse:	Hermelinweg 9
	58553 Halver
	marvin_nicholas.hallweger@smail.th-koeln.de
eingereicht bei:	Prof. Dr. Lutz Köhler
Zweitgutachter	Prof. Dr. Wolfgang Konen

Gummersbach, 17.02.2020

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum	Rechtsverbindliche Unterschrift

Thema der Arbeit

Konzeption und Evaluation eines Job-basierten Entity-Component-Systems anhand einer Massenpaniksimulation auf Basis von Unity DOTS

Kurzfassung

Diese Arbeit umfasst die Implementierung einer Standalone-Festival-Massenpaniksimulation auf Basis von der *Unity3D* (Spiel-Engine)-Erweiterung *Unity DOTS (**D**ata-**O**riented-**T**echnology-**S**tack)*. Hierfür werden zuerst ähnliche Softwareprojekte und die entsprechende Erweiterung analysiert, um im Anschluss eine eigene Massenpaniksimulation zu realisieren. Das Ergebnis ist eine voll funktionsfähige Massenpaniksimulation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CPU	central processing unit
GPU	graphics processing unit
DOTS	Data Oriented Technology Stack
ECS	Entity Component System
FPS	frames per second
ms	millisecond

1. Einleitung

Rückblickend auf das Jahr 2010 gibt es ein Ereignis, welches bis heute in den Köpfen der Menschen ist, sobald das Wort “Massenpanik” fällt, die Massenpanik auf der “Loveparade” in Duisburg. An diesem Tag kamen 21 Menschen ums Leben, 541 Menschen wurden schwer verletzt und mindestens sechs Menschen begingen laut dem Selbsthilfeverein LoPa-2010, durch die Katastrophe ausgelöste andauernde seelische Belastung, Suizid. Die Veranstaltung wurde daraufhin vollständig eingestellt. Laut Staatsanwaltschaft war die Genehmigung der Loveparade 2010 rechtswidrig, die zu Beginn von der Staatsanwaltschaft genannte Zahl von 16 Hauptangeklagten verringerte sich im laufe der Jahre auf sechs, diese wurden im Februar letzten Jahres freigesprochen[37].

Viele Menschen fragen sich bis heute, was genau schief gelaufen ist und wie man dieses Ereignis hätte verhindern können. Die wenigsten allerdings versuchen ernsthaft eine dauerhafte Lösung für solche Veranstaltungen zu finden.

Anmerkung

Auf den ausdrücklichen Wunsch von Prof. Dr. Lutz Köhler wurde die im Rahmen des Studienverlaufsplans erstellte Praxisprojektdokumentation in dieser Bachelorarbeit integriert. Dadurch soll eine vollständige Transparenz des Projektes garantiert werden.

1.1 Motivation

Heutzutage ist so gut wie jede CPU eine Mehrkern-CPU und dadurch in der Lage, mehrere Prozesse parallel zu berechnen. Nebenläufige Abläufe in programmen sind dabei extrem effizient und sparen dadurch Zeit und Geld. Der Spieleindustrie haben diese Vorteile bis jetzt kaum bis keinen Nutzen eingebracht, wodurch man sehr selten Spiele findet, welche mit ihrer multicore performance überzeugen können[26,23], ganz im Gegenteil, heutzutage werden "AAA Spiele" speziell auf den Single-Core Betrieb optimiert, wodurch der multicore Betrieb noch weiter "in den Hintergrund" verdrängt wird.
Durch die Einführung von *Unity DOTS (Data-Oriented-Technology-Stack)*, bestehend aus dem *Entity Component System (ECS)*, dem *Burst-Compiler* sowie dem *C#-Jobsystem*, präsentiert die Game Engine *Unity3D* eine neue Möglichkeit, Spiele hoch performant zu entwickeln. Die Game Engine übernimmt dabei die komplexen Aspekte der nebenläufigen Multithreadprogrammierung und warnt zusätzlich den Entwickler vor potentiellen Laufzeit- sowie Wettlaufproblemen. Der Entwickler kann sich somit mehr auf sein eigenes Projekt fokussieren, da sich die Game Engine um die Hintergrundabläufe der nebenläufigen Multithreadprogrammierung kümmert[38].

1.2 Problemstellung

Die Einführung von *Unity DOTS* hat eine neue Grundlage der datenorientierten Programmierung in *Unity3D* geschaffen. Diese Grundlage ermöglicht es den Entwicklern größere und detailliertere Welten zu erschaffen, welche die volle Kapazität des Prozessors ausnutzen und dabei deutlich effizienter arbeiten können.
Beispielprojekte die auf verschiedenen *Unity*-Konferenzen vorgestellt wurden, konnten auf anhieb überzeugen und sollen die zukünftige herangehensweise der Spieleentwicklung zeigen.

Da sich *Unity DOTS* allerdings immer noch in der "pre-Alpha"-Phase befindet, existieren kaum Informationen, welche die herangehensweise für die Erstellung solcher Projekte beschreiben.
Auf Basis einer Massenpanik soll nun ein solches Projekt entstehen. Dabei soll die praktische Anwendung von *Unity DOTS* vorgestellt werden. Vordergründig soll die Entwicklung einer Massenpaniksimulation auf einem Festivalgelände stehen.

1.2.1 Aktueller Forschungsstand der Problemstellung

Aktuelle Projekte, welche *Unity DOTS* als Grundlage verwenden, stammen meistens von *Unity* selbst. Diese beziehen sich auf verschiedene Alltagsprobleme in der Spieleentwicklung und präsentieren dabei nur einzelne Lösungsansätze von verschiedenen Problemen. Größere Beispielprojekte wie das *Boids* Projekt [34] präsentieren zwar vollständig umgesetzte Ideen und beziehen sich auf größere Mengen von Entitäten, allerdings agieren diese nur individuell für sich selbst und handeln nicht auf einer kollektiven Ebene, es findet also keine gegenseitige Beeinflussung statt.
Durch weitere Recherche wurde herausgefunden, dass verschiedene *Unity* Projekte sowie plugins existieren, welche eine Massenpanik behandeln, diese wurden allerdings auf der normalen *MonoBehaviour* Ebene entwickelt, welche die normale Vorgehensweise der Spieleentwicklung in *Unity* darstellt.

1.3 Ziel der Arbeit

Ziel der Arbeit ist die Implementierung einer Standalone-Festival-Massenpaniksimulation auf Basis der neuesten *Unity3D* Erweiterung *Unity DOTS*. Der Nutzer hat dabei die Möglichkeit, das Festivalgelände nach seinen wünschen zu verändern, dabei spielt es keine Rolle, ob er sich auf das Gelände selbst oder auf die Anzahl der Besucher bezieht. Neben der Anpassung des Festivalgeländes, soll der Nutzer die Fähigkeit haben, eine Massenpanik auszulösen. Dem Nutzer wird mit diesem Projekt die Möglichkeit gegeben, verschiedene Szenarien auf einem Festival zu simulieren.
Großes Augenmerk wird auf die Grundfunktionalitäten der Konzertbesucher gesetzt, diese umfassen neben dem “laufen”, “tanzen” und dem “rennen” auch das allgemeine Panikverhalten eines einzelnen Konzertbesuchers.
Hierzu gehört die Erarbeitung einer passenden Architektur des *ECS*, welche im laufe der Arbeit implementiert und durch das *Unity C#-Jobsystem* unterstützt wird.
Selbstverständlich werden die einzelnen *Unity DOTS* Komponenten (*ECS*, *C#-Jobsystem*, *Burst Compiler*) im laufe der Arbeit vollständig aufgegriffen und in das Projekt integriert.

1.4 Vorgehensweise

Kapitel 2 präsentiert die Grundlagenbasis dieser Arbeit. Nachdem gezeigt wird, wie genau eine Massenpanik entsteht, wird danach der psychologische Faktor einer Massenpanik vorgestellt. Zudem werden andere Softwareprojekte vorgestellt, welche sich bereits mit der Simulation von Menschenmassen befasst haben.
Im Anschluss wird die Spiel-Engine *Unity3D* und deren Erweiterung *Unity DOTS* präsentiert und gezeigt, welche in *Unity* entwickelten Massenpaniksimulationen bereits existieren.

Kapitel 3 befasst sich mit der allgemeinen Konzeptionierung des Projektes. Es wird zuerst auf die Plattformauswahl eingegangen, danach wird ein Konzept für die Umwelt sowie für die Benutzeroberfläche erstellt. Im Anschluss folgt die Konzeptionierung des *ECS* der Massenpaniksimulation, sowie eine Auflistung der entsprechenden Nachteile des *ECS*.

Kapitel 4 geht auf die allgemeine Umsetzung und Implementierung der Simulation ein. Hierfür wird zuerst gezeigt, wie das Konzertgelände selbst, sowie die Benutzeroberfläche der Simulation umgesetzt wurde. Im Anschluss folgt die Implementierung der *Unity DOTS* Bestandteile in das Projekt.

Kapitel 5 evaluiert das Projekt, indem auf die Nutzung der verwendeten *Unity DOTS* Bestandteile während der Entwicklung eingegangen wird. Zusätzlich wird der allgemeine Funktionsumfang der Simulation vorgestellt. Zum Schluss wird die Qualitätssicherung sowie die Erweiterbarkeit des Projektes aufgegriffen.

Kapitel 6 beendet die Arbeit. Hierfür wird die Arbeit kurz zusammengefasst. Danach folgen einige Erfahrungen und Bemerkungen, welche während der Arbeit entstanden sind, sowie ein kurzer Zukunftsausblick.

2. Grundlagen

Das folgende Kapitel vermittelt allgemeines Wissen für die Entwicklung einer Massenpaniksimulation anhand von *Unity DOTS*. Dazu gehören Unterkapitel, welche sowohl die allgemeine Psychologie einer Massenpanik, als auch die Psychologie eines Konzertbesuchers beschreiben. Zusätzlich wird die Spiel-Engine *Unity3D* und deren Erweiterung *Unity DOTS* aufgegriffen, parallel werden Projekte präsentiert, welche einerseits auf dieser Technologie basieren oder andererseits über eine andere Herangehensweise entstanden sind.

2.1 Warum entstehen Massenpaniken?

Menschen fühlen sich in größeren Menschenmassen geborgen. Das liegt unter anderem daran, dass schnell ein Gemeinschaftsgefühl entsteht, welches gleichzeitig Glücksgefühle auslöst. Die Anonymität in der Masse sowie die Freude anderer Menschen rundet das Wohlbefinden eines Menschen in einer Menschenmasse ab[40]. Diese Euphorie sorgt immer wieder dafür, dass sich größere Menschenmassen bilden können. Dabei spielt es keine Rolle, ob es sich um ein Festival, ein Konzert oder einer religiösen Großveranstaltung handelt, eine Massenpanik kann genau dann eintreten, sobald sich eine größere Menge von Menschen an einem Platz versammelt hat.
In den vergangenen 10 Jahren sind solche Ereignisse häufiger aufgetreten. Darunter zählen die “Loveparade” im Jahr 2010 mit 21 Toten und 541 verletzten[37],
die Pilgerfahrt nach Mekka 2015 mit 2411 Toten und 934 verletzten[41], sowie das neueste Ereignis, die Beisetzung des iranischen Generals Ghassem Soleimani mit mindestens 56 toten und über 200 verletzten[36].

2.2 Psychologischer Faktor bei einer Massenpanik

Um eine Simulation zu implementieren, welche sich auf eine Massenpanik bezieht, muss zuerst die Psychologische Ebene einer entstandenen Massenpanik durchdrungen werden. Aus diesem Grund bezieht sich das folgende Unterkapitel auf das allgemeine Verhalten eines Konzertbesuchers unmittelbar vor und nach dem ausbrechen einer Massenpanik. Außerdem wird die Massenpanik selbst aufgegriffen und deren wichtigsten Merkmale näher erläutert.

2.2.1 Das Verhalten eines Konzertbesuchers

Das Ziel eines jeden Konzertbesuchers ist es, Spaß auf diesem Event zu haben. Dabei achtet er nicht auf potentielle Gefahrenquellen die entstehen könnten, im Vordergrund steht lediglich das Ziel, einen guten Tag mit seinen Freunden und/oder bekannten zu verbringen. Dieser Punkt spielt dabei eine sehr wichtige Rolle, da hier ein wichtiges Merkmal eines Konzertbesuchers aufgegriffen wird. So gut wie jeder Konzertbesucher befindet sich mit Begleitung auf dem Event, in den seltensten fällen beschließt ein Mensch, alleine auf ein größeres Event zu gehen.
Da das Erlebnis für den durchschnittlichen Konzertbesucher im Vordergrund steht, kann davon ausgegangen werden, dass eventuell vorhandene Notausgänge nicht beachtet, beziehungsweise nicht eingeprägt werden. Der durchschnittliche Konzertbesucher befindet sich also mit vielen weiteren Konzertbesuchern auf einem Gelände und weiß im Notfall nicht, wie er reagieren soll.

Sollte zu diesem Zeitpunkt ein Ereignis eintreten, welches eine Menschenmenge zur Flucht anregt, wüssten die meisten Konzertbesucher nicht, wie sie reagieren sollen. Die höchste Priorität in diesem Fall wäre die Flucht zum nächsten Ausgang, diese Flucht würde allerdings nicht alleine geschehen, sondern mit den Freunden und/oder bekannten des Konzertbesuchers. Diese Flucht würde “Herdenweise” ablaufen, dabei würden sich kleinere Gruppen bilden, welche mit der Zeit stetig wachsen würden, da eine “Herde” einer anderen folgt und beide somit “verschmelzen”[42].

Ein weiteres Merkmal eines Konzertbesuchers ist das individuelle Verhalten bzw. Auftreten. Dieses spielt eine wichtige Rolle, da dieses die einzelnen “Herden” dominanter erscheinen lassen kann. Das individuelle Verhalten bzw. auftreten kann sich dabei beispielsweise auf die größe und das Gewicht eines jeden Konzertbesuchers beziehen. Während manche Konzertbesucher schnell die Gefahr erkennen und aggressiver reagieren, verfallen andere hingegen schnell in Panik und agieren unkontrolliert. Das individuelle Verhalten eines jeden Konzertbesuchers ist dabei schwer kalkulierbar.

2.2.2 Massenpanik laut Michael Schreckenberg

Der theoretische Physiker und Panikforscher Michael Schreckenberg analysierte 127 Situationen in denen Fluchtbewegungen von Menschen zu Katastrophen führten. Dabei stellte sich heraus, dass die meisten Menschen weder egoistisch noch unüberlegt reagieren [28], laut Schreckenberg neigen etwa ein Prozent aller Menschen zu panikartigem und somit irrationalen Verhalten. Die Panik allein kann somit als Grund nicht mehr im Mittelpunkt stehen. Die Flüchtenden sind dabei Opfer von verschiedenen physikalischen Prozessen, welche auf vorhersagbaren Regeln basieren. Auf etwa zehn Personen kommt ungefähr ein Anführer, dieser stellt eine Leitfigur der Gruppe dar. Weitere zehn Prozent dieser Gruppe bilden laut Schreckenberg "die Sensiblen" ab, diese Kleingruppe läuft bei kleinsten Gefahren los und ist somit Hauptverursacher einer Fluchtreaktion. Die restlichen 80 Prozent sind laut Schreckenberg diejenigen, die der Masse "blind" folgen. Sollte eine Masse ins Stocken geraten, ändern Menschen nach nahezu exakt 15 Sekunden die Richtung in die sie laufen möchten.
Außerdem ist laut Schreckenberg die Auswahl der Ausgänge zu beachten. Ein einzelner Mensch bevorzugt fast immer den Weg von dem er auch gekommen ist, sollte dieser sich aber zu diesem Zeitpunkt in einer Menschenmasse befinden, wird diese Information mit dem bevorzugten Ziel der Masse überschrieben. Sobald eine Masse einen Ausgang blockiert, wird er noch attraktiver für weitere Menschen. Es spielt dabei auch keine Rolle, ob sich ein weiterer Ausgang in unmittelbarer Nähe befindet, die Masse würde kollektiv versuchen den gleichen Ausgang zu wählen[28].

2.2.3 Prävention laut Michael Schreckenberg

Laut Schreckenberg ist es unerlässlich, dass Massen "fließen" können. "Wenn 50 Menschen von hinten drücken, so lastet auf dem ersten das Gewicht von einer Tonne", diese Aussage von Schreckenberg spiegelt die enorme physikalische Kraft wider, welche in solchen Situationen herrscht. In solchen Fällen müssen Fluchtwege so gebaut sein, dass Menschen "wie ein Schwarm Heringe", so Schreckenberg, hindurchfließen können.
Neben passenden Ausgängen betont Schreckenberg noch deutlich die Kommunikation mit der Masse. Eine "strikte Führung" über Lautsprecher sei dafür nötig, sodass die Menschen das Gefühl haben, dass die Situation unter Kontrolle gebracht wird. Selbst wenn das nicht mehr möglich sein sollte, dürften die Lautsprecher nicht schweigen, da sich die Menschen in einer solchen Situation noch mehr "allein gelassen" fühlen[28].

2.3 Simulation von Menschenmassen

Eins der bekanntesten tools für die Generierung von größeren Massen von Charakteren ist *Golaem Crowd*. Dieses *Maya*[1] plug-in wurde von dem französischen Unternehmen *Golaem* entwickelt, *Maya* präsentiert dabei ein Animationstool, mit welchem man einzelne Charaktere entwerfen kann, *Golaem Crowd* generiert aus diesen Charakteren gewünschte Massen.
Golaem Crowd erfreut sich bis heute an einer wachsenden Beliebtheit. Das plug-in wurde bereits in unzähligen Spielen aber auch in Filmen und Serien verwendet. Beispielsweise wurde das plug-in in verschiedenen “Fluch der Karibik” Filmen, in der Serie “The Walking Dead”, sowie ab der siebten Staffel auch in der beliebten Serie “Game of Thrones” verwendet, siehe Abbildung 2.1[43].

Abbildung 2.1: “Game of Thrones” Staffel 7, Feindgenerierung durch *Golaem Crowd* [44]

Fazit

Golaem Crowd konzentriert sich hauptsächlich auf die “partikel”-ähnliche Generierung der Charaktere. Diese besitzen dabei meistens eine simple Animation, welche permanent iteriert wird. Charaktere die weiter vorne stehen, erhalten dabei eine etwas detailliertere Animation, Charaktere die weiter hinten platziert sind, erhalten wenig detaillierte Animationen beziehungsweise in manchen Situationen, auch keine Animation. Der Grund dafür liegt in der Einsparung von Leistung, da die Bewegungen der etwas weiter hinten stehenden Charaktere nicht zu erkennen sind.

2.4 Bereits existierende Simulationen für Massenpaniken

In diesem Unterkapitel werden zwei verschiedene Simulationen präsentiert, welche den derzeitigen Stand in diesem Bereich verdeutlichen. Die beiden Fälle unterscheiden sich dabei hauptsächlich in der Art der Verwendung und Bedienung.

2.4.1 Massenpaniksimulation des Start-ups *accu:rate*

Das Münchener Start-up *accu:rate* hat es sich zur Aufgabe gemacht, gefährliche Situationen für eine Masse an Menschen frühzeitig zu erkennen. Die dafür in *Java* entwickelte Simulation basiert auf dem mathematischen "Optimal Steps model", welches auf der Biomechanik basiert. Dieses *model* beschreibt die simple Änderung des Auftrittswinkels eines Charakters bei Begegnung eines Hindernisses.
Das Start-up arbeitet mit verschiedenen Veranstaltern zusammen um einen sicheren Ablauf von Events zu garantieren. Beispielsweise wurde durch *accu:rate* eine Evakuierung des "Hanse Sail" Events in Rostock simuliert (zu sehen in Abbildung 2.2), dabei wurde ein Rückstau an einem bestimmten Weg ermittelt, sobald dieser Weg aus der Simulation entfernt wurde, erhielt man eine Reduzierung der Evakuierungszeit von fünf Minuten. Der Veranstalter sah das Risiko ein und blockierte den Weg.

Abbildung 2.2: Simulation der "Hanse Sail" (accu:rate) [45]

Das Start-up konzentriert sich bei der Entwicklung weniger auf die grafische Realität, vielmehr wird die Realität der Vorhersagen priorisiert.

Aus diesem Grund werden verschiedene Psychologie- und Sozialwissenschaftler in die Simulation mit einbezogen, um das Verhalten der Menschen so realistisch wie möglich wirken zu lassen[12].

Fazit

Diese Simulation stellt einen sehr guten Ansatz des Projektes dar, welches mit dieser Arbeit realisiert werden soll. Anhand der Simulation von *accu:rate* können unterschiedlichste Simulationen generiert und simuliert werden. Hierbei stellt sich allerdings die Frage, ob das Programm auch in der Lage ist, eine Paniksituation auf einem Festival/Konzertgelände mit zusätzlichen Ausgängen zu simulieren.

2.4.2 Massenpaniksimulation anhand von realen Probanden

Forscher des Max-Planck-Instituts, des Disney Research Zürich, der ETH Zürich und der Rutgers University überließen 36 Probanden einen virtuellen Avatar in einer dreidimensionalen virtuellen Umgebung. Die Probanden selbst befanden sich vor Computerbildschirmen (siehe Abbildung 2.3) und waren dabei in der Lage den jeweiligen Avatar zu steuern. Die Forscher stellten den Probanden dabei verschiedene Aufgaben, um die Verhaltensweisen zu analysieren, der Stresspegel wurde während des Experiments immer wieder künstlich angepasst, indem zeitlicher und finanzieller Druck ausgeübt wurde. Der finanzielle Druck bezog sich dabei auf die Bonuszahlung die jeder Teilnehmer am Ende des Experimentes erhalten hat.

Bei den Aufgaben, bei denen der Stresspegel niedrig gehalten wurde, zeigte sich ein allgemein ruhigeres Verhalten der Probanden, außerdem wurde festgestellt, dass Probanden immer die rechte Seite wählten, sobald sie versuchten an einem anderen Probanden vorbei zu kommen.

Um die Reaktionen in einer Notsituation zu analysieren, wurden die Probanden in ein unübersichtliches Gebäude transferiert, um eine Evakuierung zu simulieren. Das Gebäude enthielt nur vier Ausgänge, von denen allerdings nur einer passierbar war. Als Anreiz erhielten vereinzelte Mitglieder der Gruppe Signale zum richtigen Ausgang, der Gruppe selbst war diese Fähigkeit bewusst, allerdings konnte nicht festgestellt werden, wer diese Informationen erhielt. Um den Stresspegel neben dem zeitlichen und finanziellen Druck weiter zu erhöhen, wurden zusätzliche rote Beleuchtungen eingebaut, außerdem wurden Feuer an den nicht-passierbaren Ausgängen platziert.

Das Ergebnis dieses Experimentes zeigte eine erhöhte Anzahl an Kollisionen der Probanden. Der erhöhte Stresspegel sorgte für unsicheres und irrationales Verhalten unter den Probanden, deutlich wurde diese Erkenntnis an Positionen bei denen Entscheidungen getroffen werden mussten oder bei Gängen die letztendlich in “Sackgassen” endeten.

Durch die erhöhte Unsicherheit nehmen Menschen die umliegende Gruppe stärker wahr, was letztendlich zu einer Anziehung führt und dadurch gewisses “Herdenverhalten” entsteht[19].

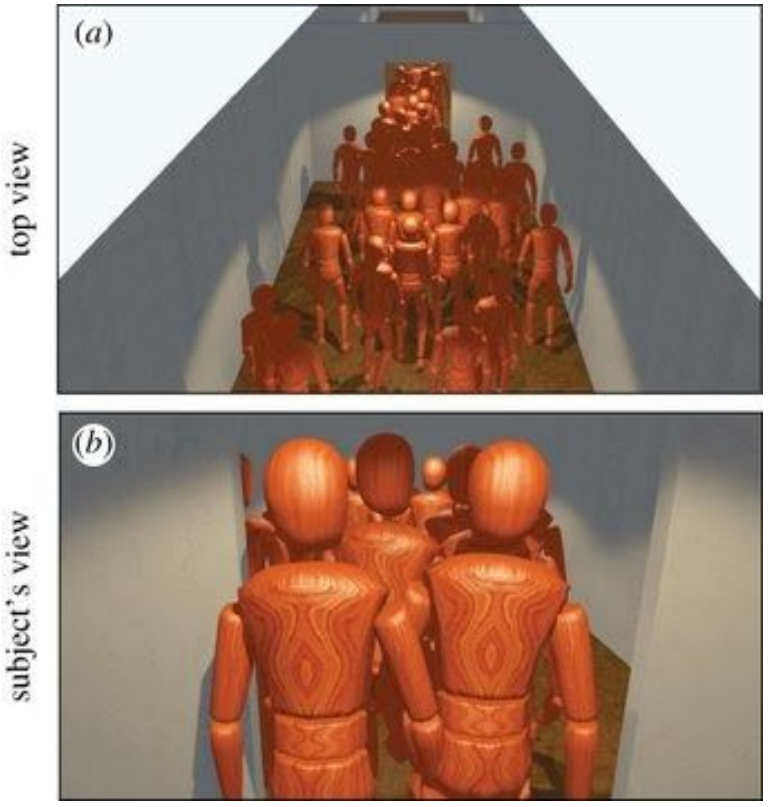


Abbildung 2.3: Reale Probanden, Oben: Sicht der Forscher, Unten: Sicht der Probanden [46]

Fazit

Diese Simulation bezieht sich auf das allgemeine Verhalten in einer Notsituation in einem geschlossenen Gebäude. Außerdem wurden in dieser Methode keine automatisierten Verfahren genutzt, sondern reale Menschen die sich vor einem Computerbildschirm befanden und miteinander agierten.

2.5 Die bisherige Entwicklung in der Spiel-Engine Unity3D

Die Spiel-Engine *Unity3D* nutzt das Programmierparadigma der objektorientierten Programmierung. Als Grundlage dafür, nutzt die Spiel-Engine so genannte “GameObjects”, diese präsentieren “container”, welche verschiedene Objekte der Spiel-Engine enthalten können. Will man beispielsweise einen Würfel darstellen, benötigt man ein *Renderer* Objekt, welches sich um die darstellung des Würfels kümmert und ein *Transform* Objekt, welches die aktuellen Maße und Positionswerte enthält. Somit existiert jedes Objekt in der aktuellen *Szene* als *GameObject*.

Für die Entwicklung steht dem Entwickler eine sehr große und ausführliche Dokumentation zur Verfügung. Diese kann genutzt werden, um in kürzester Zeit Informationen zu einem Vorhaben zu finden. Allgemein bietet die Spiel-Engine eine große Auswahl an Funktionen, welche hauptsächlich im 3D aber auch im 2D Bereich genutzt werden kann.

Nachteile der Spiel-Engine sind beispielsweise die hauptsächliche Nutzung eines "Main-threads", die Entwickler sind aus diesem Grund gezwungen, ihre Projekte für diesen "Main-thread" zu optimieren. Es besteht zwar die Möglichkeit, über die nebenläufige Programmierung mehrere *threads* mit einzubeziehen, dieses vorgehen ist allerdings sehr aufwändig, kompliziert und sehr fehleranfällig.
Allgemein werden die Projekte in *Unity* im laufe der Zeit sehr unübersichtlich, was ein erneutes aufgreifen von bereits implementierten dingen erschwert.

2.6 Unity DOTS

Das Ziel in der Simulation, welche in dieser Arbeit realisiert werden soll, ist die Skalierung der Konzertbesucher sowie deren realistische Reaktion bei einem "Panikereignis", ausgelöst vom Benutzer der Simulation.
Unity DOTS, bestehend aus dem *ECS*, dem *C#-Jobsystem* sowie dem *Burst-Compiler*, stellt alle notwendigen Komponenten bereit um dieses Vorhaben zu realisieren.

Das *ECS* löst dabei die objektorientierte Programmierung mit der datenorientierten Programmierung ab, um leistungsstärkeren code zu entwickeln, welcher sogar in anderen Projekten wiederverwendet werden kann. Das *ECS* besteht dabei aus sogenannten *Entities*, *Components* und *Systems*. Das Hauptprinzip in der datenorientierten Programmierung besteht in der Generierung von *Entities*, welche die Träger von *Components* sind, die *Systems* agieren auf diesen *Components*.
Das *C#-Jobsystem* greift den *ECS* code auf, um ihn in effizienten multi threaded code zu konvertieren, dabei werden die *Systems* des *ECS* durch sogenannte *Jobs* ergänzt, welche im laufe der Simulation durch den *Main-thread,* an die verschiedenen *Worker-threads* verteilt werden.
Der *Burst-Compiler* ergänzt das *ECS* und das *C#-Jobsystem* zusätzlich, indem er den entwickelten code in hoch performanten Maschinencode umformatiert, wodurch ein weiterer enormer Performance-Schub garantiert wird.

2.6.1 Unity DOTS - ECS - Components

Die *Components* sind die Grundlage der datenorientierten Programmierung, da sie für jedes *Entity* individuelle Daten speichern. Diese Daten werden mit sogenannten *Blitfähigen* Datentypen gespeichert, dadurch erhalten sie im verwalteten sowie im unverwalteten Speicher eine identische Präsentation, wodurch sie direkt gemeinsam genutzt werden können[22].
Entities erhalten bei der Generierung eigene *Components*, diese zeichnen ein *Entity* individuell aus, wodurch passende *Systems* auf ihnen agieren können.
Components können neben der Wiedererkennung von *Entities* sowie der Speicherung von Daten auch für die Separation von einzelnen *Entity* Gruppen verwendet werden. Indem man während der Ausführung eine *Component* von einem *Entity* entfernt oder hinzufügt, schränkt man andere *Systems* für die Ausführung auf diesen *Entities* ein, *Components* können auf *Entities* also auch während der Ausführung hinzugefügt bzw. entfernt werden. Sobald *Components* einmal definiert worden sind, können sie während der Ausführung nicht mehr verändert werden.
Unity selbst hat bereits einige *Components* vordefiniert, diese sind bereits in der *Entity* Erweiterung integriert und können jederzeit verwendet werden.

2.6.2 Unity DOTS - ECS - Systems

Systems sind der wichtigste Bestandteil im *ECS*. Sie verarbeiten die Daten in den *Components* der *Entities*. Dabei kann genau festgelegt werden, auf welchen *Entities* die *Systems* sich fokussieren müssen, um einen gewollten Effekt zu erzielen.
Viele *Systems* sind dabei auf bestimmte Daten aus den einzelnen *Components* angewiesen, weshalb es passieren kann, dass *Systems* immer wieder neu ausgeführt werden weil sie auf eine bestimmte Datenveränderung einer *Component* warten.
Damit die einzelnen *Systems* so leistungsstark wie möglich arbeiten, wird das *C#-Jobsystem* mit eingebunden. Durch die Verwendung von sogenannten *Jobs*, kann *System* code effizienter ausgeführt werden, indem er auf verschiedene *Worker-threads* verteilt wird. Normalerweise würde man hier einige kritische Wettlaufsituationen erwarten, da viele verschiedene *Systems* voneinander abhängig sind, im Fall von *Unity DOTS* allerdings, übernimmt *Unity* diesen Abschnitt für den Entwickler, dabei kümmert sich *Unity* um die korrekte Einreihung der einzelnen *Jobs* der *Systems*.

Beispielszenario:
System 1 und *System* 2 arbeiten beide mit *Entities* welche *Component* X besitzen. *System* 1 greift dabei mit einem schreibenden Zugriff auf den *Blitfähigen integer Datentypen* A der X *Component* zu und erhöht diesen jeweils um 1. *System* 2 greift ebenfalls auf A der X *Component* zu, allerdings benötigt *System* 2 nur lesenden Zugriff, da *System* 2 nur überprüft, ob A bereits größer als 100.000 ist, erst dann kann *System* 2 mit anderen Aufgaben fortfahren.
Dieses Szenario zeigt sehr gut, wie das *ECS* funktioniert. Viele einzelne *Systems* sind indirekt voneinander abhängig, da sie ständig auf veränderte Daten warten, welche sie für ihre Berechnungen benötigen.

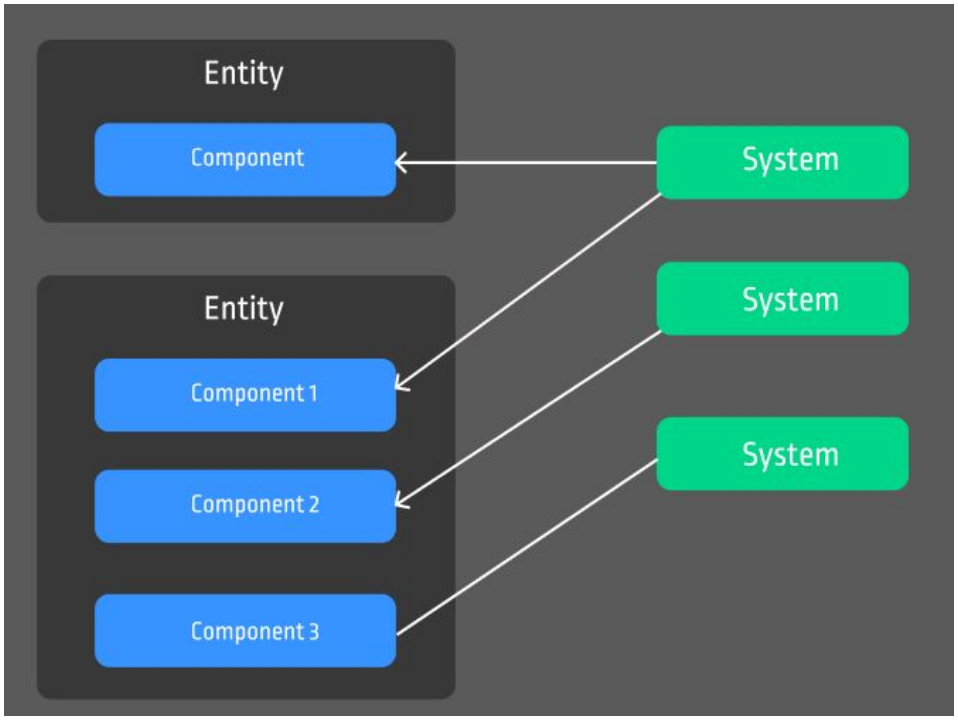


Abbildung 2.4: *Entity-Component-System* (*ECS*) - Verarbeitung von Daten durch *Systems* [47]

Abbildung 2.4 verdeutlicht den Ablauf eines *ECS* erneut. Während zwei *Entities* als Träger von *Components* existieren, arbeiten drei *Systems* mit diesen *Components*. Dabei wird auch deutlich, dass *Systems* selbstverständlich auch auf mehreren *Components* gleichzeitig arbeiten können. In diesem Beispiel hat sich das oberste *System* auf die *Component*s “Component” sowie “Component1” spezialisiert, weshalb es auch nicht von Bedeutung ist, dass sich beide *Components* auf unterschiedlichen *Entities* befinden.

2.6.3 Unity DOTS - C#-Jobsystem

Das *C#-Jobsystem* ist die *Unity DOTS* Komponente, welche den entwickelten *ECS* code aufgreift und mit Hilfe von so genannten *Jobs* auf den *Worker-threads* des Prozessors verteilt. Das *C#-Jobsystem* agiert dabei komplett autark, da die *Jobs* automatisch abgearbeitet und zwischen verschiedenen *Worker-threads* übergeben werden, um die bestmöglichste Performanz zu erzielen. Das *C#-Jobsystem* agiert dabei vom *Main-thread*, dort werden die verschiedenen *Jobs* erstellt und verwaltet.
Für die Entwicklung von Projekten, welche auf *Unity DOTS* basieren, muss der Entwickler lediglich die passenden *Jobs* definieren und dem *C#-Jobsystem* übergeben. Dieses übernimmt daraufhin die automatische nebenläufige Verarbeitung auf dem Prozessor.

2.6.4 Unity DOTS - Burst Compiler

Der *Burst-Compiler* "rundet" das Ergebnis der allgemeinen Performanz ab. Der von *Unity* entwickelte *Compiler* operiert mit den grundlegendsten *C#-Strukturen*, um den vorhandenen entwickelten code in performanten Maschinencode zu übersetzen. Dabei arbeitet der *Burst-Compiler* effizienter als andere *Compiler*, da dieser auf die speziellen Bedürfnisse der anderen *Unity DOTS* Komponenten angepasst ist. Während der Entwicklung stellt man allerdings schnell fest, dass der entwickelte code stark an den *Burst-Compiler* angepasst werden muss, da dieser wie bereits erwähnt, nur mit den grundlegendsten *C#-Strukturen* arbeitet. Beispielsweise können innerhalb von einem *Job*, welcher von *Burst* kompiliert wird, keine Operationen über einen *CommandBuffer* (wird später erläutert) durchgeführt werden, da dieser mit nicht grundlegendsten *C#-Strukturen* arbeitet.

2.7 In Unity entwickelte Massenpaniksimulationen

Dr. Roland Geraerts entwickelte mit seinen Studierenden an der Utrecht Universität in den Niederlanden eine Massenpaniksimulation, welche sich auf eine Notsituation innerhalb einer Stadt bezieht. Wie in Abbildung 2.5 zu erkennen ist, konzentrierte sich das Team nicht nur auf eine realistische Reaktion der einzelnen Charaktere, sondern auch auf das äußerliche der Simulation selbst. Für die Massenpaniksimulation wurden deswegen höher auflösende Grafiken genutzt.
Laut Geraerts ist es mit einem 1500€ PC möglich 15.000 *Agents* zu simulieren, ein 6.000€ PC mit 24 Kernen soll bis zu 65.000 *Agents* generieren können. Die *Agents* sind dabei in der Lage, realistisch mit ihrer Umwelt zu interagieren. Beispielsweise folgen sie bestimmten zielen und versuchen dabei, Kollisionen mit anderen *Agents* zu vermeiden.

Auf dynamische Änderungen der Umwelt reagieren die umliegenden *Agents* in Echtzeit, dabei übertragen sie ihr Verhalten an die jeweiligen *Agents* um sie herum. Abbildung 2.5 zeigt eine Explosion innerhalb eines Stadtteils, diese sorgt für eine realistische Panikreaktion der Charaktere um sie herum. Zu beobachten ist eine übertragung der Panik an die umliegenden Charaktere sowie eine Anstauung an verschiedenen Punkten der Stadt, dabei berechnen die Charaktere einen geeigneten Umweg.
Die Simulation selbst wurde in *C++* entwickelt, im laufe der Zeit wurde sie aber von Studierenden der Utrecht Universität in ein *Unity* plugin konvertiert, welches über die hochschuleigene Internetseite erworben werden kann, eine Veröffentlichung im *Unity Asset Store* steht noch aus[14].

Abbildung 2.5: Massenpaniksimulation der Utrecht Universität - Dr. Roland Geraerts [48]

Fazit

Die hier gezeigte Simulation vereinigt praktische sowie äußerliche Realität. Obwohl die Simulation bezüglich der Grafischen Realität deutlich anspruchsvoller ist, gelingt es dem Team der Utrecht Universität in den Niederlanden, die Anzahl der Charaktere deutlich zu erhöhen. Das Problem, welches aus diesen Resultaten absehbar ist, bezieht sich auf die Leistung der benötigten Hardware. Wie bereits erwähnt, wird ein 6.000€ PC benötigt, um 65.000 Charaktere in Echtzeit darzustellen. Zudem wird selbst für die kleinere Variante (15.000 Charaktere) ein Rechner benötigt der mindestens einen Wert von 1500€ aufweist. Für die Alltagsnutzung ist diese Simulation also weniger geeignet, da nicht jeder Benutzer diese Hardware beziehungsweise Rechenleistung vorweisen kann.

3. Konzept

In diesem Kapitel wird die Konzeptionierung einer Massenpaniksimulation, welche auf *Unity DOTS* basiert, erläutert. Die Simulation soll sich dabei nach dem Benutzer richten. Dieser soll die Möglichkeit haben, die Anzahl der zu simulierenden Charaktere zu bestimmen. Danach kann er verschiedene Ausgänge auf dem Gelände platzieren und mit Hilfe von verschiedenen “Panikoptionen” eine Panikreaktion in der Menschenmenge hervorrufen. Die Charaktere sollen ab diesem Zeitpunkt selbstständig agieren und das Gelände durch die vom Benutzer platzierten Ausgänge verlassen.

3.1 Plattformauswahl

Die Spiel-Engine *Unity3D* besitzt die Fähigkeit, Projekte auf verschiedene Plattformen zu konvertieren. Beispielsweise ist es möglich, ein Projekt zuerst für den PC zu entwickeln, um dieses zu einem späteren Zeitpunkt zu einem mobilen Projekt zu konvertieren.
Die Massenpaniksimulation, welche in dieser Arbeit umgesetzt wird, soll eine Standalone-PC Version repräsentieren. Diese Entscheidung bezieht sich hauptsächlich auf die Auswahl von *Unity DOTS*, welche als Grundlage dienen soll. *Unity DOTS* besitzt dabei als Teilkomponente das *C#-Jobsystem*. Diese Komponente nutzt die Anzahl der Kerne sowie die allgemeine Leistung des Prozessors. Damit sich die Massenpaniksimulation so leistungsstark wie möglich verhält, wird hierfür der PC als Hauptplattform ausgewählt.

3.2 Konzeptionierung der Umwelt

Auch wenn der Fokus während der Arbeit auf der Implementierung der Menschenmenge und der Panikreaktion sowie alle darum existierenden Ereignisse liegt, soll zusätzlich eine realistische Umgebung für die Simulation geschaffen werden. Die Umwelt des Projektes soll dabei vom Benutzer beeinflussbar sein, der Benutzer erhält also die Möglichkeit, mit der Umwelt der Simulation zu interagieren.
Das Hauptaugenmerk soll dabei auf dem Entwurf eines Konzert/Festivalgeländes liegen.

3.3 Konzeptionierung der Benutzeroberfläche

Um dem Benutzer verschiedene Informationen und Bedienelemente zugänglich zu machen, wird für diese Simulation eine geeignete Benutzeroberfläche benötigt. Diese soll dabei aus mehreren Sektionen bestehen und sich sowohl dynamisch als auch statisch verhalten. Dynamische Sektionen der Benutzeroberfläche sollen den Benutzer über die aktuellen Metadaten der Simulation informieren. Die Metadaten beschreiben dabei verschiedene numerische statistische Werte der Simulation.
Der Benutzeroberfläche soll eine statische Sektion hinzugefügt werden, welche dem Benutzer die aktuelle Steuerung der Simulation beschreibt. Zudem werden weitere statische Sektionen benötigt, welche als Bedienelemente dienen sollen. Diese sollen den Benutzer dazu befähigen, verschiedene Daten und Interaktionen der Simulation zu übermitteln.
Für den reibungslosen Ablauf einer Standalone-Simulation, soll der Benutzer in der Lage sein, die vollständige Simulation über die Benutzeroberfläche neu zu starten. Hierfür soll ein weiteres statisches Bedienelement hinzugefügt werden, welches diese Aufgabe übernehmen soll.

Der Benutzer soll in der Lage sein, verschiedene Aktionen auf dem Gelände selbst auszuführen. Diese sollen unmittelbar nach der Platzierung Einfluss auf die Konzertbesucher nehmen. Für diese Aktionen soll eine geeignete Benutzeroberfläche entstehen, welche sich dynamisch verhält und an der Position des Mauscursors erscheinen soll, sobald der Benutzer dieses Menü aufruft. Der Benutzer hat in diesem Menü die Möglichkeit, eine passende Aktion nach seinen Bedürfnissen auszuwählen, um diese im Anschluss zu platzieren. Das Menü soll sich nach der Auswahl automatisch selbst deaktivieren, die ausgewählte Aktion soll in den Metadaten der Simulation integriert werden, sodass diese dem Benutzer angezeigt wird.

3.4 ECS Designentscheidungen

Das folgende Kapitel präsentiert den allgemeinen Aufbau des *ECS*. Dabei wird die Konvertierung von sowohl den Konzertbesuchern als auch den Ausgängen aufgegriffen und dargestellt. Zudem werden die Abhängigkeiten der einzelnen *Components* sowie der *Systems* dargestellt und die Nachteile des *ECS* kurz erläutert.

3.4.1 Konzertbesucher als Entities

Die Konzertbesucher, welche sich auf dem Konzertgelände befinden, sollen als *Entities* dargestellt werden. Jede dieser *Entities* ist dabei Träger von verschiedenen *Components*.

Um ein realistisches Verhalten der Konzertbesucher zu simulieren, benötigen diese neben den *Unity* eigenen *Components* noch weitere *Components*, welche zusätzliches Verhalten ermöglichen. Diese werden in Kapitel 3.4.4 sowie in Kapitel 4.3.1 näher vorgestellt.
Damit sich die Konzertbesucher so realitätsnah wie möglich verhalten, benötigen diese verschiedene Verhaltensmuster, welche im laufe der Simulation zufällig ausgewählt werden sollen. In der Hauptkomponente der Konzertbesucher *Entities* soll also ein Konstrukt implementiert werden, welches Verhaltensmuster wie beispielsweise *gehen*, *rennen*, *tanzen* und *stehen* definiert und speichert.

3.4.2 Ausgänge als Entitäten

Der Benutzer hat die Möglichkeit, neben den Konzertbesuchern noch beliebig viele Ausgänge zu platzieren. Da die Konzertbesucher als *Entities* definiert sind, sind diese nur in der Lage, mit anderen *Entities* zu interagieren. Aus diesem Grund müssen zusätzlich zu den Konzertbesuchern, die Ausgänge als *Entities* dargestellt werden.
Die *Components* und *Systems*, welche sich auf die Ausgänge beziehen, werden in Kapitel 3.4.4 sowie in Kapitel 4.3.1 und in 4.3.2 näher vorgestellt.

3.4.3 Spawner Entität als Ausgangspunkt der Simulation

Verschiedene Recherchen ergaben, dass es einen Ausgangspunkt für das *ECS* geben muss. Dieses wird mit Hilfe eines gewöhnlichen *GameObjects* erzeugt, welches ein selbst definiertes “Ausgangsscript” besitzt und beim Start der Simulation in ein *Entity* konvertiert wird. Dieses “Ausgangsscript” dient dabei als “Brücke” zwischen der objektorientierten und der datenorientierten “Welt”. Beispielsweise werden hier Variablen, welche in der *Unity* eigenen objektorientierten “Welt” erzeugt werden, in die datenorientierte “Welt” übertragen. Als Beispiel kann man hier die Anzahl der *Entities* betrachten. Der Benutzer kann an diesem Punkt eine Anzahl festlegen, diese wird dann in die datenorientierte “Welt” übertragen und dort anhand von verschiedenen *Systems* verarbeitet.
Die *Components* und *Systems*, welche sich auf den Ausgangspunkt der Simulation beziehen, werden in Kapitel 3.4.4 sowie in Kapitel 4.3.1 und in 4.3.2 näher vorgestellt.

3.4.4 Komponenten und Systeme

Die verschiedenen *Components* und *Systems*, welche in dieser Arbeit umgesetzt werden sollen, beziehen sich auf verschiedene Bereiche der Simulation. Die *Components* agieren dabei als Datenträger dieser Bereiche, die *Systems* arbeiten mit diesen Daten.

Wie in Abbildung 3.1 zu sehen, wird jede *Component* von einem *System* aufgegriffen. Die *Systems* selbst stehen untereinander allerdings auch in Beziehung.

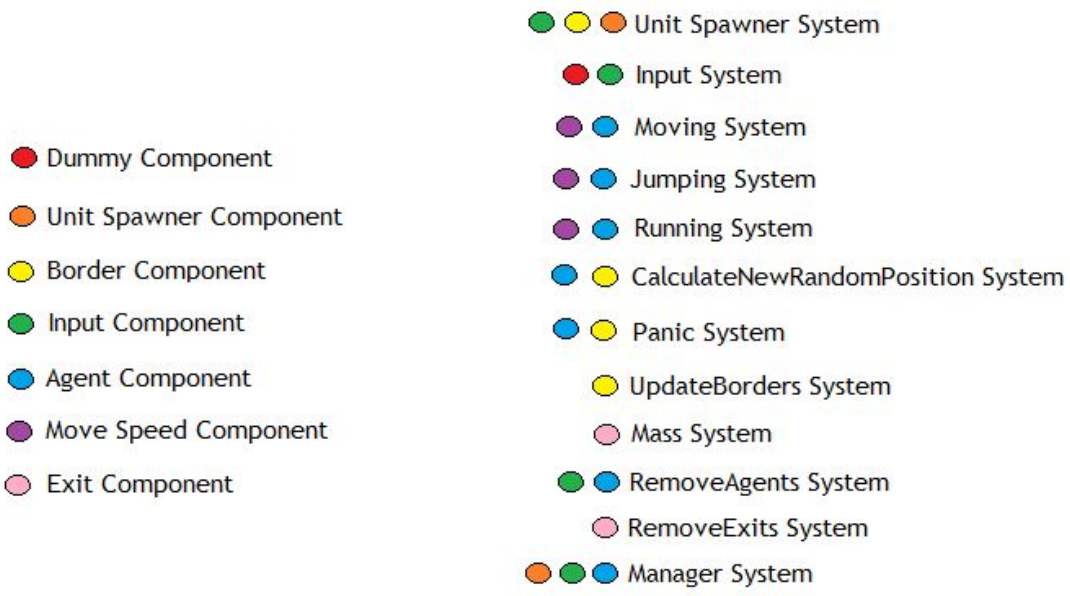


Abbildung 3.1: Abhängigkeiten von *Components* und *Systems*

3.5 Nachteile des ECS

Mehrfachnutzung von *Components*

Durch die mehrfache Nutzung von einzelnen *Components* durch unterschiedliche *Systems*(wie in Abbildung 3.1 zu sehen), wirkt ein *ECS* schnell unübersichtlich. Beispielsweise wird die *AgentComponent* in sieben verschiedenen *Systems* verwendet, da diese den jeweiligen Zugriff auf die Attribute der Konzertbesucher ermöglicht. Die *AgentComponent* ist somit in vielen *Systems* unersetzbar.

Iterationskosten

Die verschiedenen *Systems* und die dort enthaltenen *Jobs* besitzen alle eine individuelle Größe. Da sich die meisten von diesen *Jobs* auf die allgemeine Anzahl der aktuellen Konzertbesucher beziehen (der aktuelle *Job* muss über jeden Konzertbesucher ausgeführt werden), kann die Leistung der *Systems* stark variieren. Die Anzahl der Konzertbesucher beeinträchtigt also auch logischerweise die Performanz der *Systems* aus dem *ECS*.

Die Leistung des *ECS Systems* wird in Kapitel 6.1 kurz aufgegriffen.

4. Umsetzung

Das folgende Kapitel beschreibt die eigentliche Implementierung der Massenpaniksimulation. Dabei wird zuerst die Entwicklung des eigentlichen Konzertgeländes sowie der Benutzeroberfläche beschrieben, gefolgt von der Implementierung der einzelnen *Unity DOTS* Bestandteile in Form von dem *ECS*, welches durch das *C#-Jobsystem* und den *Burst-Compiler* unterstützt wird.
Das *ECS* wird in diesem Kapitel wiederum in seinen einzelnen Bestandteilen (in Form von *Components* und *Systems*) beschrieben.

4.1 Das Konzertgelände

Mit Hilfe von verschiedenen *Assets* aus dem Unity eigenen "Asset Store" wurde das Gelände so realistisch wie möglich "gebaut", dabei wurden zusätzliche *GameObjects* in die Seitenteile eingebaut, diese beinhalten keine zusätzlichen Objekte, sie dienen lediglich dazu ihre aktuelle Position zu speichern um die aktuellen Maße des Geländes zu erhalten. Diese werden im späteren Teil des Projektes benötigt, um die Konzertbesucher zu generieren. [35,16,13,21,4,17,25,24,27]

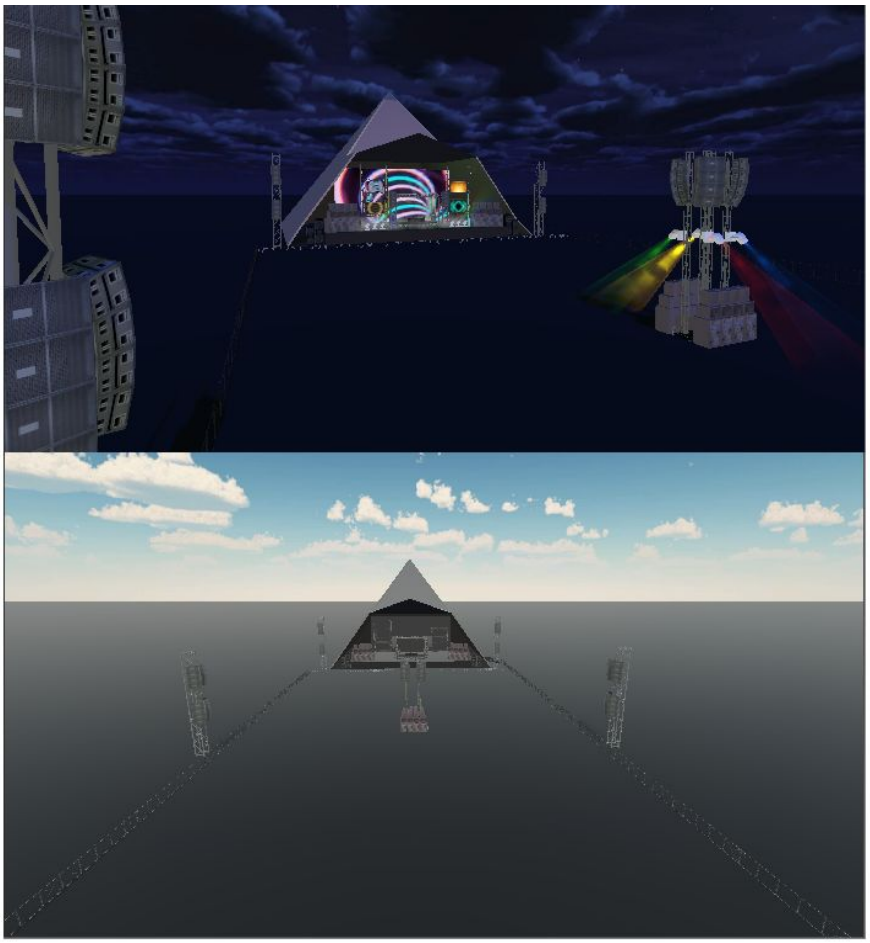


Abbildung 4.1: Das Konzertgelände - Oben: Nachtmodus + Effekte AN, Unten: Nachtmodus + Effekte AUS

Abbildung 4.1 visualisiert das Konzertgelände unter jeweils anderen Bedingungen. Das obere Abbild präsentiert das Gelände bei Nacht sowie mit aktivierten Effekten. Das untere Abbild hingegen visualisiert die Standardsimulation bei Tag und ohne Effekte.

4.1.1 Action Skripte

Die Action Skripte sorgen für einen reibungslosen Ablauf der platzierten Aktionen des Benutzers. Für dieses Vorhaben wurden vier Action Skripte realisiert, welche die Simulation realistischer wirken lassen.

4.1.1.1 Information Animation Truss - Action Skript

Das *InformationAnimationTruss* Skript beinhaltet die Methode *EnableDisableArrows()*. Diese Methode wird 60 mal pro Sekunde aufgerufen und aktiviert bzw. deaktiviert die Pfeile über den Stahlträgern, sobald eine Aktion vom Benutzer ausgewählt wurde, welche sich auf diese bezieht.[A42]

4.1.1.2 Information Animation Sound System - Action Skript

Das *InformationAnimationSoundSystem* Skript beinhaltet die Methode *EnableDisableArrows()*. Diese Methode wird 60 mal pro Sekunde aufgerufen und aktiviert bzw. deaktiviert die Pfeile über den Sound Systemen (größere Boxen auf dem mittleren Gelände), sobald eine Aktion vom Benutzer ausgewählt wurde, welche sich auf diese bezieht.[A43]

4.1.1.3 Action After Falling - Action Skript

Die Stahlträger besitzen die Funktion zu fallen, wenn der Benutzer sie anklickt (vorausgesetzt die richtige “Panikoption” wurde ausgewählt). Diese Funktion wurde mit Hilfe des *Unity* eigenen *Animator* realisiert, dieser verschiebt lediglich den entsprechenden Stahlträger auf der entsprechenden Achse.
Nachdem der Stahlträger die jeweilige Animation “abgespielt” hat, werden *Coroutinen* aus dem *ActionAfterFalling* Skript aufgerufen, welche die entsprechenden Explosionen sowie Feueranimationen generieren. Die entsprechenden Animationen werden nach einer bestimmten Zeit wieder entfernt.[A44]

4.1.1.4 Actions - Action Skript

Das *Actions* Skript reagiert auf die entsprechende Auswahl des Benutzers und generiert die entsprechende Animation. Beispielsweise ist in diesem Skript eine Methode zu finden, welche eine Explosion generiert, sobald der Benutzer mit der linken Maustaste auf das Konzertgelände klickt und die entsprechende Auswahl aus dem *Radial Menu* ausgewählt wurde.

Das *Actions* Skript enthält außerdem öffentliche *boolsche* Variablen, welche beschreiben, ob eine “Panikoption” vom Benutzer platziert wurde.
Der Zugriff erfolgt über die statische Instanzvariable *Actions.instance*.[A45]

4.2 Die Benutzeroberfläche

Die Benutzeroberfläche dient als allgemeine Schnittstelle zwischen Simulation und Benutzer. Der Benutzer kann hier verschiedene Informationen zur aktuellen Lage der Simulation entnehmen, sowie verschiedene Aktionen ausführen.
Die Benutzeroberfläche ist in verschiedene Panels aufgeteilt (zu sehen in Abbildung 4.2), wovon sich manche statisch, andere hingegen dynamisch verhalten.

Abbildung 4.2: Die Benutzeroberfläche der Massenpaniksimulation

4.2.1 Statistic Panel

Das *Statistic Panel* befindet sich in der oberen linken Ecke des Bildschirms und präsentiert die aktuellen Metadaten der Simulation. Die Metadaten bestehen dabei aus der aktuellen Anzahl der *Entities* und Ausgänge auf dem Gelände, sowie der aktuellen Bildfrequenz (*FPS*) und der Anzahl an Millisekunden (*ms*), welche die Verzögerung zwischen CPU (Prozessor) und GPU (Grafikprozessor) darstellt. Zusätzlich ist in den Metadaten noch die aktuell ausgewählte “Panikoption” enthalten, mit welcher der Nutzer auf dem Gelände Einfluss nehmen kann, dadurch weiß der Nutzer auch jederzeit, in welchem Modus er sich in diesem Moment befindet.

4.2.1.1 Aufbau

Das *Statistic Panel* besteht aus einem UI Panel. Dieses Panel enthält ein Textfeld, welches von der *Unity* Erweiterung "TextMeshPro" stammt[39] und seine Informationen aus dem *UIHandler* Skript bezieht.

4.2.1.2 Implementierung - UIHandler/ShowFPS()

Das Skript *UIHandler* besitzt eine Methode namens *ShowFPS()*, diese wird in der *Unity* eigenen Methode *Update()* 60 mal pro Sekunde aufgerufen. *ShowFPS()* nutzt dabei die sogenannte "deltaTime", um den aktuellen *FPS* und *ms* Wert zu berechnen. Die "deltaTime" beschreibt dabei den Zeitwert, welcher gebraucht wird, um den derzeitigen frame mit einem neuen frame zu ersetzen.
Die dadurch gewonnenen Daten werden über eine Objektreferenz an das aktuelle Panel übermittelt. Da die Methode *ShowFPS()* 60 mal pro Sekunde aufgerufen wird, wird immer der aktuelle *FPS*- und *ms*-Wert präsentiert[A1].

4.2.2 Information Panel

Direkt unter dem *Statistic Panel* befindet sich das *Information Panel*. Dieses dient lediglich dazu, dem Benutzer die aktuelle Steuerung der Simulation anzuzeigen. Mit Hilfe dieses Panels erlernt der Benutzer innerhalb von wenigen Sekunden, wie genau die Simulation funktioniert. Neben verschiedenen funktionellen Informationen ist außerdem noch der Hinweis enthalten, dass durch das drücken von *H* das *Information Panel* ausgeblendet werden kann. Diese Funktion ist dann sehr hilfreich, sobald die Simulation häufiger genutzt wird.
Außerdem wurde noch ein "Nachtmodus" implementiert, welcher das Gelände bei Nacht darstellt. Neben dem "Nachtmodus" existiert noch ein weiterer optionaler Modus, welcher einige Partikel und Lichter auf dem gesamten Gelände aktiviert bzw. deaktiviert. Falls der Benutzer die Seitenteile ausblenden möchte, kann er das mit dem "LOD" Modus tun, dadurch werden die Seitenteile ausgeblendet, sobald sich die Kamera nicht mehr in Sichtweite befindet.

4.2.2.1 Aufbau

Das *Information Panel* besteht aus einem *UI Panel*. Dieses Panel enthält ein Textfeld, welches von der *Unity* Erweiterung "TextMeshPro" stammt[39] und seine Informationen aus dem *Unity Inspector* selbst bezieht. Der *Unity Inspector* ist in der UI Oberfläche von *Unity* selbst zu finden.

4.2.2.2 Implementierung - UIHandler

Der *UIHandler* umfasst einige wichtige Methoden, welche für die Beeinflussung der Benutzeroberfläche sowie für die Umwelt essentiell sind. Diese werden im folgenden erläutert.

4.2.2.2.1 Implementierung CheckKeys()

Die Methode *CheckKeys()* wird selbst in der *Unity* eigenen *Update()* Methode, also 60 mal pro Sekunde aufgerufen. Bei jedem Aufruf kontrolliert *CheckKeys()* dabei, ob die Tasten *ESC*, *O*, *N* oder *L* betätigt wurden. Sollte dieser Fall eintreten, ruft *CheckKeys()* die entsprechende Methode auf, um die entsprechende Aktion auszuführen.[A9]
Tabelle 4.a erläutert diese Aktionen:

Taste	Bedeutung	Methode
ECS	Deaktiviert den aktuellen Modus. Wenn diese Taste gedrückt wird, ist kein Modus mehr ausgewählt	Über eine Instanzvariable wird der entsprechende Wert geändert.[A9]
O	Aktiviert oder deaktiviert das LOD Skript der beiden Seitenteile. Bei aktiviertem LOD Skript blenden sich die Seitenteile bei einer bestimmten Entfernung aus.	Die Methode *EnableOrDisableLODFunction()* wird aufgerufen, welche über eine objektreferenz den entsprechenden Wert ändert.[A12]
N	Aktiviert oder deaktiviert den Nachtmodus.	Die Methode *EnableOrDisableNightMode()* wird aufgerufen, welche die entsprechende Skybox aktiviert und die derzeitige Farbe des Lichtes in der Simulation ändert.[A13]

L	Aktiviert oder deaktiviert die optionalen Effekte in der Simulation.	Die Methode *EnableOrDisableEffects()* greift auf verschiedene Objektreferenzen zu und aktiviert bzw. deaktiviert diese.[A14]

Tabelle 4.a: Tastenbelegungen

4.2.2.2.2 Implementierung HandleCamera()

Die Methode *HandleCamera()* wird selbst in der *Unity* eigenen *Update()* Methode, also 60 mal pro Sekunde aufgerufen.
Die Methode *HandleCamera()* reagiert auf die Tasten *W*, *A*, *S*, *D* sowie dem *Mausrad* um daraufhin verschiedene Vektoroperationen auf die *Unity* eigene Kamera auszuführen. Dadurch ist der Benutzer in der Lage, sich in der Simulation mit Hilfe von seiner Tastatur und Maus zu bewegen.[A10]

4.2.2.2.3 Implementierung WindowCheck()

Die Methode *WindowCheck()* wird selbst in der *Unity* eigenen *Update()* Methode, also 60 mal pro Sekunde aufgerufen.
WindowCheck() reagiert auf die Tasten *I* und *H*. Sobald eine dieser beiden Tasten gedrückt wurde, wird das entsprechende Fenster aktiviert, bzw. deaktiviert.[A11]

4.2.3 Input Panel

Das *Input Panel* befindet sich in der unteren linken Ecke des Bildschirms und ist eine direkte Schnittstelle zwischen Benutzer und Simulation. Mit diesem Panel ist der Benutzer in der Lage, eine numerische Eingabe zu tätigen, welche von der Simulation verarbeitet wird. Die eingegebene Zahl beschreibt dabei die zu generierende Anzahl an Konzertbesuchern, diese wird erst verarbeitet, sobald der Nutzer auf den *Speichern* Knopf klickt. Alternativ kann der Nutzer nach der Eingabe auch auf eine der beiden *Enter* Tasten drücken um die Eingabe zu speichern.

4.2.3.1 Aufbau

Das *Input Panel* besteht aus einem *UI Panel*. Dieses Panel enthält ein Textfeld, welches von der *Unity* Erweiterung “TextMeshPro” stammt[39] und seine Informationen aus dem *Unity Inspector* selbst bezieht, die Information bezieht sich dabei auf die Darstellung “Anzahl Entities”. Der *Unity Inspector* ist in der UI Oberfläche von *Unity* selbst zu finden.

Neben dem Textfeld enthält das Panel noch ein *InputField*, welches die Benutzereingaben entgegen nimmt. Diese Eingaben werden von dem *InputWindow* Skript überprüft und verarbeitet.
Um die Eingaben zu speichern, wurde ein *UIButton* in das Panel integriert, welcher die Methode *SaveValue* aufruft sobald er angeklickt wurde.

4.2.3.2 Implementierung - InputWindow/ValidateInput(), CheckEnterKeyInput()

Die beiden Methoden *ValidateInput()* und *CheckEnterKeyInput()* werden permanent in der *Unity* eigenen *Update()* Methode in dem Skript *InputWindow* aufgerufen. Durch das aufrufen in *Update()* wird garantiert, dass die Methoden oft genug aufgerufen werden, damit sie rechtzeitig auf die Benutzereingaben reagieren können. Sobald der Benutzer eine Eingabe tätigt, überprüft die *ValidateInput()* Methode, ob die Eingabe einer numerischen Eingabe gleicht und ob das Limit der Zahlenlänge von sieben nicht überschritten wurde. Eine Eingabe wird auf dem Bildschirm also nur dann registriert, wenn *ValidateInput()* dauerhaft die Genehmigung dafür erteilt.
Die Methode *CheckEnterKeyInput()* wird direkt nach *ValidateInput()* aufgerufen. In *CheckEnterKeyInput()* werden die beiden verfügbaren *Enter* Tasten geprüft. Sobald eine *Enter* Eingabe getätigt wurde, wird die Methode *SaveValue()* aufgerufen, welche auch aufgerufen wird, sobald der *Speichern* Knopf angeklickt wurde[A2,A3].

4.2.3.3 Implementierung - InputWindow/SaveValue()

SaveValue() wird aufgerufen, sobald der *Speichern* Knopf angeklickt oder eine der beiden *Enter* Tasten betätigt wurde.
SaveValue() speichert daraufhin die Eingabe in der Instanzvariable *AmountToSpawn* aus dem *UnitSpawnerProxy* Skript, welches wie bereits erwähnt die “Brücke” der beiden Paradigma “Welten” darstellt[A4]. Dieses Skript repräsentiert den “Startpunkt” des datenorientierten Abschnitts dieses Projektes, weswegen ein Zugriff auf die Anzahl der zu generierenden *Entities* an dieser Stelle von großer Bedeutung ist.

4.2.3.4 Anmerkung - Input Panel

Eingaben, welche der Nutzer in diesem Panel tätigen kann, werden mit Hilfe der numerischen Tasten auf der Tastatur getätigt. Diese Tasten haben allerdings noch weitere Funktionen. Aus diesem Grund wurde eine “boolsche” Variable erstellt, welche jederzeit angibt, ob das *InputField* fokussiert ist oder nicht. Diese Variable ist eine statische Instanzvariable, wodurch sie in anderen Skripten aufgerufen und überprüft werden kann. Viele Methoden verwenden sie als Bedingung, um eine Aktion auszuführen. Dadurch wird verhindert, dass zum Beispiel der Nutzer frühzeitig *Entities* mit der Taste *eins* generiert, obwohl er nur 10.000 in das *InputField* schreiben wollte.

4.2.4 Slot Panel

Das *Slot Panel* ist eine visuelle Darstellung der verschiedenen Aktionen die das äußerliche des Geländes verändern können, es befindet sich am unteren Bildschirmrand. Die Aktionen selbst werden mit Hilfe der Tasten 1-7 auf der oberen Zahlenreihe der Tastatur ausgelöst.
Folgende Aktionen wurden implementiert (Tabelle 4.b):

Taste	Aktion	Kommentar
1	Generierung der Entities	Die zuletzt abgespeicherte Zahl des *InputPanels* wird für die Generierung verwendet.
2	Löschung der Entities	Alle Konzertbesucher werden entfernt.
3	Rotation der Seitenteile nach außen	Um die Seitenteile nach außen zu drehen, muss die Taste nach unten gehalten werden.
4	Rotation der Seitenteile nach innen	Um die Seitenteile nach innen zu drehen, muss die Taste nach unten gehalten werden.
5	Hinzufügen eines Seitenteils	Das Limit liegt hier bei maximal 5 Seitenteilen.
6	Entfernen eines Seitenteils	Das Limit liegt hier bei mindestens einem Seitenteil, welches platziert sein muss.
7	Entfernen der Ausgänge	Entfernt alle visuellen Ausgänge auf dem Gelände sowie alle Entities in Echtzeit.

Tabelle 4.b: Tastenbelegungen (obere Zahlenreihe)

4.2.4.1 Aufbau

Das *Slot Panel* besteht aus einem *UI Panel*. Dieses *UI Panel* enthält eine *Horizontal Layout Group*, welche *GameObjects* automatisch horizontal in einem angemessenem Abstand anordnet.
Die hier verwendeten *GameObjects* beinhalten mehrere verschachtelte *ImageViews*, welche aufeinander aufbauen und verschiedene Ebenen abbilden. Die Hauptebene stellt dabei das zu sehende Bild des Slots dar. Insgesamt wurden sieben dieser *GameObjects* in die *Horizontal Layout Group* integriert.
Neben den einzelnen Slots befinden sich dort noch kleinere Zahlen, welche mit Hilfe von Textfeldern dargestellt werden. Dadurch weiß der Nutzer direkt, welche Taste für diese Funktion gedrückt werden muss.
Damit die einzelnen Slots auch eine Aktion ausführen, wurde ein Skript erstellt, welches auf diese reagiert.

4.2.4.2 Implementierung ItemClickHandler/Tasten 3-6

Der *ItemClickHandler* kümmert sich hauptsächlich um die Tasten 3-6. Für die Tasten *eins* und *zwei* werden lediglich kleinere Anpassungen der Metadaten für das *Statistic Panel* übermittelt. Dadurch wird sichergestellt, dass immer die aktuelle Anzahl an *Entities* im *Statistic Panel* angezeigt wird. Außerdem wird die Anzahl zurückgesetzt, sobald die Taste *zwei* gedrückt wird.

4.2.4.2.1 Implementierung OnButtonHoldingDownRotateOut()

OnButtonHoldingDownRotateOut() wird aufgerufen, sobald der Nutzer die Taste *drei* nach unten gedrückt hat.
In der Methode selbst wird der derzeitige *y*-Wert der beiden Seitenteile überprüft. Sobald dieser für die linke Seite kleiner als 0.5 und für die rechte Seite größer als -0.5 ist, werden beide Seitenteile mit einer bestimmten *rotationSpeed* nach außen rotiert.
Die letzten beiden Sound Systeme auf dem Gelände werden bei einer bestimmten Rotation der Seitenteile ein-, beziehungsweise ausgeblendet, da die *Entities* nur bis zu einem bestimmten Punkt des Geländes generiert werden. Die Sound Systeme würden dabei “hinausragen”[A5].

4.2.4.2.2 Implementierung OnButtonHoldingDownRotateIn()

OnButtonHoldingDownRotateIn() wird aufgerufen, sobald der Nutzer die Taste *vier* nach unten gedrückt hat.
In der Methode selbst wird der derzeitige *y*-Wert der beiden Seitenteile überprüft. Sobald dieser für die linke Seite kleiner als -0.5 und für die rechte Seite größer als 0.5 ist, werden beide Seitenteile mit einer bestimmten *rotationSpeed* nach innen rotiert. Die letzten beiden Sound Systeme auf dem Gelände werden bei einer bestimmten Rotation der Seitenteile ein-, bzw. ausgeblendet, da die *Entities* nur bis zu einem bestimmten Punkt des Geländes generiert werden. Die Sound Systeme würden dabei "hinausragen"[A6].

4.2.4.2.3 Implementierung AddNewSideBarrier()

AddNewSideBarrier() wird aufgerufen, sobald der Nutzer die Taste *fünf* nach unten gedrückt hat.
In der Methode selbst wird ein statischer *increaseCounter* überprüft, welcher die derzeitige Anzahl der Seitenteile enthält. Dadurch kann überprüft werden, ob ein weiteres Seitenteil platziert werden darf oder nicht.
Wenn die Bedingung für das platzieren erfüllt ist, also der *increaseCounter* kleiner als fünf ist, wird ein weiteres Seitenteil platziert.
Anzumerken ist hierbei noch, dass die Seitenteile nicht neu platziert werden, sondern nur "freigeschaltet" werden. Um Leistung zu sparen, werden die Seitenteile einmal zu Beginn der Simulation generiert und direkt deaktiviert. Nach belieben werden diese dann vom Nutzer nacheinander aktiviert beziehungsweise deaktiviert. Dadurch wird verhindert, dass die Seitenteile immer wieder neu generiert werden müssen[A7].

4.2.4.2.4 Implementierung RemoveNewSideBarrier()

RemoveNewSideBarrier() wird aufgerufen, sobald der Nutzer die Taste *sechs* nach unten gedrückt hat.
In der Methode selbst wird ein statischer *increaseCounter* überprüft, welcher die derzeitige Anzahl der Seitenteile enthält. Dadurch kann überprüft werden, ob ein weiteres Seitenteil entfernt werden darf oder nicht.
Wenn die Bedingung für das entfernen erfüllt ist, also der *increaseCounter* größer als eins ist, wird ein weiteres Seitenteil entfernt.
Anzumerken ist hierbei noch, dass die Seitenteile nicht direkt entfernt werden, sondern nur "deaktiviert" werden. Um Leistung zu sparen, werden die Seitenteile einmal zu Beginn der Simulation generiert und direkt deaktiviert. Nach belieben werden diese dann vom Nutzer nacheinander aktiviert bzw. deaktiviert. Dadurch wird verhindert, dass die Seitenteile immer wieder neu generiert werden müssen[A8].

4.2.5 Reload Scene Panel

Das *ReloadScenePanel* ist gleichzeitig auch ein *Unity Button*, es befindet sich in der unteren rechten Ecke des Bildschirms. Sobald man diesen *Button* anklickt, wird die gesamte *Szene* neu geladen. Alle Einstellungen werden dabei verworfen. Der Nutzer erhält dadurch die Möglichkeit, seine Einstellungen zu verwerfen und von vorne zu beginnen.

4.2.5.1 Aufbau

Das *ReloadScenePanel* besteht aus einem *Unity Button*. Sobald dieser *Button* betätigt wird, wird die Methode *ReloadActualScene()* aus dem *ReloadScene* Skript aufgerufen.

4.2.5.2 Implementierung - ReloadScene/ReloadActualScene()

Damit ein vollständiger Neustart der gesamten Szene garantiert werden kann, müssen neben den visuellen Aspekten der Simulation, auch alle Hintergrundprozesse zurückgesetzt werden.
Die Methode *ReloadActualScene()* ruft deswegen zuerst die Methode *DestroyAllEntities()* auf, diese greift auf den *EntityManager* der momentanen "Weltinstanz" zu und zerstört über *DestroyEntity()* jede *Entity* auf dem Gelände. Als Eingabeparameter wird eine sogenannte *Query* gefordert. Diese muss alle *Entities* enthalten, welche entfernt werden sollen. Über den zuvor erstellten *EntityManager* wird deswegen eine "UniversalQuery" abgefragt, welche jede *Entity* in der Simulation enthält. Diese "UniversalQuery" wird als Parameter übergeben, dadurch wird jede *Entity* entfernt[A16].
Nachdem die Methode *DestroyAllEntities()* aufgerufen wurde, wird eine sogenannte *Coroutine()* namens *DisposeAllWorlds()* aufgerufen. Diese *Coroutine()* wartet bis zum Ende der Lebenszeit des aktuellen frames, um dann die aktuelle "Welt" zu zerstören[A17].
Sobald diese "Welt" zerstört worden ist, wird über den *SceneManager* die Methode *LoadScene()* aufgerufen, dort wird der aktuelle *buildIndex* übergeben, welcher der zuvor zerstörten Szene zugeordnet ist. Dadurch wird die zuvor zerstörte Szene erneut geladen[A15]. Die Simulation wurde somit neu gestartet.

4.2.6 Status Panel

Das *Status Panel* beinhaltet ein *Unity Image* welches den derzeitigen Status der *Entities* anzeigt. Sobald diese in "Panik verfallen", wird dieser Status in diesem Panel angezeigt. Das *Status Panel* befindet sich in der oberen rechten Ecke des Bildschirms.

4.2.6.1 Aufbau

Das *Status Panel* besteht aus einem *Unity Image*. Die Methode *UpdateStatusIcon()* aus dem *UIHandler* sorgt dafür, dass dieses Image jederzeit dem aktuellen Status der *Entities* entspricht.

4.2.6.2 Implementierung - UIHandler/UpdateStatusIcon()

Die Methode *UpdateStatusIcon()* wird in der *Unity* eigenen Methode *Update()* 60 mal pro Sekunde aufgerufen. Bei jedem Aufruf überprüft die Methode *UpdateStatusIcon()* dabei, ob eine "Panik auslösende" Aktion vom Benutzer platziert wurde. Das entsprechende *Sprite* wird dabei über die entsprechende Objektreferenz in das *Unity Image* geladen[A18].

4.2.7 Radiales Panik Menü

Damit der Benutzer jederzeit die Option hat, eine "Panikreaktion" mit Hilfe von verschiedenen "Optionen" zu generieren, wurde nach einem Weg gesucht, diese immer so schnell wie möglich zur Auswahl zu haben. Nach kurzer Überlegung hat sich ein *Radial Menu* angeboten[2]. Dieses wird beim betätigen einer Taste an der derzeitigen *Mauscursor* Position generiert (wie in Abbildung 4.3 gezeigt), wodurch der Nutzer es sofort in seinem Blickfeld hat. Da die Tastatur bereits eine der haupt-Eingabequellen für die Simulation ist, wurde die *TAB* Taste dafür ausgewählt.
Sobald der Nutzer also die *TAB* Taste gedrückt hält, erscheint an der derzeitigen Mauscursor Position ein *Radial Menu*. Aus diesem *Radial Menu* kann der Nutzer dann verschiedene *Optionen* auswählen und auf dem Gelände platzieren.

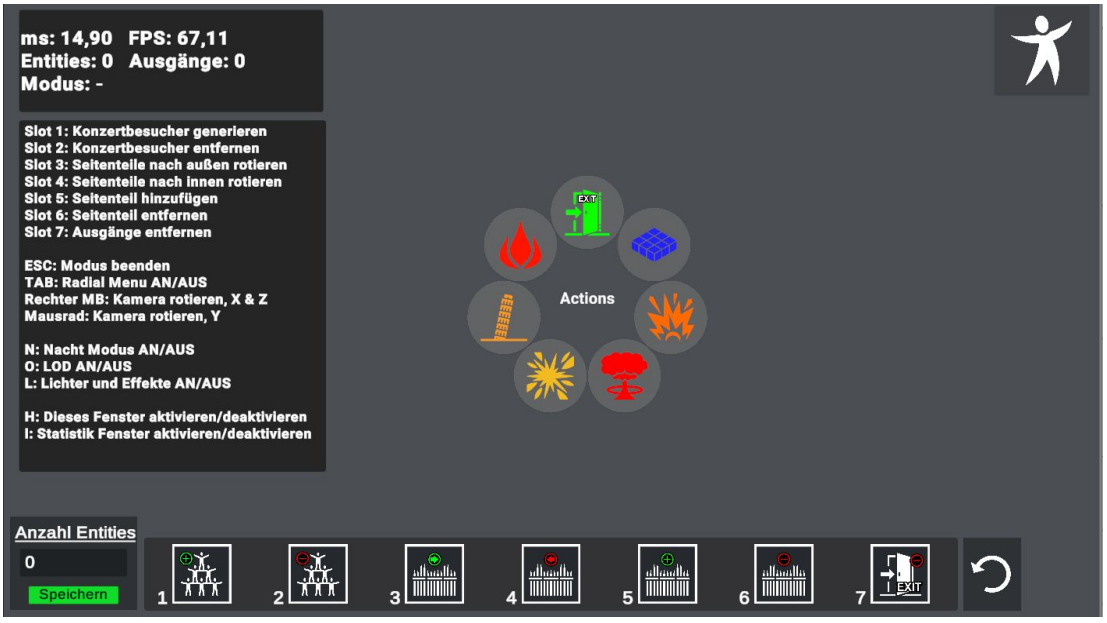


Abbildung 4.3: Radiales Panik Menü der Massenpaniksimulation bei Betätigung der *TAB*-Taste

4.2.7.1 Aufbau

Das *Radial Menu* besteht aus sieben verschiedenen Optionen, jede einzelne besteht aus einem einfachen *Sprite GameObject* mit einem individuellen Bild sowie verschiedenen Informationen wie einem Titel und einer Farbe, diese Informationen wurden im *Unity* eigenen *Inspector* vordefiniert abgespeichert. Das *Sprite GameObject* wird vordefiniert ohne Titel und Farbe als sogenanntes *Prefab* abgespeichert.
In der *Coroutine* "AnimateButtons" [A19] erhält man über eine *GameObject* Referenz Zugriff auf dieses *Prefab*, um die verschiedenen *Sprite GameObjects* zu generieren.
Über ein vordefiniertes *Unity Interface* wird die Information bezogen, ob sich der Nutzer mit der Maus über einer *Option* befindet oder nicht. In der Methode *CheckRadialMenuSelection()* aus dem *RadialMenu* Skript wird diese Information verarbeitet.
Folgende Optionen befinden sich in dem *Radial Menu* (Im Uhrzeigersinn)(Tabelle 4.c):

Option	Funktion	Besonderheiten/Anmerkungen
Ausgänge erstellen	Der Nutzer kann mit dieser Option verschiedene Ausgänge an den Seitenteilen erstellen.[15]	Bei dieser Option existiert kein Limit. Der Nutzer kann so viele Ausgänge erstellen wie gewünscht. Besucher fokussieren später diese Ausgänge.
Plattform erstellen	Der Nutzer kann mit dieser Option zusätzliche Sound Systeme erstellen.[21]	Diese Option existiert nur noch aus ästhetischen Gründen. Diese Sound Systeme verwendeten ein mal *Collider*, um die Besucher davon abzuhalten, durch diese Sound Systeme hindurch zu laufen. Diese Idee wurde aber später verworfen.
Mittlere Explosion	Mit dieser Option können Explosionen mit einer mittleren Intensität generiert werden.	In einer früheren Version waren die Besucher in der Lage die Explosion zu erkennen und diese Stelle zu meiden.

	Die Besucher verfallen dadurch in Panik.[30]	Diese Funktion wurde aber ebenfalls aus performance Gründen entfernt.
Große Explosion	Mit dieser Option können Explosionen mit einer hohen Intensität generiert werden. Die Besucher verfallen dadurch in Panik.[30]	In einer früheren Version waren die Besucher in der Lage die Explosion zu erkennen und diese Stelle zu meiden. Diese Funktion wurde aber ebenfalls aus performance Gründen entfernt.
Kleine Explosion	Mit dieser Option können Explosionen mit einer kleinen Intensität generiert werden. Die Besucher verfallen dadurch in Panik.[30]	In einer früheren Version waren die Besucher in der Lage die Explosion zu erkennen und diese Stelle zu meiden. Diese Funktion wurde aber ebenfalls aus performance Gründen entfernt.
Fallender Mast	Mit dieser Option kann der Nutzer jeden Mast auf dem Gelände anklicken, dieser fällt dann auf den Boden, erzeugt Explosionen und fängt Feuer. Die Besucher verfallen dadurch in Panik.[30,25]	Um Leistung zu sparen, wurden die generierten Feuer mit einem *timer* versehen. Dadurch bestehen diese Feuer nur einige Sekunden. Die Animation selbst wurde mit dem *Unity* eigenen *Animator* erstellt.
Feuer	Mit dieser Option kann der Nutzer Feuer auf Sound Systemen legen.[30]	Diese Option funktioniert auch auf Benutzer generierten Sound Systemen.

Tabelle 4.c: Panikoptionen

4.2.7.2 Implementierung - RadialMenu/SpawnButtons()

Die Methode *SpawnButtons()* wird aus dem Skript *UIHandler()* aufgerufen, sobald der Nutzer die *TAB* Taste gedrückt hält. *SpawnButtons()* ruft daraufhin die *Coroutine* "AnimateButtons()" auf. Diese *Coroutine* erhält Zugriff auf die im *Unity* eigenen *Inspector* vordefinierten Informationen der *Sprite GameObjects*. Die verschiedenen *Sprite GameObjects* werden generiert und gelangen mit Hilfe von dem aktuellen *Sprite GameObject* Index und der Berechnung des Umkreises an die richtige Stelle.
Sobald ein *Sprite GameObject* generiert und platziert wurde, werden zusätzliche Attribute wie die Farbe, Bild und Titel übergeben.
Um einen zusätzlichen "Generierungseffekt" zu erstellen, wurde eine *Coroutine* für die Generierung der *Sprite GameObjects* gewählt. Diese sorgt dafür, dass nach jeder Generierung von einem *Sprite GameObject* 0.07 Sekunden gewartet wird, bevor das nächste *Sprite GameObject* generiert wird. Dadurch erhält man einen "Verzögerungseffekt" bei der Generierung der *Sprite GameObjects*.[A19]

Für dieses Vorhaben hätte man auch den *Unity* eigenen *Animator* verwenden können, dieser hätte auch andere Animationen bereitgestellt. Aus performance Gründen wurde aber die *Coroutine* gewählt, da diese nur zwischen der Generierung warten muss, um einen Effekt zu erzielen.

4.2.7.3 Implementierung - RadialMenu/CheckRadialMenuSelection()

Jedes *Sprite GameObject* verwendet die *Unity* eigenen *Interfaces IPointerEnterHandler* und *IPointerExitHandler*. Sobald der Nutzer mit dem Mauscursor also über einen *Button* "fährt", können verschiedene Informationen präferiert werden. So existiert in der Klasse *RadialMenu* eine *Unity Button* Variable namens *selected*, diese wird, mit Hilfe von der Interfacemethode *IPointerEnterHandler.OnPointerEnter,* ständig mit dem aktuell "überfahrenen" Button ersetzt. Das gleiche gilt für die Interfacemethode *IPointerExitHandler.OnPointerExit*, hier wird *selected* auf *null* gesetzt. In den beiden Interfacemethoden werden außerdem noch zusätzliche Informationen wie Farbe und Titel geändert, diese ändern sich nämlich, wenn man mit dem Mauscursor über die einzelnen Buttons "fährt".

Die Methode *CheckRadialMenuSelection()* aus dem *RadialMenu* Skript wird in der *Unity* eigenen Methode *Update()*, also 60 mal pro Sekunde aufgerufen. Dadurch kann sie ständig überprüfen, ob sich *selected* geändert hat.
Je nach Auswahl, verändern sich verschiedene *boolsche* Variablen, dadurch erhalten die entsprechenden Methoden (Generierung von Explosionen, Feuer usw.) die Berechtigung, die Effekte zu platzieren.

An diesem Punkt wird auch der *String* gespeichert, welcher den aktuellen Modus in dem *Statistic Panel* repräsentiert[A20].

4.3 Die Generierung und gegenseitige Beeinflussung der Konzertbesucher durch Unity DOTS

Das Ziel in der Simulation ist die Skalierung der Konzertbesucher sowie deren realistische Reaktion bei einem “Panikereignis”, ausgelöst vom Benutzer der Simulation.
Unity DOTS , bestehend aus dem *ECS*, dem *C#-Jobsystem* sowie dem *Burst-Compiler*, stellt alle notwendigen Komponenten bereit um dieses Vorhaben zu realisieren.

Wiederholung
Das *ECS* löst dabei die objektorientierte Programmierung ab, um leistungsstärkeren code zu entwickeln, welcher sogar in anderen Projekten wiederverwendet werden kann. Das *ECS* besteht dabei aus sogenannten *Entities*, *Components* und *Systems*. Das Hauptprinzip in der datenorientierten Programmierung besteht in der Generierung von *Entities*, welche die Träger von *Components* sind, die *Systems* agieren auf diesen *Components.*
Das *C#-Jobsystem* greift den *ECS* code auf, um ihn in effizienten multi threaded code zu konvertieren, dabei werden die *Systems* des *ECS* durch sogenannte *Jobs* ergänzt, welche im laufe der Simulation durch den *Main-thread,* an die verschiedenen *Worker-threads* verteilt werden.
Der *Burst-Compiler* ergänzt das *ECS* und das *C#-Jobsystem* zusätzlich, indem er den entwickelten code in hoch performanten Maschinencode formatiert, wodurch ein weiterer enormer Performance-Schub garantiert wird.

4.3.1 Datenorientierte Programmierung (ECS) - Components

Neben den *Unity* eigenen *Components* wurden für diese Simulation zudem noch sieben zusätzliche *Components* erstellt, welche alle von dem *Interface IComponentData* erben. Diese werden im folgenden näher erläutert.

4.3.1.1 Unit Spawner Component

Das *crowd GameObject* (enthalten in der main-project-scene) wird der Träger dieser *Component* sein und stellt damit den "Beginn" des *ECS* Abschnitts dar. An dem *crowd GameObject* befindet sich ein vordefiniertes *Unity* Skript namens *ConvertToEntity*. Sobald die Simulation gestartet wird, konvertiert *Unity* das *crowd GameObject* in ein *Entity*. Dieses *Entity* wird der Träger von der *UnitSpawnerComponent* sein.

Die folgenden Blitfähigen Datenträger sind in der *UnitSpawnerComponent* enthalten[A21] (Tabelle 4.d):

Blitfähiger Datenträger	Bedeutung
int AmountToSpawn	Die Anzahl der Konzertbesucher welche generiert werden soll.
Entity Prefab	Eine erste Kopie eines Konzertbesuchers, diese Kopie wird im späteren Verlauf beliebig oft geklont.

Tabelle 4.d: Datenträger der *UnitSpawnerComponent*

4.3.1.2 Border Component

Damit jeder Konzertbesucher Zugriff auf die derzeitigen Maße des Konzertgeländes hat, benötigt jedes *Entity*, welches einen Konzertbesucher darstellt, eine *BorderComponent*. Diese beinhaltet Datenträger, welche die derzeitigen Maße des Konzertgeländes speichern.

Die folgenden Blitfähigen Datenträger sind in der *BorderComponent* enthalten[A22] (Tabelle 4.e):

Blitfähiger Datenträger	Bedeutung
float frontRight	Der vordere rechte Punkt auf dem Konzertgelände (Richtung Bühne).
float frontLeft	Der vordere linke Punkt auf dem Konzertgelände (Richtung Bühne).
float backRight	Der hintere rechte Punkt auf dem Konzertgelände (Richtung Bühne).

float backLeft	Der hintere linke Punkt auf dem Konzertgelände (Richtung Bühne).

Tabelle 4.e: Datenträger der *BorderComponent*

4.3.1.3 Input Component

Da der datenorientierte Abschnitt dieses Projektes keinen Zugriff auf den objektorientierten Teil auf dem *Main-thread* hat (Standard Operationen wie *Input.getButtonDown("enter")* können nur auf dem *Main-thread* ausgeführt werden) aber auf Eingaben des Benutzers reagieren muss, wurde eine *InputComponent* erstellt, welche im laufe der Simulation von einem *System* aktualisiert wird.
Die *InputComponent* wird ebenfalls an alle Konzertbesucher angehängt, da diese in den meisten Fällen von den Eingaben des Benutzers betroffen sind.
Die folgenden Blitfähigen Datenträger sind in der *InputComponent* enthalten[A23] (Tabelle 4.f):

Blitfähiger Datenträger	Bedeutung
bool keyOnePressedDown	Beinhaltet die Information, ob Taste *eins* nach unten gedrückt wurde.
bool keyOnePressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *eins* beendet wurde.
bool keyTwoPressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *zwei* beendet wurde.
bool keyThreePressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *drei* beendet wurde.
keyFourPressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *vier* beendet wurde.
keyFivePressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *fünf* beendet wurde.

keySixPressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *sechs* beendet wurde.
keySevenPressedUp	Beinhaltet die Information, ob der Tastendruck von Taste *sieben* beendet wurde.

Tabelle 4.f: Datenträger der *InputComponent*

4.3.1.4 Agent Component

Um verschiedene Attribute sowie Status der Konzertbesucher zu speichern, wurde eine *AgentComponent* erstellt. Jedes *Entity* welches einen Konzertbesucher darstellt, erhält diese *AgentComponent*.

Neben den in *Components* standard vorkommenden Blitfähigen Datentypen, besitzt die *AgentComponent* außerdem noch einen *enum* namens *AgentStatus*. Dieser *AgentStatus* wird als Datentyp in der eigentlichen *AgentComponent* verwendet, um den derzeitigen Status des Konzertbesuchers zu speichern[A24].

Der *AgentStatus* ist wie folgt aufgebaut (Tabelle 4.g):

Enum-Eintrag	Bedeutung
Idle(0)	Der Konzertbesucher steht still.
Moving(1)	Der Konzertbesucher läuft mit normalem Tempo.
Dancing(2)	Der Konzertbesucher springt.
Running(3)	Der Konzertbesucher rennt.

Tabelle 4.g: *AgentStatus* Einträge

Dieser *enum* wird in der *AgentComponent* wie folgt verwendet (Tabelle 4.h):

Blitfähiger Datenträger	Bedeutung
bool hasTarget	Der Konzertbesucher besitzt aktuell ein Ziel, um dorthin zu laufen oder zu rennen.
float3 target	Die *x*,*y* sowie *z* Position von dem Ziel des Konzertbesuchers.
AgentStatus agentStatus	Der derzeitige Status des Konzertbesuchers (still stehen, tanzen, laufen, rennen).
bool exitPointReached	Der Konzertbesucher hat den anvisierten Ausgang erreicht.
bool foundTemporaryNewRandomPosition	Eine zufällige Position wurde für den Konzertbesucher generiert als eine *Panikoption* platziert wurde.
bool foundFinalExitPoint	Der Konzertbesucher hat sich für einen finalen Ausgang entschieden.
bool marked	Der Konzertbesucher hatte sich für einen finalen Ausgang entschieden. Auf dem Weg dorthin hat er aber bemerkt, dass sich dort eine größere Menge an Konzertbesuchern angesammelt hat. Er beschließt also, diesen Ausgang zu meiden und “dreht um”. Er wird also markiert.
float fleeProbability	Die Wahrscheinlichkeit, dass der Konzertbesucher einen überfüllten Ausgang meidet, während er diesen bereits als Ziel markiert hat.

Tabelle 4.h: Datenträger der *AgentComponent*

4.3.1.5 Move Speed Component

Die Konzertbesucher sollen sich je nach *AgentStatus* mit einer unterschiedlichen Geschwindigkeit bewegen. Aus diesem Grund wurde eine weitere *Component* namens *MoveSpeedComponent* angelegt.

Die folgenden Blitfähigen Datenträger sind in der *MoveSpeedComponent* enthalten[A25] (Tabelle 4.i):

Blitfähiger Datenträger	Bedeutung
float moveSpeed	Die normale Bewegungsgeschwindigkeit der Konzertbesucher.
float runningSpeed	Die Bewegungsgeschwindigkeit der Konzertbesucher beim rennen.
float jumpSpeed	Die Sprunggeschwindigkeit der Konzertbesucher
float panicJumpSpeed	Die Sprunggeschwindigkeit der Konzertbesucher wenn Panik ausgebrochen ist. Die Konzertbesucher springen, während sie rennen.

Tabelle 4.i: Datenträger der *MoveSpeedComponent*

4.3.1.6 Dummy Component

Der Benutzer soll in der Simulation in der Lage sein, verschiedene Ausgänge an den Seitenteilen zu erstellen. Damit die Ausgänge nur von einem Startpunkt aus erstellt werden, wurde eine *DummyComponent* erstellt. Diese wird einem einzelnen *Entity* zugeordnet, wodurch die Erstellung der Ausgänge nur ein mal ausgeführt wird. Würde man die *DummyComponent* beispielsweise jedem Konzertbesucher zuordnen, dann würde das *System*, welches die Ausgänge erstellt, über jeden Konzertbesucher einen Ausgang an Position *x* erstellen.
In diesem Fall wird die *DummyComponent* einem normalen *Entity* zugeordnet, dadurch existiert nur ein *Entity*, womit das *System* der “Ausgangserstellung” arbeiten kann.
Die *DummyComponent* besitzt keine Blitfähigen Datenträger, da sie lediglich zur Separation von *Entities* verwendet wird[A26].

4.3.1.7 Exit Component

Neben einer visuellen Darstellung der Ausgänge, werden im Hintergrund weitere *Entities* generiert, welche die Ausgänge repräsentieren. Diese *Entities* besitzen ebenfalls Attribute, welche in der *ExitComponent* festgehalten werden.

Die folgenden Blitfähigen Datenträger sind in der *ExitComponent* enthalten[A27] (Tabelle 4.j):

Blitfähiger Datenträger	Bedeutung
bool overloaded	Die Information, ob ein Ausgang mit Konzertbesuchern “überladen” ist.
float amount	Die Anzahl der Konzertbesucher im Umkreis von einem Ausgang.

Tabelle 4.j: Datenträger der *ExitComponent*

4.3.2 Datenorientierte Programmierung (ECS) + C#-Jobsystem - Systems

Das folgende Kapitel präsentiert die Implementierungen der einzelnen *Systems* der Massenpaniksimulation. Zudem werden alle zusätzlichen Verwendungen des *C#-Jobsystems* erläutert.

4.3.2.1 Jobs in ECS Systems - Vorwort

Diese Arbeit bezieht sich auf den vollen Umfang von *Unity DOTS*. Aus diesem Grund wurde versucht, dass *C#-Jobsystem* mit in die *ECS Systems* einzupflegen.
Die Syntaxstruktur besitzt dabei immer den gleichen Aufbau.
Damit ein *System Jobs* verwenden kann, muss es zuerst von dem *Interface JobComponentSystem* erben, erst dann können *Jobs* erstellt werden, welche von den unterschiedlichen *Interfaces IJobForEach* und *IJobForEachWithEntity* erben können. *IJobForEach* und *IJobForEachWithEntity* sind sich dabei sehr ähnlich, sie benötigen beide *Components* als Parameter, dadurch erhält der erstellte *Job* die Information, auf welchen *Entities* er operieren soll, nämlich auf *Entities* mit diesen übergebenen *Components* als Parametern. Dabei ist es egal, wie viele zusätzliche *Components* die *Entities* besitzen, wichtig ist nur, dass die *Components* welche als Parameter übergeben wurden, vorhanden sind.

Jeder *Job* implementiert eine Methode namens *Execute*. Diese Methode erlaubt den Zugriff auf das derzeitige *Entity*, welches von diesem *Job* "bearbeitet" wird, außerdem kann man auch auf die *Components* des *Entitys* zugreifen, wichtig ist dabei nur, dass man nur Zugriff auf *Components* erhält, welche vorher im *Job* selbst "gesucht" wurden, dieser Zugriff kann dabei lesend über das "keyword" *ReadOnly* oder schreibend über das "keyword" *WriteOnly* geschehen. Der *Job* operiert also auf *Entities* mit bestimmten *Components*, *Execute* hat dann Zugriff auf diese *Components* von dem aktuell behandelten *Entity*.
Dieser Punkt beschreibt auch den Unterschied zwischen *IJobForEach* und *IJobForEachWithEntity*, während *IJobForEach* nur den Zugriff auf die *Components* des aktuell behandelten *Entitys* garantiert, liefert *IJobForEachWithEntity* zusätzlich noch das *Entity* und deren *Index* selbst noch als Parameter. Dadurch können beispielsweise zusätzliche *Components* über einen *CommandBuffer* an das *Entity* angehängt oder entfernt werden oder verschiedene Werte in einem *Native Array* abgespeichert und zwischen verschiedenen *Jobs* verteilt werden, der *Entity Index* hilft anschließend dabei, den gespeicherten Wert aus einem früheren *Job* im neuen *Job* wieder zu finden. *CommandBuffer* und *Native Arrays* spielen eine wichtige Rolle in *Jobs*. Diese Datenstrukturen sorgen dafür, dass Daten zwischen unterschiedlichen *Jobs* transferiert oder ausgeführt werden können.
Durch Wettlaufbedingungen ist es nicht erlaubt, dass *Components* in einem *Job*, also auf *Worker-threads*, an *Entities* angehängt werden. Aus diesem Grund müssen die auszuführenden Operationen in einem sogenannten *CommandBuffer* gespeichert werden. Sobald der *Job* abgearbeitet wurde, wird der *CommandBuffer* auf dem *Main-thread* "geöffnet" und die dort enthaltenen Befehle werden ausgeführt. Der *Main-thread* stellt dabei einen sogenannten *Synchpoint* dar, an dem solche Operationen ausgeführt werden dürfen. Der *Main-thread* ist außerdem der einzige Ort, an dem auf *Unity* eigene Klassen zugegriffen werden darf, beispielsweise kann man nur hier eine zufällige Zahl über die Klasse *Random* erstellen.
Ein *Native Array* hingegen besitzt die Fähigkeit, Daten zwischen verschiedenen *Jobs* zu transferieren. So können in einem *Job* beispielsweise Daten in einem *Native Array* über den *Entity Index* gespeichert und in einem anderen *Job* wieder geladen werden. Ein *Native Array* kann außerdem auf dem *Main-thread* Daten zugewiesen bekommen und diese dann auf verschiedene *Jobs* verteilen. *Native Arrays* sind deutlich performanter als normale *MonoBehaviour* Arrays.

Die letzte Methode eines jeden *Systems* ist die *OnUpdate* Methode. Diese stellt den *Main-thread* dar und beinhaltet die eigentliche Initialisierung eines *Jobs*, also das "starten" eines *Jobs* an einem bestimmten Punkt. Das ist auch der Ort, an dem die Operationen aus einem *CommandBuffer* geladen und ausgeführt werden.
Sobald es dazu kommt, dass mehrere *Jobs* in *OnUpdate* ausgeführt werden, müssen sogenannte *JobHandles* verwendet werden, diese speichern die Ergebnisse der *Jobs* für *Unity* interne Hintergrundprozesse ab, um kritische Wettlaufsituationen zu verhindern.

4.3.2.2 Unit Spawner Proxy - Der ECS Ausgangspunkt
Das *UnitSpawnerProxy* Skript beschreibt die Grundlage des gesamten *ECS* Systems. In diesem Skript befinden sich neben Variablen wie der Anzahl der *Entities*, den derzeitigen Abmessungen der Seitenteile oder einem *Prefab* für den Konzertbesucher, auch verschiedenste Methoden um den *ECS* Abschnitt dieses Projektes zu deklarieren. *UnitSpawnerProxy* ist dabei das einzige *ECS* Skript, welches noch von *MonoBehaviour* erben darf. *MonoBehaviour* bildet hier die Schnittstelle zwischen dem Entwickler und dem *ECS*, da das *UnitSpawnerProxy* Skript an das *crowd GameObject* im *Unity* Editor angeheftet wurde. Damit das *ECS* eingeleitet werden kann, erbt das *UnitSpawnerProxy* Skript noch von den *Interfaces IDeclareReferencedPrefabs* und *IConvertGameObjectToEntity*.

Durch diese *Interfaces* kann der Konzertbesucher (dargestellt durch ein Kapsel *GameObject*) in der Methode *DeclareReferencedPrefabs* in ein *Entity* konvertiert werden. *Unity* erhält also ein fertiges *GameObject*, welches in der *MonoBehaviour* Welt entwickelt wurde und konvertiert dieses in ein *Entity* um.

Die Methode *Convert* greift letztendlich das bereits existierende *crowd Entity* (wurde automatisch erstellt, da das *crowd GameObject* ein *Unity* eigenes *ConvertToEntity* Skript besitzt) auf und fügt über den sogenannten *EntityManager* die zuvor erstellten *Components UnitSpawnerComponent*, *BorderComponent* und *InputComponent* dem *crowd Entity* hinzu.
Zu diesem Zeitpunkt existiert also ein *Entity*, welches die zuvor erwähnten *Components* besitzt. Die hinzugefügten *Components* wurden dementsprechend mit den in diesem Skript deklarierten Variablen gespeichert, so wurde auch eine *Entity* Kopie des derzeitigen Konzertbesuchers in der *UnitSpawnerComponent* gespeichert, diese wird im späteren Zeitpunkt verwendet, um beliebig viele Konzertbesucher in einem *System* zu erstellen[A28].

4.3.2.3 Unit Spawner System

Das *UnitSpawnerSystem* generiert die vom Nutzer bestimmte Menge der Konzertbesucher, dabei enthält das System einen *Job* namens *SpawnJob*. Dieser *Job* erbt von *IJobForEachWithEntity* und operiert auf *Entities* mit den *Components* *UnitSpawnerComponent*, *BorderComponent* und *InputComponent*.
Aus diesen *Components* werden verschiedene Daten entnommen. In der *Execute* Methode wird dauerhaft geprüft, ob die Taste *eins* betätigt wurde, sollte das passiert sein, bezieht der *Job* aus der *UnitSpawnerComponent* die Anzahl der zu generierenden *Entities* "AmountToSpawn" und generiert darauf hin, abhängig von *AmountToSpawn*, Konzertbesucher in Form von weiteren *Entities*. Dabei wird die *Entity* Vorlage aus der *UnitSpawnerComponent* bezogen.
Sobald ein Konzertbesucher *Entity* generiert wurde, wird eine zufällige Position auf dem Konzertgelände generiert und der *TranslationComponent* des *Entitys* zugeordnet, die *TranslationComponent* eines *Entitys* speichert dabei Informationen wie die derzeitige Rotation und Position. Da in einem *Job*, beziehungsweise auf einem *Worker-thread* keine *Unity* Klassen wie *Random* aufgerufen werden können, wurde ein eigenes *Random* Objekt erstellt, welches als sogenannten *Seed* den *Entity Index* verwendet und somit für jedes *Entity* eigene zufällige Werte erstellen kann.
Sobald der Konzertbesucher eine zufällige Position besitzt, werden über einen *CommandBuffer* verschiedene *Components* an das *Entity* angeheftet.
Auf diese Weise werden alle Konzertbesucher generiert. Nachdem dieser Prozess abgearbeitet wurde, deaktiviert der *SpawnJob* das gesamte *UnitSpawnerSystem.* Dadurch wird verhindert, dass der *Job* erneut ausgeführt wird und alle Konzertbesucher erneut generiert werden.

Die *OnUpdate* Methode (äquivalent zum *Main-thread*) in diesem *System* beinhaltet die eigentliche Initialisierung und Ausführung des *SpawnJobs.* Außerdem werden die im *CommandBuffer* gespeicherten Operationen hier gelesen und ausgeführt[A29].

4.3.2.4 Input System

Das *InputSystem* aktualisiert die “Inputdaten” des Nutzers im *ECS*, dabei enthält das *System* zwei *Jobs*, den *PlayerInputJob* und den *CreateExitEntities Job*.
Die Aufgabe des *PlayerInputJob*s, welcher auf allen *Entities* mit der *InputComponent* operiert, besteht darin, die übergebenen “Inputdaten” aus dem *Main-thread* (bei der *Job* Initialisierung in *OnUpdate*) in die entsprechende *InputComponent* des aktuellen *Entitys* zu transferieren, dadurch erhalten andere *Systems* aktuelle Informationen über das Eingabeverhalten des Benutzers.
Der *CreateExitEntities Job* operiert auf allen *Entities*, welche die *DummyComponent* besitzen. Sobald dieser *Job* im *Main-thread* gestartet wird, werden verschiedene Daten, darunter die Position des erstellten Ausgangs, mit transferiert, welche später im *Job* selbst verarbeitet werden. Die *Execute* Methode behandelt die eigentliche Erstellung der Ausgangs *Entities* mit Hilfe des *CommandBuffers*.
An diesem Punkt wird auch sichergestellt, dass das *RemoveExitsSystem* ausgeschaltet ist, da sonst die soeben erstellten Ausgangs *Entities* sofort wieder entfernt werden.

Auf dem *Main-thread*, also in *OnUpdate* wird der *PlayerInputJob* sowie der *CreateExitEntities Job* gestartet, dabei ist zu beachten, dass der *CreateExitEntities Job* nur dann gestartet wird, wenn die passende Panikoption (“Ausgang erstellen”) im radialen Benutzer Menü ausgewählt und mit der linken Maustaste auf einem Seitenteil geklickt wurde. Die Position für das Ausgangs *Entity* wird dabei von dem angeklickten Seitenteil *GameObject* bezogen und gespeichert. Das Seitenteil selbst wird ausgeblendet[A30], der passende “Markierungspfosten” wird im *Actions* Skript erstellt.

4.3.2.5 Moving System

Das *Moving System* berechnet, sobald gefordert, das “normale gehen” des *Entitys*. In diesem *System* befindet sich ein *Job* namens *MovementBurstJob*, welcher nur auf Konzertbesucher *Entities* operiert. Sobald ein *Entity* den *AgentStatus Moving* besitzt und im derzeitigen Zustand ein Ziel hat (*hasTarget*), wird die Richtung sowie die Distanz zwischen der aktuellen Position und dem Ziel berechnet. Abhängig von der Distanz wird daraufhin das *Entity* zum Ziel bewegt, indem die Richtung, welche als *float3* dargestellt wird, auf die derzeitige Position über *translation.value* hinzuaddiert wird.
Sollte der Konzertbesucher an seinem Ziel angekommen sein, wird dessen *hasTarget* Wert zurück auf *false* gesetzt, dadurch kann der *MovementBurstJob* nicht mehr vollständig ausgeführt werden, da die vorherige Bedingung nicht mehr gilt.

Dieser Punkt markiert den Startpunkt des *CalculateNewRandomPositionSystem*, da dieses erst operiert, sobald der *AgentStatus* auf *Moving* und *hasTarget* auf *false* gestellt ist.

Auf dem Main-thread, also in *OnUpdate* wird der MovementBurstJob gestartet.[A31]

4.3.2.6 Jumping System

Das *Jumping System* verhält sich ähnlich wie das *Moving System*. Das *Jumping System* berechnet die "Tanzbewegungen" des Konzertbesuchers. Der im *Jumping System* enthaltene *Job* namens *JumpingJob* operiert auf allen Konzertbesucher *Entities*. In der *Job* eigenen *Execute* Methode wird zwischen dem *AgentStatus* "Dancing" sowie "Running" unterschieden, um zwischen "normalem Springen" sowie dem "springen während dem rennen" zu unterscheiden. In der Ausführung sind beide Bedingungen dieselben, sie unterschieden sich nur in der Geschwindigkeit während des springens. Die Konzertbesucher werden bei jeder Iteration des *Jobs* weiter nach oben bewegt, danach wird geprüft, ob sie eine bestimmte Höhe erreicht haben. Sollte das der Fall sein, wird die Richtung geändert, indem das Datum der *MoveSpeedComponent jumpSpeed* oder *panicJumpSpeed* negativ konvertiert wird, dadurch wird der Konzertbesucher bei der nächsten Iteration des *Jobs* nach unten bewegt.
Die zweite Abfrage behandelt die positive Konvertierung bei einem bestimmten *y* Wert (0.5, da sich bei diesem Wert der Konzertbesucher wieder auf dem Boden befindet), sodass sich der Konzertbesucher bei der nächsten *Job* Iteration wieder nach oben bewegt.
Durch dieses Umschalten der *jumpSpeed* beziehungsweise *panicJumpSpeed* Daten in den positiven oder negativen Bereich und durch das auf und ab bewegen des Konzertbesuchers entsteht eine "Sprunganimation", welche in der Simulation selbst später als "Tanzanimation" angesehen werden kann.
Der *JumpingJob* beinhaltet außerdem noch das "Ausgleichen" während des springens der Konzertbesucher, sobald sie wieder anfangen zu laufen, zu rennen oder still zu stehen. Die Konzertbesucher werden dabei unauffällig auf den Boden geladen, damit sie, währenddessen sie andere Aktionen ausführen, nicht in einer höheren *y* Position als 0.5 stehen bleiben.

Auf dem Main-thread, also in *OnUpdate* wird der *JumpingJob* gestartet.[A32]

4.3.2.7 Running System

Das *Running System* enthält den *RunningJob* und berechnet die eigentliche "Rennbewegung" der Konzertbesucher. Dabei verhält sich das *RunningSystem* ähnlich wie das *MovingSystem*.
Es wird geprüft, ob der Konzertbesucher den *Running AgentStatus* sowie ein Ziel *hasTarget* besitzt. Sollte das der Fall sein, wird die Distanz zwischen der derzeitigen Position des Konzertbesuchers und des Ziels berechnet. Wenn diese Distanz größer als 0.1 ist, wird erneut die Richtung zwischen der derzeitigen Position und Ziel berechnet (*float3*) und der aktuellen Position des Konzertbesuchers hinzuaddiert.
Sollte die Distanz kleiner als 0.1 sein, also sobald der Konzertbesucher sehr nah am Ziel ist, wird geprüft, ob das Datum *foundFinalExitPoint* aus der *AgentComponent* auf *true* gestellt ist, um zu prüfen, ob das Ziel ein Ausgang ist. Wenn das Ziel ein Ausgang ist, wird wieder der *Moving AgentStatus* gesetzt, damit sich der Konzertbesucher auf dem äußeren Gelände "laufend" verteilt, dies geschieht mit Hilfe des *Moving Systems* sowie dem *CalculateNewRandomPositionSystem* (späteres Kapitel), außerdem wird *exitPointReached* auf *true* gesetzt, damit im *CalculateNewRandomPositionSystem* entschieden werden kann, ob dieser Konzertbesucher durch einen Ausgang gelaufen ist. Wenn *foundFinalExitPoint* auf *false* gesetzt ist, wird *hasTarget* auf *false* gesetzt, damit das *PanicSystem* (späteres Kapitel) eingreifen kann um eine neue Position in dieser "Paniksituation" zu berechnen.

Auf dem Main-thread, also in *OnUpdate* wird der *RunningJob* gestartet.[A33]

4.3.2.8 CalculateNewRandomPosition System

Das *CalculateNewRandomPositionSystem* enthält einen *Job* namens *CalculateNewRandomPositionBurstJob* sowie eine Methode namens *IsInsideFestivalArea()*. Das *System* berechnet neue zufällige Positionen für die Konzertbesucher. Diese Positionen beziehen sich auf neue zufällige Positionen, welche während des *Moving AgentStatus* benötigt werden.
Der *CalculateNewRandomPositionBurstJob* unterscheidet dabei zwischen Konzertbesuchern welche sich außerhalb oder innerhalb des Geländes aufhalten. Sollte sich ein Konzertbesucher außerhalb des Geländes aufhalten, wird dementsprechend eine zufällige Position berechnet, welche sich außerhalb des Geländes befindet.
Der andere Fall behandelt *Entities*, welche sich innerhalb des Geländes mit dem *Moving AgentStatus* aufhalten und während des laufens eine neue Position benötigen. In diesem Fall wird ebenfalls eine neue zufällige Position berechnet, diese allerdings befindet sich dann innerhalb des Geländes.[11]

Die *IsInsideFestivalArea* Methode greift die Berechnungen aus dem *CalculateNewRandomPositionBurstJob* auf, damit diese aus einem anderen *System* verwendet werden können. Die Methode überprüft im allgemeinen, ob eine *testPosition* innerhalb des Geländes ist oder nicht. Die Methode gibt dementsprechend *true* oder *false* zurück.

Auf dem *Main-thread*, also in *OnUpdate* wird der *CalculateNewRandomPositionBurstJob* gestartet, außerdem werden dort zufällige *seeds* generiert, mit denen zufällige Werte innerhalb des *Jobs* erstellt werden können.[A34]

4.3.2.9 Panic System

Das *PanicSystem* bestehend aus der Methode *CheckForClosestExit* sowie den drei *Jobs* *EnablePanicModeJob*, *PanicJob* und *ReactOnPanicInsideQuadrantJob* präsentiert den Hauptteil dieses Projektes. Das *PanicSystem* berechnet das allgemeine Verhalten der *Entities*, sobald Panik ausgelöst wurde. Das *System* berechnet die Reaktion der *Entities*, welche sich in unmittelbarer Nähe der "Panikposition" aufhalten, als auch die Reaktion der *Entities*, welche sich in der Nähe von "panischen" *Entities* aufhalten.

Innerhalb des *Main-threads*, also in *OnUpdate* wird abhängig von der aktuellen "Panikoption" der *EnablePanicModeJob* mit einem unterschiedlichen *panicRadius* gestartet, woraufhin der *PanicJob* initialisiert und gestartet wird. Am Ende von *OnUpdate* wird der *ReactOnPanicInsideQuadrantJob* gestartet, dies geschieht aber nur, wenn eine "Panikoption" vom Benutzer platziert wurde.[A35]

4.3.2.9.1 CheckForClosestExit

Diese Methode nimmt ein *Native Array* aus den aktuellen Ausgangspositionen sowie verschiedenen zusätzlichen Variablen entgegen und berechnet daraus den am naheliegendsten Ausgang bezüglich des aktuellen Konzertbesuchers.
Sobald dieser Ausgang ermittelt wurde, erhält der aktuelle Konzertbesucher diesen Ausgang als Ziel, *hasTarget* sowie *foundFinalExitPoint* der *AgentComponent* werden dabei auf *true* gesetzt, damit das *RunningSystem* an diesem Punkt übernimmt und den Konzertbesucher zum ausgewählten Ausgang führt.[A35]

4.3.2.9.2 EnablePanicModeJob

Dieser *Job* operiert auf jedem aktiven Konzertbesucher und prüft dabei, ob sich der aktuelle Konzertbesucher in der Nähe von einer "Panikoption" befindet. Dieser *Job* wird gestartet, sobald eine "Panikoption" platziert wurde.

Dafür erhält der *Job* vom *Main-thread* die Position der “Panikoption” sowie einen *panicRadius* um die Konzertbesucher *Entities* einzugrenzen. Aus diesen Daten wird dann geprüft, ob die Distanz eines *Entitys* kleiner als der übergebene *panicRadius* ist, wenn das der Fall sein sollte, erhält das aktuelle *Entity* den *Running AgentStatus*. Dadurch können andere *Jobs* in diesem *System* mit diesem *Entity* arbeiten.[A35]

4.3.2.9.3 PanicJob

Der *PanicJob* wird dauerhaft ausgeführt und berechnet dabei individuelles Verhalten der Konzertbesucher bei einem “Panikausbruch”.
Der *PanicJob* besteht aus insgesamt drei Abfragen, jede davon optimiert das Verhalten der Konzertbesucher bei einem “Panikausbruch” und sorgt dafür, dass der “Panikausbruch” der Konzertbesucher so realistisch wie möglich wirkt.

Die *erste* Abfrage bezieht sich auf *Entities*, welche sich im *Running AgentStatus* befinden aber zum jetzigen Zeitpunkt kein Ziel haben (*hasTarget = false*), diese *Entities* beschreiben Konzertbesucher, welche sich in unmittelbarer Nähe von einer “Panikoption” aufgehalten haben, durch den *EnablePanicModeJob* wurden diese nämlich mit dem *Running AgentStatus* versehen.
In diesem Fall wird eine zufällige Position auf dem Konzertgelände berechnet, dabei wird die *IsInsideFestivalArea* Methode aus dem *CalculateNewRandomPositionSystem* verwendet. Sobald diese Methode bestätigt, dass sich die generierte Position auf dem Gelände befindet, wird diese dem aktuellen *Entity* als Ziel zugeordnet, *hasTarget* sowie *foundTemporaryNewRandomPosition* werden auf *true* gesetzt.[A35]

Während der *zweiten* Abfrage befinden sich die Konzertbesucher auf dem Weg zu einem Ziel. In der *ersten* Abfrage wurde eine zufällige Position auf dem Konzertgelände berechnet und zugewiesen, diese Datenveränderung wurde daraufhin vom *RunningSystem* aufgegriffen und bearbeitet, der Konzertbesucher befindet sich auf dem Weg zu seinem Ziel, das *PanicSystem* allerdings wird weiterhin ausgeführt. Da die *erste* Abfrage für dieses *Entity* nun nicht mehr gilt (*Entity* besitzt bereits ein Ziel), wird die *zweite* Abfrage aufgegriffen, diese wurde nämlich genau für solche *Entities* konzipiert. Während der Konzertbesucher auf dem Weg zu seinem Ziel ist, läuft eine Schleife permanent durch alle verfügbaren Ausgänge und berechnet, ob sich ein Ausgang in unmittelbarer Nähe des Konzertbesuchers befindet. Sollte das der Fall sein, wird zusätzlich noch geprüft, ob dieser Ausgang überladen ist, also ob sich zu viele andere Konzertbesucher an diesem Ausgang aufhalten (mit Hilfe von dem *Mass System* (späteres Kapitel)).

Der Ausgang wird als Ziel ausgewählt, wenn dieser nicht überladen ist, außerdem werden die Daten *hasTarget* und *foundFinalExitPoint* auf *true* gesetzt. Sollte er überladen sein, wird *hasTarget* auf *false* gesetzt, damit Abfrage *eins* wieder gelten kann, dadurch wird wieder eine neue zufällige Position auf dem Gelände berechnet.[A35]

Die *dritte* Abfrage behandelt die letzte Situation die eintreten kann. Sobald der Konzertbesucher auf dem Weg zu einem nicht überladenen Ausgang ist, dieser Zustand sich aber auf dem Weg ändert, sprich der Ausgang plötzlich überladen ist, gilt die *dritte* Abfrage.
Während der Konzertbesucher auf dem Weg zu seinem Ziel (dem Ausgang) ist, werden erneut alle Ausgänge überprüft, dabei wird zuerst der richtige Ausgang rausgesucht, danach wird überprüft, ob dieser in diesem Moment überladen ist. Sollte das der Fall sein, wird eine zufällige Zahl zwischen 1 und 1000 generiert und mit der *fleeProbability* des derzeitigen Konzertbesuchers verglichen. Wenn die generierte Zahl kleiner als die *fleeProbability* ist, wird das Ziel verworfen, *hasTarget* sowie *foundFinalExitPoint* werden wieder auf *false* gesetzt, dadurch gilt wieder Abfrage *eins*, eine neue zufällige Position auf dem Gelände wird berechnet, der Zyklus beginnt von vorne.[A35]

4.3.2.9.4 ReactOnPanicInsideQuadrantJob - Vorwort Quadrant System

Damit sich die Konzertbesucher gegenseitig “anstecken”, sobald Panik ausbricht, musste eine Lösung entwickelt werden, die Informationen an die “benachbarten” Konzertbesucher zu übermitteln. Aus performance Gründen wurde für dieses Vorhaben ein speziell für das *ECS* ausgerichtete *Quadrant System* angewandt.
Ein *Quadrant System* unterteilt den verwendeten Teil der momentanen *Szene* in einzelne so genannte *Quadrants*. Einzelne *Entities* können mit diesen *Quadrants* Informationen abfragen und austauschen. Dieser Vorgang kann dabei innerhalb des eigenen *Quadrants* oder bei einem benachbarten *Quadrant* geschehen. Dieses Vorhaben ist also ideal dafür geeignet, eine “Panikreaktion” exponentiell auf dem gesamten Gelände zu verbreiten.[6, A36]

Für das *QuadrantSystem* wurde zuerst eine Methode namens *GetPositionHashMapKey* implementiert, diese nimmt eine Position entgegen und erzeugt daraus einen individuellen Schlüssel für diesen aktuellen *Quadrant*. Beispielsweise kann dieser Methode eine *Entity* Position übergeben werden, wodurch dann der *Quadrant* Schlüssel von dem *Quadrant* unter dem *Entity* ermittelt werden kann.
Um Informationen zu speichern, wurde ein *struct* namens *QuadrantData* angelegt. Dieses *struct* enthält verschiedene Datenträger, welche dieses *Quadrant* speichern kann.

Der eigentliche Träger dieses *QuadrantData* structs ist eine *NativeMultiHashMap<int,QuadrantData>* namens quadrantMultiHashMap.[6, A36]

Um die *Entities* (Konzertbesucher und Ausgänge) mit dem *QuadrantSystem* zu verbinden, wurden zwei *Jobs* geschrieben, welche die verschiedenen *Entities* in die *quadrantMultiHashMap* laden, *SetQuadrantDataHashMapJob* und *SetQuadrantHashMapJobForExits*. Die beiden *Jobs* unterscheiden sich in der Auswahl der *Entities* auf denen sie operieren.
Der *SetQuadrantDataHashMapJob* operiert auf Konzertbesuchern und der *SetQuadrantDataHashMapJobForExits* operiert auf Ausgängen.
Um einen *Quadrant* Datensatz zu speichern, werden zuerst die *Quadrant* Schlüssel über *GetPositionHashMapKey* berechnet, diese werden dann zusammen mit einem *QuadrantData struct* gespeichert. Auf diese Weise kann man jederzeit diesen Datensatz wiederfinden, man braucht lediglich den richtigen Schlüssel.[6, A36]

Damit die Konzertbesucher in einem späteren Zeitpunkt an einem Ausgang gezählt werden können, wurde bereits in diesem *System* eine Methode namens *GetEntityCountInHashMap* geschrieben. Diese Methode iteriert durch die gesamte *NativeMultiHashMap* und zählt dabei jeweils um eins hoch. Zum Schluss gibt die Methode einen *int* Wert zurück, welcher die eigentliche Anzahl der Konzertbesucher um einen Ausgang darstellt.[6, A36]

Auf dem *Main-thread*, also in *OnUpdate* werden die beiden *Jobs* *SetQuadrantDataHashMapJob* und *SetQuadrantDataHashMapJobForExits* initialisiert und gestartet. Außerdem wird dort dauerhaft die größe der *NativeMultiHashMap* überprüft, sollte sich die Anzahl der verfügbaren *Entities* (Konzertbesucher und Ausgänge) erhöhen, dann muss sich auch die *NativeMultiHashMap* an die Situation anpassen und sich selbst vergrößern.[6, A36]

Mit diesem *QuadrantSystem* ist es nun möglich, Informationen zwischen einzelnen Konzertbesuchern auszutauschen, dabei kann eingestellt werden, wie weit der Abstand zwischen einzelnen *Entities* sein soll. Außerdem kann die Anzahl an *Entities* in Gruppierungen gezählt werden, sodass andere *Entities* auf diese Gruppierungen reagieren können, indem sie den Ausgang in der Nähe dieser Gruppierungen meiden.

4.3.2.9.5 ReactOnPanicInsideQuadrantJob

Der *ReactOnPanicInsideQuadrantJob* wird ebenfalls dauerhaft ausgeführt und berechnet die Reaktion der Konzertbesucher, sobald sie auf einen anderen Konzertbesucher treffen, welcher bereits in "Panik verfallen" ist.
Für dieses Vorhaben wurde eine Methode namens *CalculatePanicReaction* erstellt. Diese Methode wird innerhalb des *ReactOnPanicInsideQuadrantJobs* mit unterschiedlichen Schlüsseln aufgerufen, die Schlüssel werden dabei zufällig generiert, dadurch reagiert ein Konzertbesucher nicht nur auf andere Konzertbesucher in seinem eigenen *Quadrant*, sondern auch auf zufällig gewählte *Quadrants* in unmittelbarer Nähe zu seinem eigenen *Quadrant*.

Da *CalculatePanicReaction* innerhalb des *Jobs ReactOnPanicInsideQuadrantJob* aufgerufen wurde, erhält *CalculatePanicReaction* Zugriff auf die vom *Main-thread* übergebene *NativeMultiHashMap*.
CalculatePanicReaction durchläuft nun die *NativeMultiHashMap* und überprüft jeden Konzertbesucher.
Dabei wird geschaut, ob ein Konzertbesucher im *Quadrant* des aktuellen Besuchers den *Running AgentStatus* besitzt, sollte das der Fall sein, erhält der aktuelle Konzertbesucher den *Running AgentStatus*. Außerdem wird überprüft, ob dieser Konzertbesucher bereits ein Ziel besitzt, wenn das der Fall sein sollte, dann kopiert der aktuelle Konzertbesucher dieses Ziel und "rennt dem anderen Konzertbesucher hinterher".

Der *ReactOnPanicInsideQuadrantJob* wird öfters pro Sekunde auf mehreren *Worker-threads* aufgerufen, *CalculatePanicReaction* wird innerhalb des *ReactOnPanicInsideQuadrantJobs* neun mal für verschiedene *Quadrants* aufgerufen und *CalculatePanicReaction* überprüft jedes *Entity* dieser *Quadrants*. Diese Berechnungen präsentieren dabei einen der aufwändigsten Teile des gesamten Projektes, zeigt gleichzeitig aber auch, wie performant *Unity DOTS* ist, da das gesamte *PanicSystem* gerade einmal 0.02 Sekunden braucht (bei einer durchschnittlichen *Entity* Anzahl) um einmal ausgeführt zu werden, wenn man zusätzlich noch bedenkt, dass das *PanicSystem* nicht nur aus einem *Job* besteht.[A35]

4.3.2.10 Mass System

Wie bereits erwähnt, wird ein *System* benötigt, welches die Konzertbesucher Anzahl im direkten Umfeld eines Ausgangs berechnet. Das *MassSystem* besitzt einen *Job* namens *CalculateEntitiesAroundExitJob*, welcher diese Aufgabe übernimmt.

Der *CalculateEntitiesAroundExitJob* operiert dabei auf allen Ausgängen der Simulation und ruft dabei die Methode *GetEntityCountInHashMap* aus dem *QuadrantSystem* für jeden *Quadrant* im direkten Umfeld des Ausgangs auf, welcher eine Einheit entfernt ist. Der zurückgegebene Wert der Methode wird dabei in der Variable *amount* gespeichert und im späteren Verlauf des *Jobs* kontrolliert.
Sollte der finale Wert größer als 10 sein, gilt dieser Ausgang als überladen, dies wird mit Hilfe des Datenträgers *overloaded* dargestellt.

Auf dem *Main-thread*, also in *OnUpdate* wird der *CalculateEntitiesAroundExitJob* gestartet.[A37]

4.3.2.11 Update Borders System

Das *UpdateBordersSystem* aktualisiert die derzeitigen Geländemaße der Konzertbesucher *Entities*. Dabei enthält das *System* einen *Job* namens *UpdateBordersJob*, welcher die übergebenen Informationen aus dem *Main-thread* an das derzeitige *Entity* überträgt.
Das *UpdateBordersSystem* enthält außerdem noch einen eigenen *Job* namens *DisableUpdateBordersSystemJob*, welcher nach Beendigung der aktualisierungen das gesamte System deaktiviert, wodurch Leistung eingespart wird. Das *System* wird aus dem *ManagerSystem* gestartet (späteres Kapitel).

Auf dem *Main-thread*, also in *OnUpdate* wird der *UpdateBordersJob* gestartet, dieser erhält die aktuellen Maße der Seitenteile des Konzertgeländes. Im Anschluss wird der *DisableUpdateBordersSystemJob* initialisiert und gestartet.[A38]

4.3.2.12 Remove Agents System und Remove Exits System

Das *RemoveAgentsSystem* und das *RemoveExitsSystem* werden beide aktiviert, sobald die passende Taste vom Benutzer betätigt wurde, dieser Vorgang geschieht aus dem *ManagerSystem* (späteres Kapitel).
Beide *Systems* enthalten dafür einen *Job*, welcher auf den jeweiligen *Entities* (Konzertbesucher und Ausgänge) operiert und jedes einzelne *Entity* über einen *CommandBuffer* entfernt. Der *CommandBuffer* wird benötigt, da das Entfernen der *Entities* nur auf dem *Main-thread* geschehen kann.

Außerdem enthalten beide *Systems* wieder einen *Job*, welcher die jeweiligen *Systems* deaktiviert.

Auf dem *Main-thread*, also in *OnUpdate* wird der jeweilige “Remove Job” gestartet. Außerdem werden hier die *CommandBuffer* “geöffnet” und die dort enthaltenen Operationen (das Löschen der *Entities*) ausgeführt.[A39, A40]

4.3.2.13 Manager System

Das *ManagerSystem* koordiniert die verschiedenen *Systems* in diesem Projekt. Das *ManagerSystem* enthält dafür drei *Jobs* namens *ManagerJob*, *ManagerInputJob* und *UpdateEntityAmount*.

Der *ManagerJob* kontrolliert dabei die Zuordnung der aktuellen *AgentStatus* der Konzertbesucher wenn noch keine “Panikoption” platziert wurde, dabei wird eine zufällige Zahl generiert und mit vordefinierten Wahrscheinlichkeiten abgewogen. Je nach Wahrscheinlichkeit erhalten die Konzertbesucher dann den *AgentStatus Idle*, *Dancing* oder *Moving*.

Wie bereits erwähnt, kontrolliert das *ManagerSystem* die Aktivierung sowie die Deaktivierung von verschiedenen *Systems* , um Leistung einzusparen. Diese Aufgabe übernimmt in diesem *System* der *ManagerInputJob*. Dieser verwendet die *InputComponent*, um die aktuellen Eingaben des Benutzers zu analysieren. Sobald der Benutzer verschiedene Tasten betätigt (beispielsweise Taste *eins* um Konzertbesucher zu generieren), werden einzelne *Systems* aktiviert, damit diese ihre enthaltenen *Jobs* ausführen. Sobald die dort enthaltenen *Jobs* abgearbeitet wurden, deaktivieren sich die *Systems* von selbst.
Die vom Benutzer übergebenen Werte für die Anzahl der Konzertbesucher werden im *UpdateEntityAmount Job* an die *UnitSpawnerComponent* übertragen.

Auf dem *Main-thread*, also in *OnUpdate* werden die entsprechenden *Jobs* gestartet.[A41]

4.3.3 Burst Compiler

Wie bereits erwähnt, beinhaltet *Unity DOTS* neben dem *ECS* und dem *C#-Jobsystem* noch den so genannten *Burst-Compiler*, welcher den implementierten code der einzelnen *Jobs* sowie Methoden in hoch performanten Maschinencode konvertiert, indem sich dieser nur auf die grundlegendsten C# Operationen konzentriert. Aus diesem Grund ist es beispielsweise nicht möglich eine zufällige Zahl über die Klasse *Random* innerhalb von einem *Job* zu erstellen, da diese Klasse nicht zu der Grundlage von C# gehört. Solche Operationen müssen auf dem *Main-thread* ausgeführt und an die einzelnen *Jobs* übermittelt werden. Durch diese Konvertierung wird der bereits nebenläufig ablaufende *Job* code noch effizienter ausgeführt.
Der *Burst-Compiler* ist standardmäßig aktiviert, sobald die passende *Unity* Erweiterung heruntergeladen wurde. Damit der *Burst-Compiler* weiß, auf welchen *Jobs* er operieren soll, muss der "[BurstCompile]" "tag" über dem jeweiligen *Job* oder die jeweilige Methode gesetzt werden. Um einzelne *Jobs*/Methoden vom *Burst-Compiler* auszuschließen, wurde der "[BurstDiscard]" "tag" verwendet.

Wie bereits erwähnt, kann der *Burst-Compiler* nur mit grundlegendsten C# Operationen arbeiten. Aus diesem Grund wurden im Projekt häufig einzelne *Jobs* in zwei *Jobs* aufgeteilt, damit der *Burst-Compiler* auf den performance lastigen Operationen arbeiten kann.
Beispielsweise entfernt der *RemoveAgentsJob* alle Konzertbesucher auf dem Gelände, indem er die *DestroyEntity* Operation über den *CommandBuffer* für jedes *Entity* aufruft. Dieser *Job* besitzt den "[BurstCompile]" "tag", da dieser *Job* nur grundlegendste C# Operationen verwendet. Im Anschluss von diesem *Job* allerdings wird innerhalb vom *Main-thread* der *Job DisableRemoveAgentsSystemJob* aufgerufen, dieser besitzt als einzige Aufgabe die Deaktivierung des aktuellen *Systems* über den Zugriff über die Klasse "World.Active…". Da diese Operation nicht zu den grundlegendsten C# Operationen gehört, darf der "[BurstCompile]" "tag" bei diesem *Job* nicht gesetzt werden. Auf diese Weise wurde effizient ein *Job* so aufgeteilt, dass dieser nicht komplett auf den "[BurstCompile]" "tag" verzichten muss, sondern nur der *Job*, welcher die nicht grundlegendsten C# Operationen enthält.

5. Evaluation

Das folgende Kapitel evaluiert, ob das gesetzte Ziel dieser Arbeit erreicht, und eine Standalone-Festival-Massenpaniksimulation auf Basis von *Unity DOTS* umgesetzt wurde.

5.1 Eine auf Unity DOTS basierende Massenpaniksimulation

Ziel der Arbeit war die Implementierung einer Standalone-Festival-Massenpaniksimulation auf Basis der neuesten *Unity3D* Erweiterung *Unity DOTS*.
Großes Augenmerk wurde auf die Grundfunktionalitäten der Konzertbesucher gesetzt, diese umfassen neben dem “laufen”, “tanzen” und dem “rennen” auch das allgemeine Panikverhalten eines einzelnen Konzertbesuchers.

5.1.1 Unity3D und Unity DOTS

Die Entwicklung über die Spiel-Engine *Unity3D* generell verlief wie gewohnt sehr gut. Die Realisierung des Konzertgeländes wäre ohne den *Unity* eigenen *Assetstore* nicht möglich gewesen, da dieser viele kostenlose *Assets* wie z.B 3D Modelle, Grafiken, Texturen usw. anbietet, welche von anderen Entwicklern und Designern veröffentlicht wurden. Die jeweiligen Modelle mussten also lediglich heruntergeladen, im *Unity Editor* entpackt und an der gewünschten Position platziert werden. Zudem ergab sich auch die Möglichkeit, verschiedene Effekte einzubauen. Dadurch wirkt die Simulation in der Darstellung (wenn aktiviert) deutlich interessanter.
Die ersten Versuche, über ein 3D Modellierungsprogramm, eigene Modelle zu entwickeln, wurden schnell verworfen, da diese nicht der gewünschten Qualität entsprachen und die dafür benötigte Zeit nicht einkalkuliert war.
Die Navigation im *Unity* eigenen *Editor* verlief reibungslos, benötigte Erweiterungen wie *Unity DOTS* können schnell über den *Package Manager* heruntergeladen werden.

Unity DOTS konnte zu Beginn dieser Arbeit nur manuell über den Onlinedienst *github* heruntergeladen und im *Unity* Verzeichnis integriert werden. Verfügbare Updates wurden also nicht automatisch geladen, wodurch es häufiger zu Problemen kam, sobald die neueste Version wieder manuell in das *Unity* Verzeichnis geladen wurde. Im Spätsommer 2019 wurde *Unity DOTS* schließlich offiziell im *Package Manager* aufgenommen, wodurch eine Weiterführung der Erweiterung durch die Spiel-Engine *Unity3D* offiziell fest stand. *Unity DOTS* erhielt ab diesem Zeitpunkt fast täglich Aktualisierungen, wodurch die Erweiterung ständig wuchs.

Durch die Verwendung einer Erweiterung, welche sich in der frühen “pre-Alpha” Phase befand, war die allgemeine Entwicklung der Simulation zusätzlich komplizierter. Viele Konstrukte der Erweiterung wurden in den offiziellen *Unity*-Entwickler Manuals nur erwähnt, allerdings nicht erläutert. Beispielprojekte auf diversen Plattformen wie *github* waren nur unkommentiert (oder in verkürzter Form) vorzufinden und diverse Foren enthielten nur wenige Informationen um ein eigenes Projekt zu realisieren.

5.1.2 ECS

Das *ECS* präsentierte während der Entwicklung den kompliziertesten Abschnitt der Erweiterung *Unity DOTS*. Im laufe des Projektes wurden immer wieder einzelne *Components* und *Systems* erstellt aber nach kürzerer Zeit wieder verworfen, da diese nicht (mehr) mit anderen *Components* und *Systems* funktionierten. Es wurden mehrere Wochen benötigt, um eine geeignete Struktur (wie sie in dieser Arbeit vorgestellt wurde) zu entwickeln, welche die eigentliche Massenpaniksimulation präsentiert. Diese Struktur wurde dann im laufe von wenigen Monaten mit Zusatzfunktionen und Absicherungsmethoden ausgefüllt. Dabei verhielten sich die einzelnen Konzertbesucher immer realistischer, verschiedene Fehlerquellen wurden dabei behoben.

Die spezielle *ECS* Syntax und die dazu korrelierende neue Denkweise zur datenorientierten Programmierung erzeugten zu Beginn fehlerhafte Ausgaben, welche meistens zu Abstürzen der Simulation führten. Die Fehlerquellen gingen dabei meistens von den *Systems* aus, da diese den eigentlichen *ECS* code beinhalteten und die *Components* nur die Träger der Daten waren, auf denen sie operierten. Durch die simple Syntax der *Components*, mussten diese nur einmal definiert werden, um mit Ihnen zu arbeiten. In Ausnahmefällen wurden vereinzelte *Components* (beispielsweise die *AgentComponent*) im laufe der Arbeit ergänzt, um zusätzliche Funktionen der Massenpaniksimulation hinzuzufügen.

5.1.3 C#-Jobsystem

Die Integrierung des *ECS* codes in verschiedene *Jobs* verlief zu Beginn (mangels an Informationen) ebenfalls etwas problematischer. Allerdings wurde nach kurzer Zeit ein Muster in der Syntax erkannt, welches immer wieder angewandt werden musste, um verschiedene *Jobs* zu erstellen, welche dann dem *C#-Jobsystem* übergeben wurden. Generell wurde überlegt, ob ein Wechsel zwischen den *Interfaces IJobForEach* und *IJobForEachWithEntity* in den einzelnen *Jobs* Sinn ergeben würde (vorausgesetzt *IJobForEachWithEntity* wird nicht explizit benötigt), da diese sonst bei jeder Iteration das passende *Entity* sowie deren *Index* bereit stellen müssen.

Nach kurzer Recherche[20] stellte sich allerdings heraus, dass die Bereitstellung der beiden Parameter, keinen direkten Einfluss auf die Performanz des Projektes nimmt. Die Bereitstellung erfolgt nur “theoretisch” im code, sollte der Entwickler diese Parameter nicht nutzen, werden diese auch während der Ausführung nicht vom *ECS* genutzt.

5.1.4 Burst-Compiler

Der *Burst-Compiler* war die am einfachsten zu integrierende *Unity DOTS* Komponente. Sobald die *Unity DOTS* Erweiterung über den *Package Manager* geladen und in das Projekt integriert wurde, erhielt man automatisch Zugriff auf den *Unity* eigenen *Burst-Compiler*, da dieser standardmäßig in der *Unity* Benutzeroberfläche aktiviert ist. Wie bereits in dieser Arbeit erwähnt, wurden lediglich verschiedene *keywords* (“[BurstCompile]”,”[BurstDiscard]”) benötigt, um dem *Burst-Compiler* mitzuteilen, auf welchen *Jobs*/Methoden er operieren soll.

5.2 Funktionsumfang der entwickelten Simulation

Das folgende Kapitel soll einen allgemeinen Überblick über den Funktionsumfang der erstellten Massenpaniksimulation geben. Die Unterkapitel beziehen sich dabei auf die verschiedenen Aspekte der Simulation. Dieses Kapitel kann als kurze Zusammenfassung angesehen werden, um zu sehen, wie genau die Simulation funktioniert.

5.2.1 Konzertgelände

Zu Beginn der Simulation befindet sich das Konzertgelände in einem “default”-Zustand (siehe Abbildung 4.1 unten). Der Benutzer hat zu diesem Zeitpunkt die Möglichkeit, über das untere *SlotPanel* verschiedene Aktionen auszuwählen, welche das Konzertgelände physisch verändern können (Länge und Rotation). Eine detailliertere Darstellung kann in Kapitel 4.2.4 gefunden werden.
Neben dem *SlotPanel* befindet sich am linken Bildschirmrand ein *InformationPanel*, welches dem Benutzer verschiedene *hotkeys* erläutert. Diese verändern ebenfalls das Konzertgelände, beziehungsweise die ganze *Szene*, in welcher sich das Konzertgelände befindet. Ein Neustart der Simulation kann mit Hilfe des richtigen Knopfes in der unteren rechten Ecke des Bildschirms ausgelöst werden.
Ein verändertes Konzertgelände kann in Abbildung 5.1 eingesehen werden.

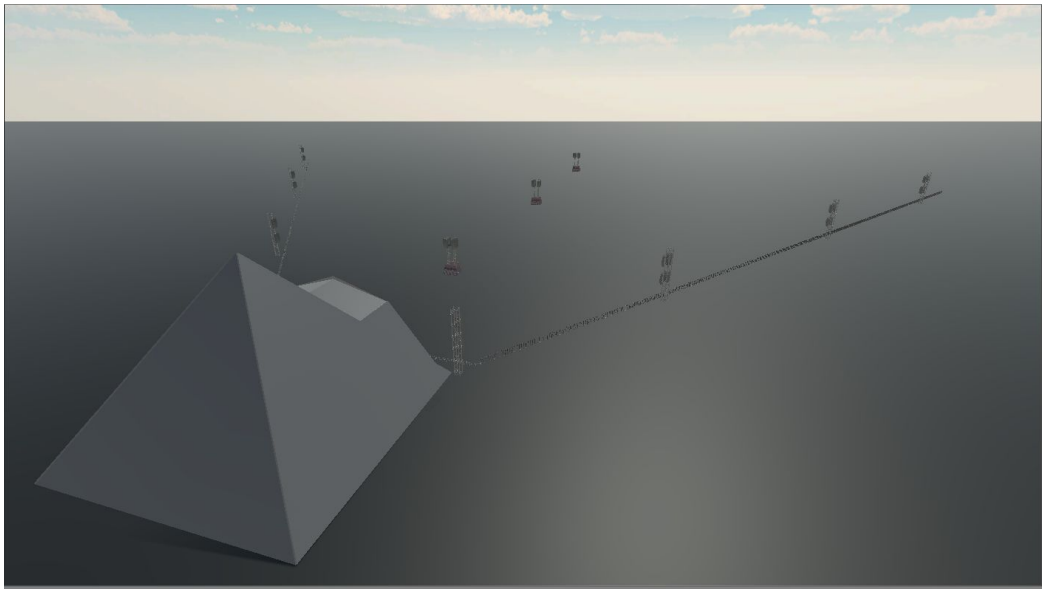
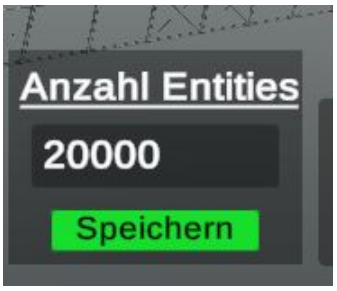
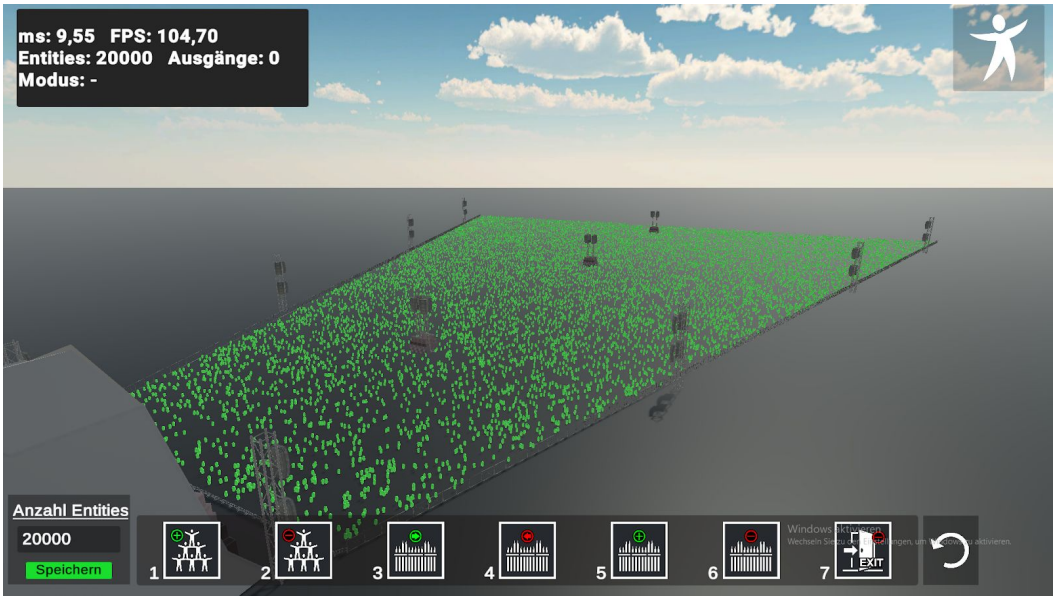


Abbildung 5.1: Ein durch den Benutzer verändertes Konzertgelände

5.2.2 Generierung von Konzertbesuchern

Über das *InputPanel* in der unteren linken Ecke des Bildschirms (Abbildung 5.2) kann der Benutzer eine Anzahl an Konzertbesuchern eingeben, welche er generieren will. Über das *SlotPanel* (Slot 1) kann er diese Anzahl auf dem Konzertgelände generieren (Abbildung 5.3). Die Konzertbesucher können mit der Betätigung von Slot 2 wieder entfernt werden, um eine andere Anzahl an Konzertbesuchern zu generieren.





Oben: Abbildung 5.2 Eingabe der Konzertbesucheranzahl, Unten: Abbildung 5.3 Generierte Konzertbesucheranzahl

5.2.3 Auswahl und platzierung von Panikoptionen

Der Benutzer kann über die *TAB*-Taste ein Menü mit verschiedenen Aktionen einblenden. Diese werden benötigt, um eine Panikreaktion auf dem Konzertgelände auszulösen.

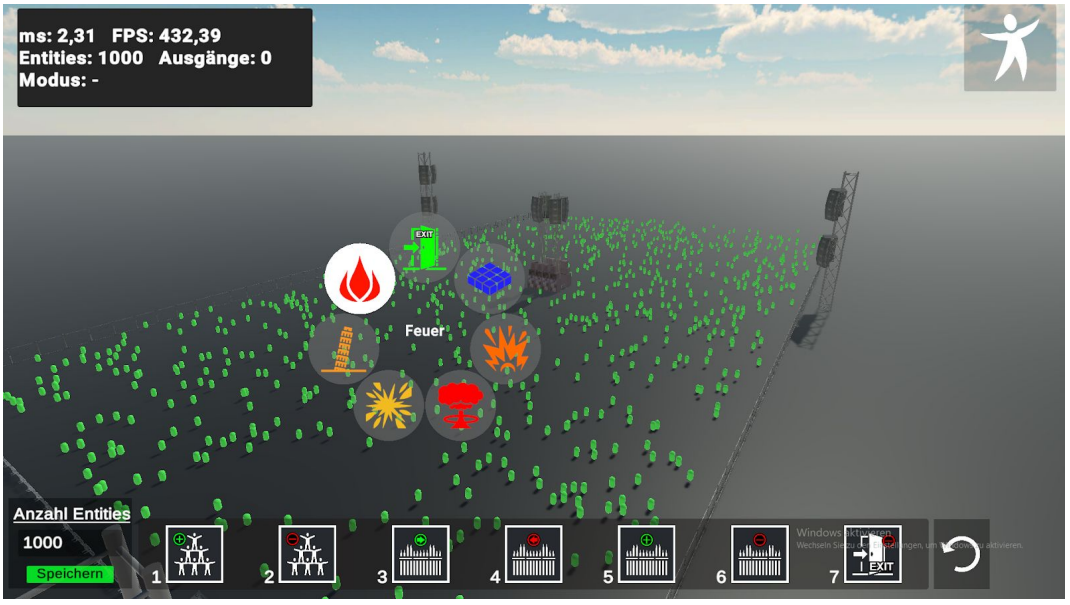


Abbildung 5.4: Panikmenüauswahl der Simulation

Für eine logisch auszuführende Reaktion werden Ausgänge benötigt, diese können in diesem Menü ausgewählt und platziert werden. Außerdem befinden sich in diesem Menü verschiedene “Panikoptionen”, welche die eigentliche Panik in der generierten Menschenmasse auslösen können (Abbildung 5.4). Beispielsweise kann hier eine Feueroption ausgewählt werden, diese wird an der entsprechenden Position platziert, um eine Panikreaktion auf dem Konzertgelände auszulösen. Die Konzertbesucher reagieren automatisch und unmittelbar nach der Platzierung auf dieses Ereignis (Abbildung 5.5).

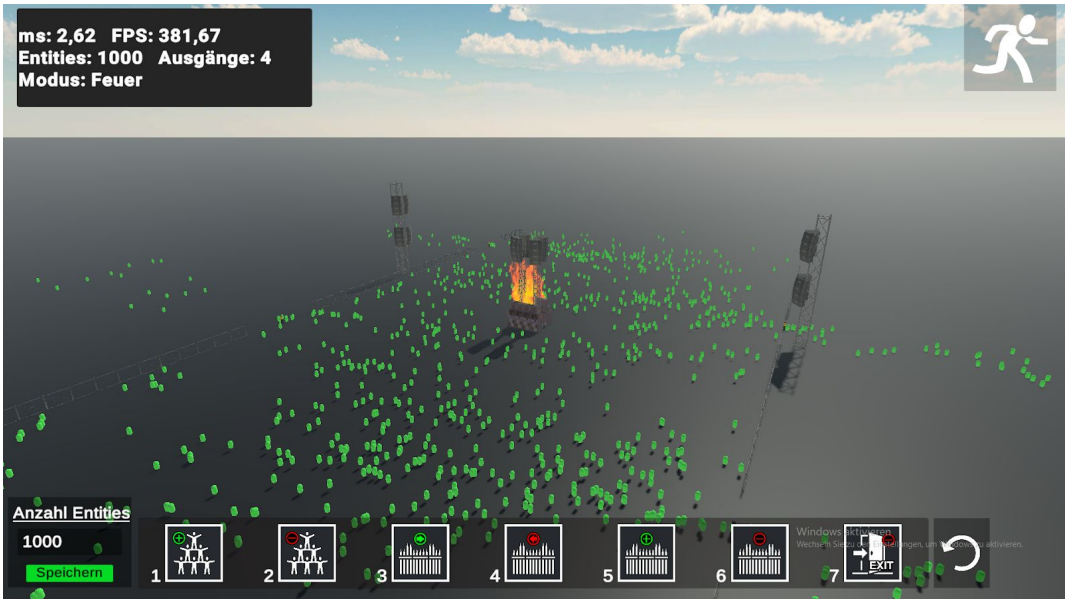


Abbildung 5.5: Platzierte Panikoption (Feuer), Vier Ausgänge platziert

5.3 Qualitätssicherung

Das in dieser Arbeit realisierte Projekt kann wie jede andere Software auch, durch verschiedene *Usability* Methoden geprüft werden. Diese können sich dabei auf die Benutzeroberfläche oder die allgemeine Leistung der Simulation beziehen. Es kann getestet werden, ob ein Benutzer die Simulation vollständig bedienen kann, welche Elemente er dafür betrachtet (*eyetracking*) und wie er sich bei der Verwendung fühlt.

Die hier genannten tests wurden aus zeitgründen in dieser Bachelorarbeit nicht durchgeführt[49].

5.4 Erweiterbarkeit der Simulation

Ein Vorteil von *Unity DOTS* ist die eigentliche Struktur die man während der Entwicklung des Projektes aufbaut. Durch diese ist es jederzeit möglich, neue Ideen und Funktionen in das Projekt zu integrieren. Da *Unity DOTS* außerdem nur eine Erweiterung von der Spiel-Engine *Unity3D* ist, können selbstverständlich auch Faktoren aus der objektorientierten “Unitywelt” geändert und selbstverständlich auch dort neue Ideen und Funktionen hinzugefügt werden. Beispielsweise ist es dort möglich, verschiedene neue Panikereignisse in das *Radial Menu* aufzunehmen und das allgemeine Konzertgelände zu ändern.
Das allgemeine Verhalten der Konzertbesucher kann jederzeit im *PanicSystem* überarbeitet werden. Dabei können die alten Bedingungen überarbeitet oder neue hinzugefügt werden. Allgemein können jederzeit durch das erstellen von neuen *Components* und *Systems* neue Ideen und Funktionen in das Projekt integriert werden. Diese würden dabei keinen Einfluss auf andere Funktionalitäten nehmen, da jede *Component* sowie jedes *System* für sich selbst arbeitet. Eine Idee welche wieder entfernt werden soll, kann jederzeit durch das entfernen der entsprechenden *Components* beziehungsweise der entsprechenden *Systems*, aus dem Projekt entfernt werden.

6. Fazit

Das folgende Kapitel fasst noch einmal das Ergebnis dieser Arbeit zusammen. Außerdem wird über gemachte Erfahrungen berichtet sowie ein kurzer Ausblick dieses Projektes vorgestellt

6.1 Zusammenfassung

Das Ziel dieser Arbeit, eine Standalone-Festival-Massenpaniksimulation auf Basis von *Unity DOTS* zu implementieren, konnte erfolgreich umgesetzt werden. Die Simulation kann als *.exe* Anwendung auf jedem beliebigen Rechner gestartet werden.
Der Benutzer hat anschließend die Möglichkeit, über diese Simulation eine bestimmte Anzahl an *Entities* zu generieren und über das Panikmenü eine Panikreaktion hervorzurufen. Die Anzahl der *Entities* ist dabei stark von der Leistung des Rechners abhängig, auf dem die Simulation ausgeführt wird. Ein durchschnittlicher Rechner sollte allerdings in der Lage sein, eine ausreichende Anzahl an Entities zu generieren (ca. 25.000).
Die in Abbildung 5.3 zu sehende Abbildung wurde als Vergleichswert mit einem 8-Kern AMD Ryzen 7 2700X mit jeweils 3.70GHz aufgenommen. Dieser stellt 16 *Threads* zur Verfügung, wovon nur wenige vom *C#-Jobsystem* verwendet wurden. Abbildung 6.1 visualisiert eine Auslastung dieses Prozessors.

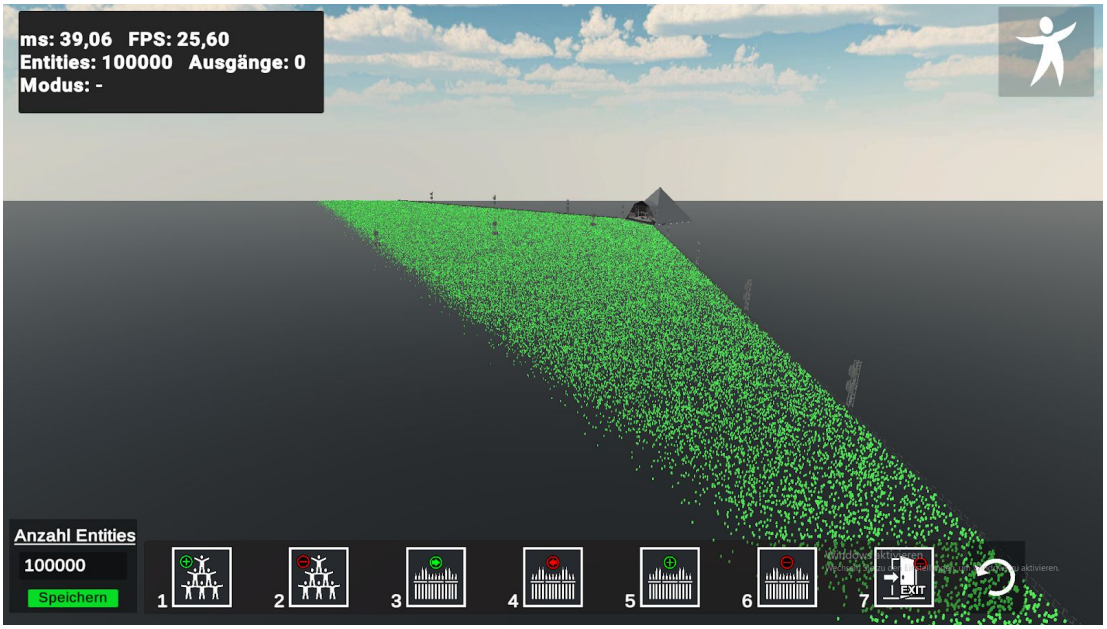


Abbildung 6.1: 100.000 individuelle Konzertbesucher - 30 FPS

6.2 Erfahrungen

Durch mangelnde Informationen seitens *Unity* in Form von Dokumentationen oder ähnlichem, viel die Eingewöhnung zu Beginn nicht leicht aus. Nach längerer Einarbeitung in die *Unity DOTS* Erweiterung und in die datenorientierte Programmierung durch verschiedene "Communityquellen" in Form von Forenbeiträgen, *Unity DOTS* "Discord Servern" sowie verschiedenen *youtubeplaylists* (leider viele davon veraltet, wodurch der code selbst bereits deprecated war), war die Grundlage geschaffen, um das Projekt umzusetzen.
Positiv aufgefallen ist die allgemeine Trennung von Logik und Daten, dadurch konnten viele Ideen und Funktionen testweise in das Projekt implementiert werden. Bei fehlerhaften Ergebnissen, konnten diese dann wieder sehr einfach entfernt werden.
Unity selbst verspricht einen allgemeinen "sauberen" code, welcher bei der Entwicklung mit *Unity DOTS* entstehen soll, während der Entwicklung wurde versucht, diesen Punkt umzusetzen, jedoch funktionierte das nicht in jedem *System*. Sobald ein *System* eine gewisse Größe erreicht hat, fällt es immer schwerer, den einzelnen *Jobs* zu folgen.
Weiterhin positiv aufgefallen ist die wiederverwendbarkeit von code, welcher einmalig entwickelt wurde, laut *Unity* soll das ein weiter Vorteil von *Unity DOTS* sein, während der Entwicklung konnten immer wieder ganze Codeabsätze in Form von *Jobs* übernommen werden. Beispielsweise mussten für die verschiedenen Bewegungen der Konzertbesucher (*running*, *moving*, *dancing*, *idle*) nur vereinzelte Zeilen oder Werte geändert werden, die *Systems* (oder auch *Jobs*) selbst wurden jeweils kopiert und umbenannt.

6.2.1 Bemerkungen

In diesem Unterkapitel werden verschiedene Ideen vorgestellt, welche nur kurz im Projekt präsent waren und deswegen keine Erwähnung in der bisherigen Arbeit fanden.

Die Konzertbesucher nutzen neben dem gehen und dem rennen noch das "tanzen" beziehungsweise das springen als Bewegungsmittel. Diese Fähigkeit wurde, wie bereits beschrieben, mit Hilfe von "Auf und Ab" Bewegungen realisiert. Der Konzertbesucher bewegt sich so lange nach oben, bis er einen bestimmten Punkt erreicht hat, sobald dieser Punkt erreicht wurde, bewegt er sich wieder zurück auf den Boden.
Dieser Prozess wurde zu Beginn mit der "Schwestererweiterung" *Unity Physics* realisiert, welche das Äquivalent zur eigenen *Unity* Physik *Engine* mit Bezug auf *Unity ECS* bildet. Der Nutzer bewegt sich dabei nicht "strikt" auf der Achse nach oben und nach unten, sondern wird mit Hilfe von *Unity Physics* nach oben "geworfen", dies geschieht mit Hilfe von einer Kraft.

Unity Physics besitzt außerdem noch die Fähigkeit, *Entities* über die *Unity* Schwerkraft zu beeinflussen, aus diesem Grund wurden die *Entities* direkt nach dem Kraft Ausstoß mit Hilfe der Schwerkraft wieder nach unten auf den Boden bewegt. So wurde ein sehr realistischer Sprung generiert, welcher allerdings deutlich mehr Leistung in Anspruch genommen hat als eine simple Bewegung auf der *y* Achse. Sobald dieser Prozess auf vielen tausend Konzertbesuchern angewandt wurde, ist ein deutlicher performance Einbruch bemerkbar gewesen. Im Laufe des Projektes hat sich herausgestellt, dass *Unity Physics* nicht für eine größere Menge an *Entities* geeignet ist, weswegen die derzeitige Methode gewählt wurde. Diese Methode ermöglicht eine sehr ähnliche und auch realistische Animation und kostet dabei viel weniger an Leistung.

Unity Physics bietet neben verschiedenen *Kraft* und *Schwerkraft* Modellen auch verschiedene *Collider* Modelle an. Um die Simulation realistischer wirken zu lassen, erhielt jedes Konzertbesucher *Entity* einen eigenen *Collider*, dieser bezweckte eine Interaktion mit anderen Konzertbesuchern und dadurch eine allgemein realistischere Panikreaktion. Wie bereits erwähnt, ist *Unity Physics* leider nicht für größere Mengen an *Entities* ausgelegt, wodurch auch diese Idee bzw. bereits implementierte Möglichkeit, verworfen werden musste. Im Anschluss wurde auch versucht, ein eigenes *Collider System* mit Hilfe von so genannten “axis aligned bounding boxes (AABB)”[50] zu implementieren, dieses wurde aber auch nach kürzerer Zeit verworfen, da es die gleichen Mängel wie die *Unity* Erweiterung *Unity Physics* im Bereich der performance aufwies.

Um *Entities* zu separieren, existierten zu Beginn des Projektes deutlich mehr *Components*, diese wurden genutzt, um einzelne Aktionen/Zustände von *Entities* für andere *Systems* darzustellen. Beispielsweise erhielt ein Konzertbesucher *Entity* eine *Component* sobald dieser in Panik verfallen war, sobald dieser das Konzertgelände wieder verlassen hat wurde dieses wieder entfernt. Da *Components* innerhalb von einem *Job* nur mit Hilfe von einem *CommandBuffer* hinzugefügt, entfernt oder bearbeitet werden können, musste dieser auch in diesem Beispiel das hinzufügen und entfernen von einem so genannten “Panic Tag” übernehmen. Andere *Systems* haben dann auf dieses “Panic Tag” reagiert, indem *Jobs* auf diese *Component* reagiert haben. Die einzelnen *Worker-threads* konnten mit dieser Methode zwar gut arbeiten, da sie die zu erledigenden Aufgaben in den *CommandBuffer* verschoben, dieser musste aber auf dem *Main-thread* abgearbeitet werden, wodurch dieser extrem überlastet war.
Aus diesem Grund wurden verschiedene zusätzliche Datenträger innerhalb von verschiedenen *Components* implementiert (beispielsweise *hasTarget*, *AgentStatus* und *foundFinalExitPoint)* und versucht, *Jobs* von diesen abhängig zu machen.

Beispielsweise befinden sich innerhalb des *PanicJobs* Abfragen, welche nur zulässig sind, sobald *AgentStatus* und *hasTarget* den richtigen Status besitzen.

Punkte an denen Panik ausgebrochen ist, wurden zu Beginn des Projektes von den Konzertbesuchern gemieden, dabei wurde die “Panikposition” dauerhaft in den Konzertbesucher *Entities* gespeichert (in der *AgentComponent*). Das *RunningSystem*, welches für die Panik Bewegungen der Konzertbesucher zuständig ist, bezog sich auf diese Position und überprüfte bei jeder Iteration, ob diese Position mit der derzeitigen Position (zusätzlich einem Radius) übereinstimmt.
Da der *RunningJob* in dem *RunningSystem* für jede kleinste Panik Bewegung eines Konzertbesuchers zuständig ist, musste diese Abfrage für jede Positionsänderung durchgeführt werden. Für wenige hundert Konzertbesucher funktionierte diese Funktion problemlos, sobald aber auf mehrere tausend Konzertbesucher hochskaliert wurde, erhielt man einen starken performance Einbruch.
Aus diesem Grund wurde diese Idee verworfen, da die performance während dieser Arbeit immer Priorität hatte.

6.3 Ausblick

Im folgenden werden Ideen präsentiert, welche in der Zukunft für dieses Projekt umgesetzt werden könnten.

Zusätzliche Hindernisse
Für die weitere Gestaltung der Simulation, sollte der Benutzer zusätzlichen Einfluss auf die Umwelt nehmen können. Beispielsweise sollte er befähigt sein, innerhalb der Simulation verschiedene Gebäude sowie einzelne Absperrungen per *drag and drop* hinzuzufügen und zu verändern.

Mobile Version
Wie bereits in dieser Arbeit erwähnt, ist *Unity* in der Lage ein Projekt für mehrere Plattformen gleichzeitig zur Verfügung zu stellen und diese umzukonvertieren.
Das in dieser Arbeit implementierte Projekt könnte theoretisch auf ein smartphone übertragen werden, hierfür müssten allerdings zuerst die Bedienelemente geändert werden.
Die eigentliche Performanz würde selbstverständlich sinken, da die Leistung eines smartphones nicht mit der Leistung eines PCs vergleichbar ist (in den meisten fällen).

Realistischere Models für die Konzertbesucher

Um die Massenpaniksimulation noch realistischer wirken zu lassen, könnten die *Entities* selbst ein überarbeitetes *Model* in Form von einem Menschenabbild erhalten. Hierbei würde allerdings das Problem entstehen, dass dieses *Model* animiert werden müsste, mit dem *Unity* eigenen *Animator* wäre dieser Vorgang kein Problem, allerdings bezieht sich dieser nur auf *GameObjects* und nicht auf *Entities*. Zur Zeit ist von seitens *Unity* keine offizielle Unterstützung des *Animators* auf *Entities* geplant, weswegen für solche Situationen ein eigener *Animator* implementiert werden müsste. Aus diesem Grund wurde in dieser Arbeit auch eine Kapsel als *Model* gewählt, da diese nicht animiert werden musste.

Eine *Entity* Integration in den *Unity* eigenen *Animator* ist in der Zukunft realistisch, allerdings konzentriert sich *Unity* momentan darauf, die Erweiterung offiziell zu veröffentlichen.

Literaturverzeichnis

[1]
AUTODESK, o. Datum, „maya“, https://www.autodesk.de/products/maya/overview, (26.01.2020)

[2]
Board To Bits Games, 06.11.2015, „Unity Tutorial: Radial Menu (Part 1) from Board to Bits“, https://www.youtube.com/watch?v=WzQdc2rAdZc, (14.12.2019)

[3]
Brackeys, 03.06.2018, „New way of CODING in Unity! ECS Tutorial“, https://youtu.be/_U9wRgQyy6s, (01.08.2019)

[4]
Butterfly World, 06.04.2017, „[BFW]Simple Dynamic Clouds“, https://assetstore.unity.com/packages/tools/particles-effects/bfw-simple-dynamic-clouds-85665, (23.12.2019)

[5]
Clarke, lee, „panic: myth or reality?“, https://web.archive.org/web/20060217135939/http://www.contextsmagazine.org/content_sample_v1-3.php, (25.01.2020)

[6]
Code Monkey, 07.06.2019, „Quadrant System in Unity ECS (Find Target/Obstacle Avoidance/Boids)“, https://www.youtube.com/watch?v=hP4Vu6JbzSo, (20.12.2019)

[7]
Code Monkey, 24.04.2019, „Unity DOTS Explained (ECS, Job System, Burst Compiler)“, https://www.youtube.com/watch?v=Z9-WkwdDoNY, (27.07.2019)

[8]
Code Monkey, 03.05.2019, „Getting Started with ECS in Unity 2019“, https://www.youtube.com/watch?v=ILfUuBLfzGI, (27.07.2019)

[9]
Code Monkey, 10.05.2019, „Getting Started with the Job System in Unity 2019“,
https://www.youtube.com/watch?v=C56bbgtPr_w,
(27.07.2019)

[10]
Dapper Dino, 22.01.2019, „How and Why to use ECS (Entity Component System)“,
https://youtu.be/bop2PYuvRB8,
(01.08.2019)

[11]
ET 0.618, 12.01.2010, „How to determine if a point is in a 2D triangle“,
https://stackoverflow.com/questions/2049582/how-to-determine-if-a-point-is-in-a-2d-triangle/2049712#2049712,
(20.12.2019)

[12]
Fiedler, C, „Simulation statt Massenpanik“,
https://gs-9.com/simulation-statt-massenpanik/,
(26.01.2020)

[13]
Gamemag Creation Studio, 24.01.2017, „Down Town Pack – Lite“,
https://assetstore.unity.com/packages/3d/props/down-town-pack-lite-77864,
(23.12.2019)

[14]
Geraerts, Roland, „Crowd simulation“,
https://www.staff.science.uu.nl/~gerae101/UU_crowd_simulation_software.html,
(28.01.2020)

[15]
Kawetofe, 22.03.2019, „AAA Quality - Road Barricades“,
https://assetstore.unity.com/packages/3d/props/aaa-quality-road-barricades-142191,
(14.12.2019)

[16]
Kawetofe, 25.06.2014, „Medieval Tent Big“,
https://assetstore.unity.com/packages/3d/environments/historic/medieval-tent-big-19023
(23.12.2019)

[17]
Kobra Game Studios, 18.10.2016, „Chainlink Fences“,
https://assetstore.unity.com/packages/3d/chainlink-fences-73107,
(23.12.2019)

[18]
Lague, Sebastian, 26.08.2019, „Coding Adventure: Boids“,
https://www.youtube.com/watch?v=bqtqltqcQhw,
(11.12.2019)

[19]
Max-Planck-Gesellschaft, 15.09.2016, “Wie entsteht Massenpanik in einer Notsituation?”,
https://www.mpg.de/10731913/massenpanik-simulation,
(27.01.2020)

[20]
Nek1113, 03.09.2019, „Difference between IJobForEachWithEntity nad and IJobForEach?“,
https://forum.unity.com/threads/difference-between-ijobforeachwithentity-nad-and-ijobforeach.739169/,
(08.02.2020)

[21]
Next Level 3D, 24.07.2015, „HQ Acoustic system“,
https://assetstore.unity.com/packages/3d/props/electronics/hq-acoustic-system-41886,
(14.12.2019)

[22]
Olprod, OpenLocalizationService, Saisang, 30.10.2019, „Blitfähige und nicht blitfähige Typen“,
https://docs.microsoft.com/de-de/dotnet/framework/interop/blittable-and-non-blittable-types,
(16.12.2019)

[23]
Petzold, Sara, 22.03.2018, „Acht CPU-Kerne in Spielen - Chancen und Probleme in Game Engines“,
https://www.gamestar.de/artikel/multi-core-cpus-intel-will-acht-cpu-kerne-und-mehr-fuer-spiele-salonfaehig-machen,3327673.html,
(10.12.2019)

[24]
Pixel Indie, 17.08.2015, „Metal Floor (Rust Low) Texture“,
https://assetstore.unity.com/packages/2d/textures-materials/metals/metal-floor-rust-low-texture-40351,
(23.12.2019)

[25]
PoliStudio, 26.09.2018, „Aluminium truss systems free“,
https://assetstore.unity.com/packages/3d/environments/industrial/aluminium-truss-systems-free-128696,
(14.12.2019)

[26]
Pryjda, Witold, 28.06.2018, „Experte zum Kerne-Wahnsinn: PC-Hardware geht in die falsche Richtung“,
https://winfuture.de/news,103875.html,
(10.12.2019)

[27]
Rivermill Studios, 20.06.2017, „Disco Nightclub FX“,
https://assetstore.unity.com/packages/3d/environments/disco-nightclub-fx-91701,
(23.12.2019)

[28]
Traufetter, Gerald, 06.10.2008, „Geordnet in den Untergang“,
https://www.spiegel.de/spiegel/print/d-60883197.html,
(25.01.2020)

[29]
Unity Technologies, o. Datum,
„Demos to be converted to Unity DOTS (Data-Oriented Tech Stack)“,
https://github.com/Unity-Technologies/DOTS-training-samples,
(11.12.2019)

[30]
Unity Technologies, 30.08.2018, „Unity Particle Pack“,
https://assetstore.unity.com/packages/essentials/tutorial-projects/unity-particle-pack-127325,
(14.12.2019)

[31]
Unity Technologies, 09.04.2019, „Converting your game to DOTS - Unity at GDC 2019“,
https://youtu.be/QbnVELXf5RQ, (01.08.2019)

[32]
Unity Technologies, o. Datum, „Unity DOTS github repository“,
https://github.com/Unity-Technologies/EntityComponentSystemSamples,
(27.07.2019)

[33]
Unity Technologies, o. Datum, „Official Unity DOTS video tutorials (Entity Component System)“,
https://learn.unity.com/tutorial/entity-component-system,
(04.08.2019)

[34]
Unity Technologies, 30.03.2018, „Unity at GDC - A Data Oriented Approach to Using Component Systems“,
https://www.youtube.com/watch?v=p65Yt20pw0g&feature=youtu.be&t=660,
(25.01.2020)

[35]
Vertex Studio, 02.06.2017, „Free Laptop“,
https://assetstore.unity.com/packages/3d/props/electronics/free-laptop-90315,
(23.12.2019)

[36]
zdf, 07.01.2020, „Mindestens 56 Tote bei Massenpanik“,
https://www.zdf.de/nachrichten/heute/soleimani-trauerzug-im-iran-mindestens-50-tote-bei-massenpanik-100.html,
(26.01.2020)

[37]
o. V., o. Datum, „Unglück bei der Loveparade 2010“,
https://de.wikipedia.org/wiki/Ungl%C3%BCck_bei_der_Loveparade_2010,
(10.12.2019)

[38]
o. V, o. Datum, „Unity-DOTS“,
https://unity.com/de/dots,
(11.12.2019)

[39]
o. V, o. Datum, „TextMeshPro“,
https://assetstore.unity.com/packages/essentials/beta-projects/textmesh-pro-84126,
(12.12.2019)

[40]
o. V, o. Datum, „Wie kommt es zu einer Massenpanik?“,
https://www.philognosie.net/wissen-technik/wie-kommt-es-zu-einer-massenpanik,
(26.01.2020)

[41]
o. V, o. Datum, „Massenpanik in Mekka 2015“,
https://de.wikipedia.org/wiki/Massenpanik_in_Mekka_2015,
(26.01.2020)

[42]
o. V, o. Datum, „Massenpanik“,
https://de.wikipedia.org/wiki/Massenpanik#cite_note-:0-10,
(25.01.2020)

[43]
o. V, o. Datum, „Golaem Crowd“,
https://en.wikipedia.org/wiki/Golaem_Crowd#cite_note-Golaem:_la_start-up_rennaise_s%27invite_dans_Game_of_Thrones-28,
(26.01.2020)

[44]
o. V, o. Datum, „GoT_S7“,
http://golaem.com/sites/default/files/got7_frozenLake.jpg,
(26.01.2020)

[45]
o. V, o. Datum, „accu:rate“,
https://gs-9.com/wp-content/uploads/2016/09/accurate_hanse-sail_30grad-2.jpg,
(26.01.2020)

[46]
o. V, o. Datum, „original-1510580428“,
https://www.mpg.de/11631635/original-1510580428.jpg?t=eyJ3aWR0aCI6MzQxLCJvYmpfaWQiOjExNjMxNjM1fQ==--621d9d813ded563695b582d8a88cb0ffc4f43161,
(27.01.2020)

[47]
o. V, o. Datum, „miro-medium“,
https://miro.medium.com/max/671/0*6521b8iIuKKa33Ky,
(27.01.2020)

[48]
o. V, o. Datum, „UUCS_crowd_simulation_plugin_cropped“,
http://www.staff.science.uu.nl/~gerae101/images/UUCS_crowd_simulation_plugin_cropped.jpg,
(28.01.2020)

[49]
o. V, o. Datum, „Usability-Test“,
https://de.wikipedia.org/wiki/Usability-Test,
(11.02.2020)

[50]
o. V, o. Datum, „Hüllkörper“,
https://de.wikipedia.org/wiki/H%C3%BCllk%C3%B6rper,
(12.02.2020)

Anhang

[A1]
Die Methode ShowFPS() aus dem UIHandler Skript
12.12.2019

```
/// <summary>
/// Handle ms and fps calculation. Assign this information to infoText.
/// </summary>
private void ShowFPS()
{
    deltaTime += (Time.unscaledDeltaTime - deltaTime) * 0.1f;
    ms = deltaTime * 1000.0f;
    fps = 1.0f / deltaTime;

    infoText.text = "ms: " + ms.ToString("F2") + "    FPS: " + fps.ToString("F2") + "\n" + "Entities: " + entityAmount + "
            Ausgänge: " + exitsAmount + "\n" + "Modus: " + mode;
}
```

[A2]
Die Methode ValidateInput() aus dem InputWindow Skript
12.12.2019

```
    /// <summary>
    /// Call method that checks valid int input.
    /// </summary>
    private void ValidateInput()
    {
        inputField.characterLimit = 7;
        inputField.onValidateInput = (string text, int charIndex, char addedChar) =>
        {
            return ValidateCharacter(validCharacters, addedChar);
        };
    }
```

[A3]
Die Methode CheckEnterKeyInput() aus dem InputWindow Skript
12.12.2019

```
    /// <summary>
    /// Checks if the user pressed one of the enter buttons to save his input. When pressed, call SaveValue() Method.
    /// </summary>
    private void CheckEnterKeyInput()
    {
        if (Input.GetKeyDown("return") || Input.GetKeyDown("enter"))
        {
            SaveValue();
        }
    }
```

[A4]
Die Methode SaveValue() aus dem InputWindow Skript
12.12.2019

```
    /// <summary>
    /// Method that checks if the save button was pressed.
    /// If the button was pressed, save the new value.
    /// </summary>
    public void SaveValue()
```

```
{
    UnitSpawnerProxy.instance.AmountToSpawn = int.Parse(inputField.text);
}
```

[A5]
Die Methode OnButtonHoldingDownRotateOut() aus dem ItemClickHandler 12.12.2019

```
/// <summary>
/// Rotates the two pivot barriers on the left and on the right, so that the user can change the festival area.
/// There is a limit of 60°, so the User cannot rotate infinitely.
/// </summary>
private void OnButtonHoldingDownRotateOut()
{
    if (barrierLeftWithPivot.transform.rotation.y <= .5f && barrierRightWithPivot.transform.rotation.y >= -.5f)
    {
        // Compare the actual rotation of the first pivot to the Unity own rotation value.
        // If -60°/60° is not reached, rotate more.
        barrierLeftWithPivot.transform.Rotate(Vector3.up * Time.deltaTime * (rotationSpeed / 4), Space.World);
        barrierRightWithPivot.transform.Rotate(Vector3.down * Time.deltaTime * (rotationSpeed / 4), Space.World);

        // Control the last two Sound Systems
        // Otherwise they will exists on an empty festival area where no people are
        if (barrierLeftWithPivot.transform.rotation.y <= .25f && barrierRightWithPivot.transform.rotation.y >= -.25f)
        {
            if (penultimateSoundSystem.activeInHierarchy == true && lastSoundSystem.activeInHierarchy == true)
            {
                penultimateSoundSystem.SetActive(true);
                lastSoundSystem.SetActive(true);
            }
        }
        else
        {
            penultimateSoundSystem.SetActive(false);
            lastSoundSystem.SetActive(false);
        }
    }
    else
    {
        // If it is, throw a log message.
        Debug.Log("You can only stretch the area out, until you reach a degree of 60°.");
    }
}
```

[A6]
Die Methode OnButtonHoldingDownRotateIn() aus dem ItemClickHandler 12.12.2019

```
/// <summary>
/// Rotates the two pivot barriers on the left and on the right, so that the user can change the festival area.
/// There is a limit of 0.0°, so the User cannot rotate infinitely.
/// </summary>
private void OnButtonHoldingDownRotateIn()
{
    if (barrierLeftWithPivot.transform.rotation.y <= .51f && barrierRightWithPivot.transform.rotation.y >= -.51f
        && barrierLeftWithPivot.transform.rotation.y > 0.0005f && barrierRightWithPivot.transform.rotation.y <
-0.0005f)
    {
        // Compare the actual rotation of the first pivot to the Unity own rotation value.
        // Rotate with inside direction if the barriers are in the given area.
        // Stop with an offset of 0.0005. This helps for stopping on an rotation of Vector3.zero
        barrierLeftWithPivot.transform.Rotate(Vector3.down * Time.deltaTime * (rotationSpeed / 4), Space.World);
```

```
        barrierRightWithPivot.transform.Rotate(Vector3.up * Time.deltaTime * (rotationSpeed / 4), Space.World);

        // Control the last two Sound Systems
        // Otherwise they will exists on an empty festival area where no people are
        if (barrierLeftWithPivot.transform.rotation.y <= .25f && barrierRightWithPivot.transform.rotation.y >= -.25f)
        {
            if (lastSoundSystem.transform.parent.Find("Sound System_3").gameObject.activeInHierarchy)
            {
                penultimateSoundSystem.SetActive(true);
                lastSoundSystem.SetActive(true);
            }
        }
        else
        {
            penultimateSoundSystem.SetActive(false);
            lastSoundSystem.SetActive(false);
        }
    }
    else
    {
        // If they are not inside the given area
        Debug.Log("You can only stretch the area in, until you reach a degree of 0.0°.");
    }
}
```

[A7]
Die Methode AddNewSideBarrier() aus dem ItemClickHandler
12.12.2019

```
/// <summary>
/// Adds a new barrier to both of the starting barriers.
/// The barriers exists already, they are disabled and need to be enabled.
/// There is an increaseCounter which counts all the time for knowing, how many barriers are added.
/// Extra: Set an extra Sound System for each additional Barrier.
/// </summary>
private void AddNewSideBarrier()
{
    if (!InputWindow.instance.inputField.isFocused)
    {
        if (instance.increaseCounter == additionalBarriersLeft.Length)
        {
            // The counter has the same value like the length of the additionalBarriers Array. The Next Barrier would
lead to NullPointer
            // Throw warning/log and return
            Debug.LogWarning("Limit reached!");
            Debug.Log("The barrier limit for this simulation was reached.");
            return;
        }

        // Set the next Barriers and sound system to active
        additionalBarriersLeft[instance.increaseCounter].SetActive(true);
        additionalBarriersRight[instance.increaseCounter].SetActive(true);
        additionalSoundSystems[instance.increaseCounter].SetActive(true);

        // Set spawnpoint 1 position to spawnpoint additionalBarriersLeft[increaseCounter] position
        // The DOTS System will be able now to spawn agents in the new area
        spawnPlaceLeft.transform.position = additionalSpawnObjectsLeft[instance.increaseCounter +
1].transform.position;
        spawnPlaceRight.transform.position = additionalSpawnObjectsRight[instance.increaseCounter +
1].transform.position;
        #region Debug Barriers
        //Debug.Log(spawnPlaceLeft.transform.position);
```

```
        //Debug.Log(spawnPlaceRight.transform.position);
        Debug.DrawLine(spawnPlaceLeft.transform.position, new Vector3(spawnPlaceLeft.transform.position.x,
spawnPlaceLeft.transform.position.y + 50f, spawnPlaceLeft.transform.position.z), Color.blue, 1f);
        Debug.DrawLine(spawnPlaceRight.transform.position, new Vector3(spawnPlaceRight.transform.position.x,
spawnPlaceRight.transform.position.y + 50f, spawnPlaceRight.transform.position.z), Color.blue, 1f);
        #endregion // Debug Barriers
        if (!(instance.increaseCounter == additionalBarriersLeft.Length))
        {
            // Only increase the counter when the limit is not reached yet.
            instance.increaseCounter++;
        }
    }
  }
```

[A8]
Die Methode RemoveNewSideBarrier() aus dem ItemClickHandler
12.12.2019

```
   /// <summary>
  /// Removes Side Barriers and Sound Systems.
  /// Increase counter helps to identify how many Barriers are active.
  /// </summary>
  private void RemoveNewSideBarrier()
  {
     if (!InputWindow.instance.inputField.isFocused)
     {
        if (instance.increaseCounter != 0) // Prevent of getting the -1th Object of the Arrays != 0
        {
           // Set the Objects to false, to disable the GameObjects and going back with the barriers and the Sound
Systems
           additionalBarriersLeft[instance.increaseCounter - 1].SetActive(false);
           additionalBarriersRight[instance.increaseCounter - 1].SetActive(false);
           additionalSoundSystems[instance.increaseCounter - 1].SetActive(false);

           if (instance.increaseCounter == 1)
           {
              // Change the spawnplace back
              spawnPlaceLeft.transform.position = originalStartingSpawnObjectPositionLeft;
              spawnPlaceRight.transform.position = originalStartingSpawnObjectPositionRight;
           }
           else
           {
              // Change the spawnplace back
              spawnPlaceLeft.transform.position = additionalSpawnObjectsLeft[instance.increaseCounter -
1].transform.position;
              spawnPlaceRight.transform.position = additionalSpawnObjectsRight[instance.increaseCounter -
1].transform.position;
           }
           #region Debug Barriers
           //Debug.Log(spawnPlaceLeft.transform.position);
           //Debug.Log(spawnPlaceRight.transform.position);
           Debug.DrawLine(spawnPlaceLeft.transform.position, new Vector3(spawnPlaceLeft.transform.position.x,
spawnPlaceLeft.transform.position.y + 50f, spawnPlaceLeft.transform.position.z), Color.red, 1f);
           Debug.DrawLine(spawnPlaceRight.transform.position, new Vector3(spawnPlaceRight.transform.position.x,
spawnPlaceRight.transform.position.y + 50f, spawnPlaceRight.transform.position.z), Color.red, 1f);
           #endregion // Debug Barriers
           // Decrease the counter
           instance.increaseCounter--;

        }
        else if (instance.increaseCounter <= 0)
        {
```

```
            // Throw Warning/log when increaseCounter is 0, return
            Debug.LogWarning("Limit reached!");
            Debug.Log("The minimum limit for the barriers is one.");
            return;
        }
    }
}
```

[A9]
Die Methode CheckKeys() aus dem UIHandler
13.12.2019

```
/// <summary>
/// A method that takes care of different Keycodes that can be pressed from the user.
/// </summary>
private void CheckKeys()
{
    if (Input.GetKeyDown(KeyCode.Escape)) // Disable active mode and update the UI
    {
        UIHandler.instance.mode = "-";
        Actions.instance.createExits = false;
        Actions.instance.fallingTruss = false;
        Actions.instance.smallGroundExplosion = false;
        Actions.instance.mediumGroundExplosion = false;
        Actions.instance.bigGroundExplosion = false;
        Actions.instance.dropSoundSystem = false;
    }
    else if (Input.GetKeyDown(KeyCode.O)) // Enable/Disable LOD Function
    {
        EnableOrDisableLODFunction();
    }
    else if (Input.GetKeyDown(KeyCode.N)) // Enable/Disable NightMode
    {
        EnableOrDisableNightMode();
    }
    else if (Input.GetKeyDown(KeyCode.L)) // Enable/Disable Effects (Lights, Displays, Smoke)
    {
        EnableOrDisableEffects();
    }
}
```

[A10]
Die Methode HandleCamera() aus dem UIHandler
13.12.2019

```
/// <summary>
/// Controls Camera with WASD.
/// Look around with right mouse click (holded).
/// Zoom up and down with mouse wheel.
/// </summary>
private void HandleCamera()
{
    // Control Camera with WASD
    if (Input.GetKey(KeyCode.W) /*|| Input.mousePosition.y >= Screen.height - panBorderThickness*/)
    {
        transform.Translate(Vector3.forward * cameraSpeed * Time.deltaTime);
    }
    if (Input.GetKey(KeyCode.A) /*|| Input.mousePosition.x <= panBorderThickness*/)
    {
        transform.Translate(Vector3.left * cameraSpeed * Time.deltaTime);
    }
```

```
    if (Input.GetKey(KeyCode.S) /*|| Input.mousePosition.y <= panBorderThickness*/)
    {
        transform.Translate(Vector3.back * cameraSpeed * Time.deltaTime);
    }
    if (Input.GetKey(KeyCode.D) /*|| Input.mousePosition.x >= Screen.width - panBorderThickness*/)
    {
        transform.Translate(Vector3.right * cameraSpeed * Time.deltaTime);
    }

    // Look around with Right Mouse click
    // Use Unity own Axis
    if (Input.GetMouseButton(1))
    {
        yaw += lookSpeedH * Input.GetAxis("Mouse X");
        pitch += lookSpeedV * Input.GetAxis("Mouse Y");

        transform.eulerAngles = new Vector3(pitch, yaw, 0.0f);
    }

    // Zoom up and down with mouse wheel
    transform.Translate(0, Input.GetAxis("Mouse ScrollWheel") * zoomSpeed, 0, Space.Self);
}
```

[A11]
Die Methode WindowCheck() aus dem UIHandler
13.12.2019

```
 /// <summary>
/// Method that takes care of N and I keys to enable/disable the information or statistic window.
/// Disable when window is active in Hierarchy. Otherwise Enable.
/// </summary>
private void WindowCheck()
{
    if (Input.GetKeyDown(KeyCode.H))
    {
        if (informationWindowPanel.activeInHierarchy)
        {
            // Information Window is enabled
            informationWindowPanel.SetActive(false);
        }
        else
        {
            // Information Window is disabled
            informationWindowPanel.SetActive(true);
        }
    }

    if (Input.GetKeyDown(KeyCode.I))
    {
        if (statisticWindowPanel.activeInHierarchy)
        {
            // Information Window is enabled
            statisticWindowPanel.SetActive(false);
        }
        else
        {
            // Information Window is disabled
            statisticWindowPanel.SetActive(true);
        }
    }
}
```

[A12]
Die Methode EnableOrDisableLODFunction() aus dem UIHandler
13.12.2019

```
    /// <summary>
    /// Takes all barriers on the left and on the right and disable/enable all LOD components.
    /// </summary>
    private void EnableOrDisableLODFunction()
    {
        LODGroup[] lodsLeft = barrierLeftWithPivot.GetComponentsInChildren<LODGroup>(); // Get all LODs from left
side
        LODGroup[] lodsRight = barrierRightWithPivot.GetComponentsInChildren<LODGroup>(); // Get all LODs from
right side

        // Loop throught them and disable/enable them
        foreach (LODGroup lodGroup in lodsLeft)
        {
            if (lodGroup.enabled)
            {
                lodGroup.enabled = false;
            }
            else
            {
                lodGroup.enabled = true;
            }
        }

        foreach (LODGroup lodGroup in lodsRight)
        {
            if (lodGroup.enabled)
            {
                lodGroup.enabled = false;
            }
            else
            {
                lodGroup.enabled = true;
            }
        }
    }
```

[A13]
Die Methode EnableOrDisableNightMode() aus dem UiHandler
13.12.2019

```
    /// <summary>
    /// Method that Enable/Disable Night mode.
    /// </summary>
    private void EnableOrDisableNightMode()
    {
        // Day: Skyboxes Mega Pack 1/7/7, DL.color = DCDED4
        // Night: Night Skyboxes pack 2/2/2, DL.color = 3146BB
        if (RenderSettings.skybox == skyboxDay)
        {
            // Current Skybox is in Day mode
            RenderSettings.skybox = null;
            RenderSettings.skybox = skyboxNight;
            directionalLight.color = new Color32(0x31, 0x46, 0xBB, 0xFF);
        }
        else
        {
```

```
        // Current Skybox is in Night mode
        RenderSettings.skybox = skyboxDay;
        directionalLight.color = new Color32(0xDC, 0xDE, 0xD4, 0xFF);
    }
}
```

[A14]
Die Methode EnableOrDisableEffects() aus dem UIHandler
13.12.2019

```
/// <summary>
/// Method that Enable/Disable different effects.
/// </summary>
private void EnableOrDisableEffects()
{
    // Truss
    if (trussLights.activeInHierarchy)
    {
        trussLights.SetActive(false);
        effectsEnabled = false;
    }
    else
    {
        trussLights.SetActive(true);
        effectsEnabled = true;
    }

    // Smoke
    if (smoke.activeInHierarchy)
    {
        smoke.SetActive(false);
    }
    else
    {
        smoke.SetActive(true);
    }

    // Displays
    if (displays.activeInHierarchy)
    {
        displays.SetActive(false);
    }
    else
    {
        displays.SetActive(true);
    }

    // First Sound System
    if (frontSoundSystemLight.activeInHierarchy)
    {
        frontSoundSystemLight.SetActive(false);
    }
    else
    {
        frontSoundSystemLight.SetActive(true);
    }

    // Lights of user created Sound Systems
    foreach (GameObject soundSystem in userCreatedSoundSystems)
    {
        GameObject lightObject = soundSystem.transform.Find("Lights").gameObject;
        Debug.Log(lightObject.name);
```

```
            if (lightObject.activeInHierarchy)
            {
                lightObject.SetActive(false);
            }
            else
            {
                lightObject.SetActive(true);
            }
        }

        // Lights of additional Sound Systems
        foreach (GameObject soundSystem in additionalSoundSystems)
        {
            GameObject lightObject = soundSystem.transform.Find("Lights").gameObject;
            if (lightObject.activeInHierarchy)
            {
                lightObject.SetActive(false);
            }
            else
            {
                lightObject.SetActive(true);
            }
        }
    }
```

[A15]
Die Methode ReloadActualScene() aus dem ReloadScene Skript
13.12.2019

```
    /// <summary>
    /// Method that reloads the actual main scene.
    /// </summary>
    public void ReloadActualScene()
    {
        DestroyAllEntities();
        StartCoroutine("DisposeAllWorlds");
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
```

[A16]
Die Methode DestroyAllEntities() aus dem ReloadScene Skript
13.12.2019

```
    /// <summary>
    /// A method that destroys all entities in this world.
    /// </summary>
    private void DestroyAllEntities()
    {
        Unity.Entities.EntityManager entityManager = Unity.Entities.World.Active.EntityManager;
        entityManager.DestroyEntity(entityManager.UniversalQuery);
    }
```

[A17]
Der IEnumerator DisposeAllWorlds() aus dem ReloadScene Skript
13.12.2019

```
    /// <summary>
```

```
    /// A Coroutine that waits for the end of the frame to prevent that new default systems are created when they
aren't needed.
    /// </summary>
    /// <returns></returns>
    private IEnumerator DisposeAllWorlds()
    {
        yield return new WaitForEndOfFrame();
        Unity.Entities.World.DisposeAllWorlds();
        Unity.Entities.DefaultWorldInitialization.Initialize("Default World", false);
    }
```

[A18]
Die Methode UpdateStatusIcon() aus dem UIHandler
13.12.2019

```
    /// <summary>
    /// Method that react on changes of the actionPlaced variable. Changes the status image.
    /// </summary>
    private void UpdateStatusIcon()
    {
        if (Actions.instance.actionPlaced)
        {
            statusImage.sprite = panicMode;
        }
        else
        {
            statusImage.sprite = normalMode;
        }
    }
```

[A19]
Die Coroutine AnimateButtons() aus dem RadialMenu
14.12.2019

```
    /// <summary>
    /// Spawns the buttons and set the calculated circle position to each of the buttons.
    /// Sets some individual settings to each button and waits until spawning another button. -> Small Animation
"Spawn effect".
    /// </summary>
    /// <param name="uiHandler">For accessing different settings which are set in the Inspector.</param>
    /// <returns>Wait until spawning a new button.</returns>
    IEnumerator AnimateButtons(UIHandler uiHandler)
    {
        for (int i = 0; i < uiHandler.options.Length; i++)
        {
            // Create the button, set as a child of the Radial Menu
            RadialButton radialButton = Instantiate(buttonPrefab) as RadialButton;
            radialButton.transform.SetParent(transform, false);

            // Distance around the circle
            float theta = (2 * Mathf.PI / uiHandler.options.Length) * i;

            // Convert theta to x/y position
            float xPos = Mathf.Sin(theta);
            float yPos = Mathf.Cos(theta);

            // Set the button position
            radialButton.transform.localPosition = new Vector3(xPos, yPos, 0.0f) * buttonDistance;

            // Set individual button settings
```

```
            radialButton.circle.color = uiHandler.options[i].color;
            radialButton.icon.sprite = uiHandler.options[i].sprite;
            radialButton.title = uiHandler.options[i].title;
            radialButton.radialMenu = this;

            // Animation
            yield return new WaitForSeconds(.07f);
        }
    }
```

[A20]
Die Methode CheckRadialMenuSelection() aus dem RadialMenu Skript
14.12.2019

```
    /// <summary>
    /// React on selected Buttons, enable/disable bools.
    /// </summary>
    public void CheckRadialMenuSelection()
    {
        if (selected) // The selected Radial Menu Button
        {
            // Set default Radial Menu title
            RadialMenuSpawner.instance.radialMenu.label.text = "Actions";
            RadialMenuSpawner.instance.updatedMenuText = "Actions";

            if (selected.title == "Create Exits")
            {
                // Create Exits Button choosen
                actions.createExits = true;
                actions.actionEnabled = false;
                actions.fallingTruss = false;
                actions.trussHasFallen = false;
                actions.dropSoundSystem = false;
                actions.smallGroundExplosion = false;
                actions.mediumGroundExplosion = false;
                actions.bigGroundExplosion = false;
                actions.fire = false;
                UIHandler.instance.mode = "Create Exits";

                // Set bool true that will be used in the animator script
                UIHandler.instance.enableTrussArrows = false;
                UIHandler.instance.enableSoundSystemArrows = false;

                // Set bool in UI Handler (Synch Point) -> Triggers DOTS script to enable the panic System
            }
            else if (selected.title == "Small Explosion")
            {
                // Small Explosion Button choosen
                actions.smallGroundExplosion = true;
                actions.actionEnabled = true;

                actions.mediumGroundExplosion = false;
                actions.bigGroundExplosion = false;
                actions.createExits = false;
                actions.fallingTruss = false;
                actions.trussHasFallen = false;
                actions.dropSoundSystem = false;
                actions.fire = false;
                UIHandler.instance.mode = "Small Explosions";

                // Set bool true that will be used in the animator script
                UIHandler.instance.enableTrussArrows = false;
```

```
    UIHandler.instance.enableSoundSystemArrows = false;
}
else if (selected.title == "Medium Explosion")
{
    // Medium Explosion Button choosen
    actions.mediumGroundExplosion = true;
    actions.actionEnabled = true;

    actions.smallGroundExplosion = false;
    actions.bigGroundExplosion = false;
    actions.createExits = false;
    actions.fallingTruss = false;
    actions.trussHasFallen = false;
    actions.dropSoundSystem = false;
    actions.fire = false;
    UIHandler.instance.mode = "Medium Explosions";

    // Set bool true that will be used in the animator script
    UIHandler.instance.enableTrussArrows = false;
    UIHandler.instance.enableSoundSystemArrows = false;
}
else if (selected.title == "Big Explosion")
{
    // Big Explosion Button choosen
    actions.bigGroundExplosion = true;
    actions.actionEnabled = true;

    actions.mediumGroundExplosion = false;
    actions.smallGroundExplosion = false;
    actions.createExits = false;
    actions.fallingTruss = false;
    actions.trussHasFallen = false;
    actions.dropSoundSystem = false;
    actions.fire = false;
    UIHandler.instance.mode = "Big Explosion";

    // Set bool true that will be used in the animator script
    UIHandler.instance.enableTrussArrows = false;
    UIHandler.instance.enableSoundSystemArrows = false;
}
else if (selected.title == "Falling Truss")
{
    // Falling Truss Button choosen
    actions.fallingTruss = true;
    actions.actionEnabled = true;

    actions.createExits = false;
    actions.dropSoundSystem = false;
    actions.smallGroundExplosion = false;
    actions.mediumGroundExplosion = false;
    actions.bigGroundExplosion = false;
    actions.fire = false;
    UIHandler.instance.mode = "Falling Truss";

    // Set bool true that will be used in the animator script
    UIHandler.instance.enableTrussArrows = true;
    UIHandler.instance.enableSoundSystemArrows = false;
}
else if (selected.title == "Drop Sound System")
{
    // Drop Sound System Button choosen
    actions.dropSoundSystem = true;
    actions.fallingTruss = false;
```

```
            actions.trussHasFallen = false;
            actions.createExits = false;
            actions.smallGroundExplosion = false;
            actions.mediumGroundExplosion = false;
            actions.bigGroundExplosion = false;
            actions.fire = false;
            UIHandler.instance.mode = "Create Sound System";

            // Set bool true that will be used in the animator script
            UIHandler.instance.enableTrussArrows = false;
            UIHandler.instance.enableSoundSystemArrows = false;
        }
        else if (selected.title == "Fire")
        {
            // Fire Button choosen
            actions.fire = true;
            actions.actionEnabled = true;

            actions.dropSoundSystem = false;
            actions.fallingTruss = false;
            actions.trussHasFallen = false;
            actions.createExits = false;
            actions.smallGroundExplosion = false;
            actions.mediumGroundExplosion = false;
            actions.bigGroundExplosion = false;
            UIHandler.instance.mode = "Fire";

            // Set bool true that will be used in the animator script
            UIHandler.instance.enableTrussArrows = false;
            UIHandler.instance.enableSoundSystemArrows = true;
        }
    }
    // A button was choosen so the Radial Menu is not needed anymore
    Destroy(gameObject);
}
```

[A21]
Die UnitSpawnerComponent
16.12.2019

```
/// <summary>
/// The converted Scene GameObject (ConvertToEntity attached) gets this component to spawn the entitys.
/// </summary>
public struct UnitSpawnerComponent : IComponentData
{
    public int AmountToSpawn; // Amount to spawn
    public Entity Prefab; // Gameobject/Prefab -> converted into Entity -> accessed and copyed
}
```

[A22]
Die BorderComponent
16.12.2019

```
/// <summary>
/// This Component represents the Border of the actual festival area.
/// </summary>
public struct BorderComponent : IComponentData
{
    // 4 spawn positions
    public float3 frontRight;
    public float3 frontLeft;
```

```
    public float3 backRight;
    public float3 backLeft;
}
```

[A23]
Die Input Component
16.12.2019

```
/// <summary>
/// Each agent get access on Input information.
/// </summary>
public struct InputComponent : IComponentData
{
    public bool keyOnePressedDown; // Bool for key "1" (when pressed down) -> manager reacts and adds a tag component to crowd entity
    public bool keyOnePressedUp; // Bool for key "1" (when pressed up) -> spawns agents when pressing key "1" up (UnitSpawnerSystem)
    public bool keyTwoPressedUp;
    public bool keyThreePressedUp; // Bool for key "3" (when pressed up) after Barriers rotated outside
    public bool keyFourPressedUp; // Bool for key "4" (when pressed up) after Barriers rotated inside
    public bool keyFivePressedUp;
    public bool keySixPressedUp;
    public bool keySevenPressedUp;
}
```

[A24]
Die Agent Component
17.12.2019

```
/// <summary>
/// Describes the current AgentStatus.
/// </summary>
public enum AgentStatus
{
    Idle = 0, // standing still
    Moving = 1, // normal moving
    Dancing = 2, // jumping/dancing "animation"
    Running = 3 // fast moving if panic spot appears
}

/// <summary>
/// Each Agent gets a AgentComponent.
/// </summary>
public struct AgentComponent : IComponentData
{
    public bool hasTarget; // Agent has a target
    public float3 target; // The actual target
    public AgentStatus agentStatus; // The current status of the agent
    public bool exitPointReached; // Agent has reached the user generated exit spot
    public bool foundTemporaryNewRandomPosition; // Agent found a first random generated position
    public bool foundFinalExitPoint; // Agent found the final exit position
    public bool marked; // Agent was on a exit spot (or on the way to it) but turned around to find a new exit spot
    public float fleeProbability; // The Probability to turn around when running to an exit spot
}
```

[A25]
Die Move Speed Component
17.12.2019

```
/// <summary>
/// Each agent get their own moveSpeed Component to simulate the different speed attributes of humans.
/// </summary>
public struct MoveSpeedComponent : IComponentData
{
   public float moveSpeed; // Speed for normal moving state
   public float runningSpeed; // Speed for panic situation
   public float jumpSpeed; // Spped for jumping
   public float panicJumpSpeed; // Speed for panic Jumping
}
```

[A26]
Die Dummy Component
17.12.2019

```
/// <summary>
/// Used to create ExitEntitys.
/// </summary>
public struct DummyComponent : IComponentData
{
}
```

[A27]
Die Exit Component
17.12.2019

```
/// <summary>
/// Component for each exit entity.
/// </summary>
public struct ExitComponent : IComponentData
{
   public bool overloaded; // Bool for checking if an exit is overloaded
   public float amount; // How many entitys are in the exit quadrant
}
```

[A28]
Das Unit Spawner Proxy Skript
18.12.2019

```
using System.Collections.Generic;
using Unity.Entities;
using UnityEngine;

/// <summary>
/// Authoring
/// Starting Area of this project.
/// </summary>
[RequiresEntityConversion]
public class UnitSpawnerProxy : MonoBehaviour, IDeclareReferencedPrefabs, IConvertGameObjectToEntity
{
   #region Variables
   // Singleton instance variable
   public static UnitSpawnerProxy instance;
```

```
    // MonoBehaviour Variables
    [SerializeField] private GameObject Prefab; // Capsule human Prefab
    [SerializeField] private GameObject frontRight; // Border orientation GameObject
    [SerializeField] private GameObject frontLeft; // Border orientation GameObject
    [SerializeField] private GameObject backRight; // Border orientation GameObject
    [SerializeField] private GameObject backLeft; // Border orientation GameObject
    public int AmountToSpawn; // Amount can be set in the inspector, in later version the user can set the value for himself
    #endregion // Variables

    /// <summary>
    /// Assign instance variable.
    /// </summary>
    private void Awake()
    {
        instance = this;
    }

    /// <summary>
    /// Referenced prefabs have to be declared so that the conversion system knows about them ahead of time.
    /// </summary>
    /// <param name="gameObjects">All current assigned GameObjects, there aren't any at the moment</param>
    public void DeclareReferencedPrefabs(List<GameObject> gameObjects)
    {
        gameObjects.Add(Prefab);
    }

    /// <summary>
    /// Convert the editor data representation to the entity optimal runtime representation.
    /// </summary>
    /// <param name="entity">The first entity that handles the first steps of this process. -> Crowd GameObject from hierarchy</param>
    /// <param name="dstManager">The current actual World Entity Manager</param>
    /// <param name="conversionSystem">The Unity conversionSystem that handles everything for us</param>
    public void Convert(Entity entity, EntityManager dstManager, GameObjectConversionSystem conversionSystem)
    {
        // Create Components
        var spawnerData = new UnitSpawnerComponent
        {
            // The referenced prefab will be converted due to DeclareReferencedPrefabs
            // Mapping the game object to an entity reference
            Prefab = conversionSystem.GetPrimaryEntity(Prefab),
            AmountToSpawn = AmountToSpawn
        };

        var borderData = new BorderComponent
        {
            // Get current Border GameObject positions to save frontRight, frontLeft, backRight, backLeft position
            frontRight = new Vector3(frontRight.transform.position.x, frontRight.transform.position.y, frontRight.transform.position.z),
            frontLeft = new Vector3(frontLeft.transform.position.x, frontLeft.transform.position.y, frontLeft.transform.position.z),
            backRight = new Vector3(backRight.transform.position.x, backRight.transform.position.y, backRight.transform.position.z),
            backLeft = new Vector3(backLeft.transform.position.x, backLeft.transform.position.y, backLeft.transform.position.z)
        };

        var inputData = new InputComponent
        {
        };
```

```
        // Add created Components via EntityManager to entity (crowd GameObject)
        dstManager.AddComponentData(entity, spawnerData); // SYNC POINT // Just for the Entity Manager
        dstManager.AddComponentData(entity, borderData); // SYNC POINT // Just for the Entity Manager
        dstManager.AddComponentData(entity, inputData); // SYNC POINT // Just for the Entity Manager
    }
}
```

[A29]
Das Unit Spawner System
18.12.2019

```
<...>
using System.Collections.Generic;
using Unity.Entities;
using UnityEngine;

/// <summary>
/// Authoring
/// Starting Area of this project.
/// </summary>
[RequiresEntityConversion]
public class UnitSpawnerProxy : MonoBehaviour, IDeclareReferencedPrefabs, IConvertGameObjectToEntity
{
    #region Variables
    // Singleton instance variable
    public static UnitSpawnerProxy instance;

    // MonoBehaviour Variables
    [SerializeField] private GameObject Prefab; // Capsule human Prefab
    [SerializeField] private GameObject frontRight; // Border orientation GameObject
    [SerializeField] private GameObject frontLeft; // Border orientation GameObject
    [SerializeField] private GameObject backRight; // Border orientation GameObject
    [SerializeField] private GameObject backLeft; // Border orientation GameObject
    public int AmountToSpawn; // Amount can be set in the inspector, in later version the user can set the value for
himself
    #endregion // Variables

    /// <summary>
    /// Assign instance variable.
    /// </summary>
    private void Awake()
    {
        instance = this;
    }

    /// <summary>
    /// Referenced prefabs have to be declared so that the conversion system knows about them ahead of time.
    /// </summary>
    /// <param name="gameObjects">All current assigned GameObjects, there aren't any at the moment</param>
    public void DeclareReferencedPrefabs(List<GameObject> gameObjects)
    {
        gameObjects.Add(Prefab);
    }

    /// <summary>
    /// Convert the editor data representation to the entity optimal runtime representation.
    /// </summary>
    /// <param name="entity">The first entity that handles the first steps of this process. -> Crowd GameObject from
hierarchy</param>
    /// <param name="dstManager">The current actual World Entity Manager</param>
    /// <param name="conversionSystem">The Unity conversionSystem that handles everything for us</param>
    public void Convert(Entity entity, EntityManager dstManager, GameObjectConversionSystem conversionSystem)
```

```
   {
      // Create Components
      var spawnerData = new UnitSpawnerComponent
      {
         // The referenced prefab will be converted due to DeclareReferencedPrefabs
         // Mapping the game object to an entity reference
         Prefab = conversionSystem.GetPrimaryEntity(Prefab),
         AmountToSpawn = AmountToSpawn
      };

      var borderData = new BorderComponent
      {
         // Get current Border GameObject positions to save frontRight, frontLeft, backRight, backLeft position
         frontRight = new Vector3(frontRight.transform.position.x, frontRight.transform.position.y,
frontRight.transform.position.z),
         frontLeft = new Vector3(frontLeft.transform.position.x, frontLeft.transform.position.y,
frontLeft.transform.position.z),
         backRight = new Vector3(backRight.transform.position.x, backRight.transform.position.y,
backRight.transform.position.z),
         backLeft = new Vector3(backLeft.transform.position.x, backLeft.transform.position.y,
backLeft.transform.position.z)
      };

      var inputData = new InputComponent
      {
      };

      // Add created Components via EntityManager to entity (crowd GameObject)
      dstManager.AddComponentData(entity, spawnerData); // SYNC POINT // Just for the Entity Manager
      dstManager.AddComponentData(entity, borderData); // SYNC POINT // Just for the Entity Manager
      dstManager.AddComponentData(entity, inputData); // SYNC POINT // Just for the Entity Manager
   }
}

<...>
 /// <summary>
   /// Runs on main thread, 1 times per frame. Stops when entity (dstManger) is destroyed.
   /// </summary>
   /// <param name="inputDeps"></param>
   /// <returns>jobHandle</returns>
   protected override JobHandle OnUpdate(JobHandle inputDeps)
   {
      // Random Generator with a static ProcessorCount length
      // Used to generate individual random values inside a job
      RandomGenerator = new NativeArray<Random>(System.Environment.ProcessorCount, Allocator.TempJob);

      // Filled with random values
      // The access is via Thread Index later
      for (int i = 0; i < RandomGenerator.Length; i++)
      {
         RandomGenerator[i] = new Random((uint)Rnd.NextInt());
      }

      // Create Query for array creation (next steps)
      // Get crowd GameObject as an eneity.
      EntityQuery crowdEntity = GetEntityQuery(ComponentType.ReadOnly<UnitSpawnerComponent>());
      EntityQuery agentEntity = GetEntityQuery(ComponentType.ReadOnly<BorderComponent>());

      // Get the BorderComponent from the crowd entity
      NativeArray<BorderComponent> agentEntityBorderComponentArray =
agentEntity.ToComponentDataArray<BorderComponent>(Allocator.TempJob);

      // Get the Spawner Data from the crowd entity
```

```
        NativeArray<UnitSpawnerComponent> crowdEntityUnitSpawnerComponentArray =
crowdEntity.ToComponentDataArray<UnitSpawnerComponent>(Allocator.TempJob);

        // Create the randomPositions Native Array
        NativeArray<float3> randomPositions = new NativeArray<float3>(
            crowdEntityUnitSpawnerComponentArray[0].AmountToSpawn,
            Allocator.TempJob);

        // Value to create another seed value to create a random object to create a random value inside a job
        uint BaseSeed = (uint)UnityEngine.Random.Range(1, 100);

        // Fill the randomPositions array with random MonoBehavior float3 values
        for (int i = 0; i < randomPositions.Length; i++)
        {
            randomPositions[i] = new float3(
                UnityEngine.Random.Range(agentEntityBorderComponentArray[0].backLeft.x,
agentEntityBorderComponentArray[0].backRight.x),
                .5f,
                UnityEngine.Random.Range(agentEntityBorderComponentArray[0].frontLeft.z,
agentEntityBorderComponentArray[0].backLeft.z));
        }

        // Create SpawnJob
        SpawnJob spawnJob = new SpawnJob
        {
            CommandBuffer = m_EntityCommandBufferSystem.CreateCommandBuffer().ToConcurrent(), // Create the
commandBuffer
            randomPositions = randomPositions,
            BaseSeed = BaseSeed,
            RandomGenerator = RandomGenerator
        };

        // Schedule spawnJob with starting deps, save process inside jobHandle
        JobHandle jobHandle = spawnJob.Schedule(this, inputDeps);

        // Dispose Arrays when job finished
        agentEntityBorderComponentArray.Dispose();
        crowdEntityUnitSpawnerComponentArray.Dispose();

        // Execute the commandBuffer commands when spawnJob is finished
        m_EntityCommandBufferSystem.AddJobHandleForProducer(jobHandle);

        return jobHandle;
    }
```

[A30]
Das Input System
20.12.2019

```
<...>
 /// <summary>
    /// Job that handles different User Input.
    /// </summary>
    [BurstCompile]
    struct PlayerInputJob : IJobForEach<InputComponent>
    {
        // Data from main thread
        // Mono Behavior Input
        public bool keyOnePressedUp;
        public bool keyOnePressedDown;
        public bool keyTwoPressedUp;
        public bool keyThreePressedUp; // Bool for lifting finger from key 3 when rotating barriers outside
```

```
        public bool keyFourPressedUp; // Bool for lifting finger from key 4 when rotating barriers inside
        public bool keyFivePressedUp;
        public bool keySixPressedUp;
        public bool keySevenPressedUp;

        public bool entityInputisFocused;

        /// <summary>
        /// Assign Mono Behavior main thread Inputs to each entity with InputComponent.
        /// </summary>
        /// <param name="_inputComponent">Current Entity InputComponent</param>
        public void Execute([WriteOnly] ref InputComponent _inputComponent)
        {
            // Assign Inputs
            _inputComponent.keyThreePressedUp = keyThreePressedUp;
            _inputComponent.keyFourPressedUp = keyFourPressedUp;
            _inputComponent.keyFivePressedUp = keyFivePressedUp;
            _inputComponent.keySixPressedUp = keySixPressedUp;

            if (!entityInputisFocused)
            {
                // Seperated keys: Only enable them when the Entity Input UI Field is not focused. Otherwise the bool will
be set tu true, when you leave the UI Input field the bool will be set to true
                // and the specific action starts (e.g removing all exits).
                _inputComponent.keyOnePressedDown = keyOnePressedDown;
                _inputComponent.keyOnePressedUp = keyOnePressedUp;

                _inputComponent.keyTwoPressedUp = keyTwoPressedUp;

                _inputComponent.keySevenPressedUp = keySevenPressedUp;
            }
        }
    }

    /// <summary>
    /// Job that handles the exit entity creation.
    /// No Burst here because of the CommandBuffer and the World.Active access.
    /// </summary>
    struct CreateExitEntities : IJobForEachWithEntity<DummyComponent>
    {
        // Data from main thread
        public EntityCommandBuffer.Concurrent CommandBuffer;
        public float3 positionForExitEntity;

        /// <summary>
        /// Creating actual exit entities.
        /// </summary>
        /// <param name="entity">Current Entity</param>
        /// <param name="index">Current Entity index</param>
        /// <param name="_dummyComponent">Current _DummyComponent</param>
        public void Execute(Entity entity, int index, [ReadOnly] ref DummyComponent _dummyComponent)
        {
            // Create exit entity and add Components
            Entity exitEntity = CommandBuffer.CreateEntity(index);
            CommandBuffer.AddComponent(index, exitEntity, new Translation { Value = positionForExitEntity });
            CommandBuffer.AddComponent(index, exitEntity, new ExitComponent {});
            CommandBuffer.AddComponent(index, exitEntity, new QuadrantEntity { typeEnum =
QuadrantEntity.TypeEnum.Exit });

            // Disable Remove Exits System, otherwise the exit entity will instantly removed
            World.Active.GetExistingSystem<RemoveExitsSystem>().Enabled = false;
        }
    }
```

```
    // Variables for not creating new ones each time OnUpdate restarts
    #region // Variables
    float3 exitPosition;
    #endregion // Variables

    /// <summary>
    /// Main Thread section, where Jobs are called and connected.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        // Creating PlayerInputJob
        PlayerInputJob inputJob = new PlayerInputJob
        {
            keyOnePressedDown = UnityEngine.Input.GetKeyDown(UnityEngine.KeyCode.Alpha1), // add tag to crowd
entity
            keyOnePressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha1), // bool for allow system to
spawn entitys
            keyTwoPressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha2),
            keyThreePressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha3),
            keyFourPressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha4),
            keyFivePressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha5),
            keySixPressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha6),
            keySevenPressedUp = UnityEngine.Input.GetKeyUp(UnityEngine.KeyCode.Alpha7),
            entityInputisFocused = InputWindow.instance.inputField.isFocused
        };

        // Scheduling PlayerInputJob with starting deps
        JobHandle jobHandle = inputJob.Schedule(this, inputDeps);

        // Just for Exits placement
        if (Actions.instance.createExits)
        {
            // Create Exits mode selected
            // Now every mouseClick (0) places an exit spot.
            if (UnityEngine.Input.GetMouseButtonDown(0))
            {
                // Save mouse position and check if hitted collider is a barrier.
                // If it is a barrier, save the exit position with the hitted object and create the CreateExitEntitiesJob
                // Disable the barrier GameObject visible
                exitPosition = UnityEngine.Input.mousePosition;
                UnityEngine.Ray ray = UnityEngine.Camera.main.ScreenPointToRay(exitPosition);
                if (UnityEngine.Physics.Raycast(ray, out UnityEngine.RaycastHit hit))
                {
                    if (hit.collider != null && hit.collider.gameObject.name != "ColliderGround" &&
hit.collider.gameObject.tag != "Truss")
                    {
                        exitPosition = new float3(hit.collider.gameObject.transform.position.x, 0.5f,
hit.collider.gameObject.transform.position.z);

                        // Create CreateExitEntities
                        CreateExitEntities createExitEntitiesJob = new CreateExitEntities
                        {
                            CommandBuffer = m_EntityCommandBufferSystem.CreateCommandBuffer().ToConcurrent(), //
Create the commandBuffer
                            positionForExitEntity = exitPosition
                        };

                        // Disable barrier GameObject
                        //hit.collider.gameObject.transform.parent.gameObject.SetActive(false);
                        // Not the best implementation but there is no better solution inside ECS
```

```
                // This code only runs shortly in the frame frame where the left mouse button is pressed down
                // Disable the MeshRenderer of the first Child of the parent of the pin GameObject to disable the
GameObjects visible
                // SetActive does not work for this situation

hit.collider.gameObject.transform.parent.gameObject.transform.GetChild(0).GetComponent<UnityEngine.MeshRend
erer>().enabled = false;

                // Schedule this job when an exit spot is created with Mono behavior and the position is spotted here
                jobHandle = createExitEntitiesJob.Schedule(this, jobHandle);
            }
        }
    }
}

// Execute the commandBuffer commands when spawnJob is finished
m_EntityCommandBufferSystem.AddJobHandleForProducer(jobHandle);
return jobHandle;
}
```

[A31]
Das Moving System
20.12.2019

```
/// <summary>
/// System that reacts on changes of agentComponents of the agents. Moves the agent to the actual target position.
/// </summary>
public class MovingSystem : JobComponentSystem
{
    /// <summary>
    /// Handles the Agent movement.
    /// Runs on every entity with Translation, AgentComponent, MoveSpeedComponent Component.
    /// </summary>
    [BurstCompile]
    public struct MovementBurstJob : IJobForEachWithEntity<Translation, AgentComponent, MoveSpeedComponent>
    {
        // Data from main thread
        public float deltaTime;

        /// <summary>
        /// Handles moving behavior when Agent got the moving Tag in AgentStatus.
        /// Also check the actual y value when Agent is in Idle mode.
        /// </summary>
        /// <param name="entity">Current Entity</param>
        /// <param name="index">Current Entity index</param>
        /// <param name="_translation">Current Entity Translation Component</param>
        /// <param name="_agentComponent">Current Entity AgentComponent Component</param>
        /// <param name="_moveSpeedComponent">Current Entity MoveSpeedComponent Component</param>
        public void Execute(Entity entity, int index, ref Translation _translation, ref AgentComponent
_agentComponent, [ReadOnly] ref MoveSpeedComponent _moveSpeedComponent)
        {
            // If the agent has a target and the Moving AgentStatus
            if (_agentComponent.hasTarget && _agentComponent.agentStatus == AgentStatus.Moving)
            {
                // Calculations for checking conditions and calculating the new translation.value
                float3 direction = math.normalize(_agentComponent.target - _translation.Value);
                float distance = math.distance(_agentComponent.target, _translation.Value);

                if (distance > .5f) // agent far away
                {
                    _translation.Value += direction * _moveSpeedComponent.moveSpeed * deltaTime;
```

```
            }
            else if (distance < .5f) // close
            {
                _agentComponent.hasTarget = false;
            }
        }

        if (_agentComponent.agentStatus == AgentStatus.Idle && _translation.Value.y > .5f)
        {
            // Agent do not have a target -> Idle
            // y value is greater than .5f -> move agent down to the ground
            _translation.Value.y -= 1f * deltaTime;
        }
    }
}

/// <summary>
/// Main Thread section, where Jobs are called and connected.
/// </summary>
/// <param name="inputDeps">starting deps</param>
/// <returns>jobHandle</returns>
protected override JobHandle OnUpdate(JobHandle inputDeps)
{
    // Create MovementBurstJob
    MovementBurstJob movementBurstJob = new MovementBurstJob
    {
        deltaTime = UnityEngine.Time.deltaTime
    };

    // Schedule MovementBurstJob with starting deps
    JobHandle jobHandle = movementBurstJob.Schedule(this, inputDeps);
    return jobHandle;
}
```

[A32]
Das Jumping System
20.12.2019

```
/// <summary>
/// System that reacts on changes of agentComponents of the agents. Creates an Jumping-"dancing" Animation to simulate human behavior on a festival.
/// </summary>
public class JumpingSystem : JobComponentSystem
{
    /// <summary>
    /// Handles Jumping Behavior for each agent.
    /// Runs on every entity with Translation, MoveSpeedComponent and Agent Component Component.
    /// </summary>
    [BurstCompile]
    public struct JumpingJob : IJobForEachWithEntity<Translation, MoveSpeedComponent, AgentComponent>
    {
        // Data from main thread
        public float deltaTime;

        /// <summary>
        /// Calculate a jumping animation on current translation.value.
        /// Sum y up to .9f, if the y value is at this point, move the agent back to the ground.
        /// </summary>
        /// <param name="entity">Current Entity</param>
        /// <param name="index">Current Entity index</param>
        /// <param name="_translation">Current Translation Component</param>
```

```
    /// <param name="_moveSpeedComponent">Current MoveSpeedComponent Component</param>
    /// <param name="_agentComponent">Current AgentComponent Component</param>
    public void Execute(Entity entity, int index, ref Translation _translation, ref MoveSpeedComponent
_moveSpeedComponent, [ReadOnly] ref AgentComponent _agentComponent)
    {
        // If the agent just want to dance, handle this here
        if (!_agentComponent.hasTarget && _agentComponent.agentStatus == AgentStatus.Dancing)
        {
            // Per iteration, add a value to the y value
            _translation.Value.y += _moveSpeedComponent.jumpSpeed * deltaTime;

            // If y is at .9f, set the jumpSpeed to a negative value, to move the agent back to the ground
            if (_translation.Value.y > .9f)
            {
                _moveSpeedComponent.jumpSpeed = -math.abs(_moveSpeedComponent.jumpSpeed);
            }

            // If y is at .5f, set the jumpSpeed back to positive, to move the agent back up
            if (_translation.Value.y < .5f)
            {
                _moveSpeedComponent.jumpSpeed = +math.abs(_moveSpeedComponent.jumpSpeed);
            }
        }

        // If the Agent has a target and is in panic Mode, jump while Running
        else if (_agentComponent.hasTarget && _agentComponent.agentStatus == AgentStatus.Running)
        {
            // Per iteration, add a value to the y value
            // Use a different speed variable
            _translation.Value.y += _moveSpeedComponent.panicJumpSpeed * deltaTime;

            // If y is at .9f, set the jumpSpeed to a negative value, to move the agent back to the ground
            if (_translation.Value.y > .9f)
            {
                _moveSpeedComponent.panicJumpSpeed = -math.abs(_moveSpeedComponent.panicJumpSpeed);
            }

            // If y is at .5f, set the jumpSpeed back to positive, to move the agent back up
            if (_translation.Value.y < .5f)
            {
                _moveSpeedComponent.panicJumpSpeed = +math.abs(_moveSpeedComponent.panicJumpSpeed);
            }
        }

        // Get Agent back to the ground when it has Idle or Moving AgentStatus
        if ((_agentComponent.agentStatus == AgentStatus.Idle || _agentComponent.agentStatus ==
AgentStatus.Moving)
            && _translation.Value.y > .5f)
        {
            // Go back to normal position (Y value .5f)
            // Prevent from beeing in y > .5f while moving/running
            _translation.Value.y -= 5f * deltaTime;
        }
        #region Old Version (Physics version. Much influence on performance. Physics Scripts on Human Prefab needs
to be enabled. Unity Physics needs to be installed into this project from the Package Manager)
        //if (!agentComponent.hasTarget && agentComponent.agentStatus == AgentStatus.Dancing &&
agentComponent.jumped == false) // Agent dont have a target, actual AgentStatus is Dancing
        //{
        //    physicsVelocity.Linear.y = 3f; // Add a force to the Agent's physics component
        //    agentComponent.jumped = true;
        //}

        //if (translation.Value.y == 0.5f)
```

```
            //{
            //    agentComponent.jumped = false;
            //}
            #endregion // Old Version (Physics version. Much influence on performance. Physics Scripts on Human Prefab needs to be enabled.)
        }
    }

    /// <summary>
    /// Main Thread section, where Jobs are called and connected.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        // Create JumpingJob
        JumpingJob jumpingJob = new JumpingJob
        {
            deltaTime = UnityEngine.Time.deltaTime
        };

        // Scheduling JumpingJob
        JobHandle jobHandle = jumpingJob.Schedule(this, inputDeps);

        return jobHandle;
    }
```

[A33]
Das Running System
20.12.2019

```
/// <summary>
/// System that reacts on Running AgentStatus.
/// </summary>
public class RunningSystem : JobComponentSystem
{
    /// <summary>
    /// Calculates the Running Behavior on Agents with Translation, AgentComponent and MoveSpeedComponent.
    /// </summary>
    [BurstCompile]
    public struct RunningJob : IJobForEachWithEntity<Translation, AgentComponent, MoveSpeedComponent>
    {
        // Data from main thread
        [ReadOnly] public float deltaTime;

        /// <summary>
        /// React on different situations and move the current Agent.
        /// </summary>
        /// <param name="entity">Current entity</param>
        /// <param name="index">Current entity index</param>
        /// <param name="_translation">Current Entity Translation Component</param>
        /// <param name="_agentComponent">Current Entity Agent Component</param>
        /// <param name="_moveSpeedComponent">Current Entity Move Speed Component</param>
        public void Execute(Entity entity, int index, ref Translation _translation, ref AgentComponent _agentComponent, [ReadOnly] ref MoveSpeedComponent _moveSpeedComponent)
        {
            // If an agent got the running Tag and has a Target
            if (_agentComponent.agentStatus == AgentStatus.Running && _agentComponent.hasTarget)
            {
                // Calculate the distance between this agent and the target
                float distance = math.distance(_translation.Value, _agentComponent.target);
```

```
            // [ELSE] If this distance is greater than .1f (far away), calculate the direction with the target and the current agent position
            // and add this direction to the current position value
            // [IF] If the distance is less than .1f, change mode because the agent reached its target
            // Take a look on the target, if the target is an exit spot, change to moving and move away from the exit spot (Moving System does its job)
            // The other case is just a normal randomly generated position that has been reached, in both cases, change bools to trigger other Systems/Jobs
            if (distance < .1f)
            {
               if (_agentComponent.foundFinalExitPoint)
               {
                  // Agent is near to the target
                  _agentComponent.agentStatus = AgentStatus.Moving;
                  _agentComponent.hasTarget = false;
                  _agentComponent.exitPointReached = true; // The CalculateNewRandomPositionSystem is able now to seperate the agents and allow the correct new random generated positions
                  _agentComponent.marked = false;
               }
               else
               {
                  _agentComponent.hasTarget = false;
                  _agentComponent.marked = false;
               }
            }
            else
            {
               // Agent has panic, need to run to the next closest escape spot
               float3 direction = math.normalize(_agentComponent.target - _translation.Value);
               _translation.Value += direction * _moveSpeedComponent.runningSpeed * deltaTime;
            }
         }
      }
   }

   /// <summary>
   /// Runs on main thread, 1 times per frame
   /// </summary>
   /// <param name="inputDeps">starting deps</param>
   /// <returns>jobHandle</returns>
   protected override JobHandle OnUpdate(JobHandle inputDeps)
   {
      // Creating Running Job
      RunningJob runningJob = new RunningJob
      {
         deltaTime = UnityEngine.Time.deltaTime,
      };

      // Scheduling runningJob
      JobHandle jobHandle = runningJob.Schedule(this, inputDeps);

      return jobHandle;
   }
```

[A34]
Das CalculateNewRandomPosition System
20.12.2019

```
/// <summary>
  /// Method that checks if a position is inside the festival area.
  /// </summary>
  /// <param name="_borderComponent">Used BorderComponent</param>
  /// <param name="_testPosition">Checked test position</param>
  /// <returns>The information if the given _testPosition is inside the festival area</returns>
  [BurstCompile]
  public static bool IsInsideFestivalArea(ref BorderComponent _borderComponent, float3 _testPosition)
  {
     // All different festival area corners, saved in different variables, used for later calculations
     float3 a = _borderComponent.frontLeft;
     float3 b = _borderComponent.backLeft;
     float3 c = _borderComponent.backLeft;

     float3 d = _borderComponent.frontRight;
     float3 e = _borderComponent.backRight;
     float3 f = _borderComponent.backRight;

     // Left Triangle
     float as_x_i = _testPosition.x - a.x;
     float as_z_i = _testPosition.z - a.z;

     // Right Triangle
     float as_x_ii = _testPosition.x - d.x;
     float as_z_ii = _testPosition.z - d.z;

     bool s_ab = (b.x - a.x) * as_z_i - (b.z - a.z) * as_x_i > 0; // Front.Left to Back.Left
     bool s_de = (e.x - d.x) * as_z_ii - (e.z - d.z) * as_x_ii > 0; // Front.Right to Back.Right

     // Mathematical caluclation if given position is not inside the 2 triangles of the whole square [1]
     if ((f.x - d.x) * as_z_ii - (d.z - d.z) * as_x_ii > 0 == s_de && (c.x - a.x) * as_z_i - (a.z - a.z) * as_x_i > 0 == s_ab)
     {
        // Additional check if z value is greater than front.z and x value is between back left.x and back right.x
        if (_testPosition.z >= _borderComponent.frontLeft.z
        && _testPosition.x <= _borderComponent.backLeft.x
        && _testPosition.x >= _borderComponent.backRight.x)
        {
           // _testPosition is inside festival area
           return true;
        }
     }
     // Mathematical caluclation if given position is not inside the 2 triangles of the whole square [2]
     else if ((f.x - e.x) * (_testPosition.z - e.z) - (d.z - e.z) * (_testPosition.x - e.x) > 0 != s_de
        && (c.x - b.x) * (_testPosition.z - b.z) - (a.z - b.z) * (_testPosition.x - b.x) > 0 != s_ab)
     {
        // Additional check if z value is greater than front.z and x value is between back left.x and back right.x
        if (_testPosition.z >= _borderComponent.frontLeft.z
        && _testPosition.x <= _borderComponent.backLeft.x
        && _testPosition.x >= _borderComponent.backRight.x)
        {
           // _testPosition is inside festival area
           return true;
        }
     }
     // no match -> _testPosition is not inside the festival area
     return false;
  }

  /// <summary>
  /// System that runs on every Agent Entity, to check if this agent is in Moving AgentStatus with no current target.
  /// If this is the case, calculate a new one.
  /// </summary>
  [BurstCompile]
```

```
   struct CalculateNewRandomPositionBurstJob : IJobForEachWithEntity<Translation, AgentComponent,
BorderComponent>
   {
      // Data from main thread
      [NativeDisableParallelForRestriction] // Enables writing to any index of RandomGenerator
      [DeallocateOnJobCompletion]
      public NativeArray<Random> RandomGenerator; // For generating random values inside this job

      [Unity.Collections.LowLevel.Unsafe.NativeSetThreadIndex]
      private int threadIndex; // For generating individual random values inside this job

      /// <summary>
      /// Case 1: Agent reached goal at exit and is in Moving AgentStatus now. -> Calculate a new random position
outside the festival area.
      /// Case 2: Agent reached it's random goal inside the festival area and need a new one now. -> Calculate a new
random position inside the festival area. Check if this random position is valid.
      /// </summary>
      /// <param name="entity">Current Entity</param>
      /// <param name="index">Current Entity index</param>
      /// <param name="_translation">Current Entity Translation Component</param>
      /// <param name="_agentComponent">Current Entity AgentComponent</param>
      /// <param name="_borderComponent">Current Entity Border Component</param>
      public void Execute(Entity entity, int index, ref Translation _translation, ref AgentComponent
_agentComponent, ref BorderComponent _borderComponent)
      {
         // If an Agent is in Moving AgentStauts and don't have a target yet
         if (!_agentComponent.hasTarget && _agentComponent.agentStatus == AgentStatus.Moving)
         {
            // initialization
            var rnd = RandomGenerator[threadIndex - 1];
            float3 calculatedRandomPosition = float3.zero;

            // Case 1: Agent reached exit, check which side and calculate a random position outside the festival area
            if (_agentComponent.exitPointReached)
            {
               if (_translation.Value.x >= 180f)
               {
                  // Agent is on a left exit spot
                  // Calculate a random Position that points to the right side
                  calculatedRandomPosition = new float3(
                     rnd.NextFloat(_translation.Value.x + rnd.NextFloat(3f, 6f), _translation.Value.x +
rnd.NextFloat(3f, 6f)),
                     .5f,
                     rnd.NextFloat(_translation.Value.z - rnd.NextFloat(3f, 4f), _translation.Value.z + rnd.NextFloat(3f,
4f)));
               }
               else
               {
                  // Agent is on a right exit spot
                  // Calculate a random Position that points to the left side
                  calculatedRandomPosition = new float3(
                     rnd.NextFloat(_translation.Value.x - rnd.NextFloat(3f, 6f), _translation.Value.x - rnd.NextFloat(3f,
6f)),
                     .5f,
                     rnd.NextFloat(_translation.Value.z - rnd.NextFloat(3f, 4f), _translation.Value.z + rnd.NextFloat(3f,
4f)));
               }
            }
            else
            {
               // Case 2: Agent reached it's old random position inside the festival area, calculate a new one
               calculatedRandomPosition = new float3(
               rnd.NextFloat((_translation.Value.x - 3f), (_translation.Value.x + 3f)),
```

```
            .5f,
            rnd.NextFloat((_translation.Value.z - 3f), (_translation.Value.z + 3f)));
        }

        RandomGenerator[threadIndex - 1] = rnd; // This is necessary to update the state of the element inside
the array.

        // ##### Check if calculated random festival position is valid ##### //

        // All different festival area corners, saved in different variables, used for later calculations
        float3 a = _borderComponent.frontLeft;
        float3 b = _borderComponent.backLeft;
        float3 c = _borderComponent.backLeft;

        float3 d = _borderComponent.frontRight;
        float3 e = _borderComponent.backRight;
        float3 f = _borderComponent.backRight;

        // Left Triangle
        float as_x_i = calculatedRandomPosition.x - a.x;
        float as_z_i = calculatedRandomPosition.z - a.z;

        // Right Triangle
        float as_x_ii = calculatedRandomPosition.x - d.x;
        float as_z_ii = calculatedRandomPosition.z - d.z;

        bool s_ab = (b.x - a.x) * as_z_i - (b.z - a.z) * as_x_i > 0; // Front.Left to Back.Left
        bool s_de = (e.x - d.x) * as_z_ii - (e.z - d.z) * as_x_ii > 0; // Front.Right to Back.Right

        // Mathematical caluclation if given position is not inside the 2 triangles of the whole square [1]
        if ((f.x - d.x) * as_z_ii - (d.z - d.z) * as_x_ii > 0 == s_de && (c.x - a.x) * as_z_i - (a.z - a.z) * as_x_i > 0 ==
s_ab)
        {
            // Outside festival area
            if (_agentComponent.exitPointReached)
            {
                // No validation needed
                _agentComponent.target = calculatedRandomPosition;
                _agentComponent.hasTarget = true;
            }
            else
            {
                // Outside festival area
                // Exit point not reached
                _agentComponent.target = _translation.Value;
                _agentComponent.hasTarget = false;
            }

            // Additional check if z value is greater than front.z and x value is between back left.x and back right.x
            if (calculatedRandomPosition.z >= _borderComponent.frontLeft.z
            && calculatedRandomPosition.x <= _borderComponent.backLeft.x
            && calculatedRandomPosition.x >= _borderComponent.backRight.x
            && !_agentComponent.exitPointReached)
            {
                _agentComponent.target = calculatedRandomPosition;
                _agentComponent.hasTarget = true;
            }
        }

        // Mathematical caluclation if given position is not inside the 2 triangles of the whole square [2]
        else if ((f.x - e.x) * (calculatedRandomPosition.z - e.z) - (d.z - e.z) * (calculatedRandomPosition.x - e.x) > 0
!= s_de
```

```
                && (c.x - b.x) * (calculatedRandomPosition.z - b.z) - (a.z - b.z) * (calculatedRandomPosition.x - b.x) > 0
!= s_ab)
            {
                if (_agentComponent.exitPointReached)
                {
                    // No validation needed
                    _agentComponent.target = calculatedRandomPosition;
                    _agentComponent.hasTarget = true;
                }
                else
                {
                    // Outside festival area
                    // Exit point not reached
                    _agentComponent.target = _translation.Value;
                    _agentComponent.hasTarget = false;
                }

                // Additional check if z value is greater than front.z and x value is between back left.x and back right.x
                if (calculatedRandomPosition.z >= _borderComponent.frontLeft.z
                && calculatedRandomPosition.x <= _borderComponent.backLeft.x
                && calculatedRandomPosition.x >= _borderComponent.backRight.x
                && !_agentComponent.exitPointReached)
                {
                    _agentComponent.target = calculatedRandomPosition;
                    _agentComponent.hasTarget = true;
                }
            }
            else
            {
                // Inside Festival
                if (_agentComponent.exitPointReached)
                {
                    // No validation needed
                    _agentComponent.target = calculatedRandomPosition;
                    _agentComponent.hasTarget = true;
                }
                else
                {
                    // Outside festival area
                    // Exit point not reached
                    _agentComponent.target = _translation.Value;
                    _agentComponent.hasTarget = false;
                }
            }
        }
    }
  }

<...>

 /// <summary>
   /// Main Thread section, where Jobs are called and connected.
   /// </summary>
   /// <param name="inputDeps">starting deps</param>
   /// <returns>jobHandle</returns>
   protected override JobHandle OnUpdate(JobHandle inputDeps)
   {
      // Initialize Native Array with the processorCount length and the TempJob Allocator tag
      RandomGenerator = new NativeArray<Random>(System.Environment.ProcessorCount, Allocator.TempJob);

      // Fill the RandomGenerator with random Random objects
      for (int i = 0; i < RandomGenerator.Length; i++)
      {
```

```
            RandomGenerator[i] = new Random((uint)Rnd.NextInt());
        }

        // Create CalculateNewRandomPositionBurstJob
        CalculateNewRandomPositionBurstJob calculateNewRandomPositionBurstjob = new
CalculateNewRandomPositionBurstJob
        {
            RandomGenerator = RandomGenerator
        };

        // Schedule CalculateNewRandomPositionBurstJob with starting deps
        JobHandle jobHandle = calculateNewRandomPositionBurstjob.Schedule(this, inputDeps);

        // Execute the commandBuffer commands when spawnJob is finished
        m_EntityCommandBufferSystem.AddJobHandleForProducer(jobHandle);

        return jobHandle;
    }
```

[A35]
Das Panic System
20.12.2019

```
/// <summary>
    /// Calculates and saves the closest exit target.
    /// </summary>
    /// <param name="_agentPosition">Current Agent position</param>
    /// <param name="_agentComponent">Current Agent Component</param>
    /// <param name="_exitsTranslations">Current Exits</param>
    [BurstCompile]
    public static void CheckForClosestExit(float3 _agentPosition, [WriteOnly] ref AgentComponent _agentComponent,
[ReadOnly] NativeArray<Translation> _exitsTranslations)
    {
        // Set first value as closest target
        float3 closestExit = _exitsTranslations[0].Value;

        // Loop through each exit and calculate if an exit is closer than the actual closest exit
        for (int i = 0; i < _exitsTranslations.Length; i++)
        {
            if (math.distance(_agentPosition, _exitsTranslations[i].Value) < math.distance(_agentPosition, closestExit))
            {
                closestExit = _exitsTranslations[i].Value;
            }
        }

        //If Agent can see an exit, set this exit as target
        _agentComponent.target = closestExit;
        _agentComponent.hasTarget = true;
        _agentComponent.foundFinalExitPoint = true;
        _agentComponent.foundTemporaryNewRandomPosition = false;
    }

    /// <summary>
    /// Every Agent in given radius gets the Panic AgentStatus.
    /// </summary>
    [BurstCompile]
    public struct EnablePanicModeJob : IJobForEachWithEntity<AgentComponent, Translation>
    {
        // Data from main thread
        [ReadOnly] public float panicRadius;
        [ReadOnly] public float3 actionPosition;
```

```
    public void Execute(Entity entity, int index, [WriteOnly] ref AgentComponent _agentComponent, [ReadOnly] ref
Translation _translation)
    {
        if (math.distance(_translation.Value, actionPosition) <= panicRadius)
        {
            // Agent close to the action position
            // Enable Pre Panic Mode
            _agentComponent.agentStatus = AgentStatus.Running;
        }
    }
}

/// <summary>
/// The main panic Job to calculate a panic reaction for each agent individually.
/// </summary>
[BurstCompile]
public struct PanicJob : IJobForEachWithEntity<Translation, AgentComponent, BorderComponent>
{
    // Data from main thread
    [ReadOnly] public NativeMultiHashMap<int, QuadrantData> quadrantMultiHashMap; // Quadrant HashMap
    [ReadOnly] public NativeArray<Translation> exitsTranslations; // Current Exit Translations
    [ReadOnly] public NativeArray<ExitComponent> exitsExitComponents; // Current Exit ExitComponents

    [NativeDisableParallelForRestriction] public NativeArray<Random> RandomGenerator; // Filled
RandomGenerator
    [Unity.Collections.LowLevel.Unsafe.NativeSetThreadIndex] private int threadIndex; // Current Thread Index

    /// <summary>
    /// This Job is devided into three parts.
    /// Part 1: If there are any Agents with no target and a Running Tag (Agents near an explosion), calculate a
random position and check if this random position is inside the festival area.
    /// If it's not inside the festival area, generate a new random position. If it's inside the festival area, set this
random position to the new target of this agent. The Running System will react on this information.
    /// Part 2: Agent has a target and is running to this target. If the Agent can see an exit, check if this exit is not
overloaded.
    /// If it's overloaded, go to Part 1. If it's not overloaded, set this exit as target and set the
foundFinalExitPosition bool to true.
    /// Part 3: Agents on the way to an exit are using this part. If they notice that the current Exit as their target is
overloaded, calculate a random value.
    /// Compare this value with the current fleeProbability of themselfes. Basically the Agent decided randomly if
it keeps the exit or not, to calculate a new random value like in Part 1.
    /// </summary>
    /// <param name="entity">Current entity</param>
    /// <param name="index">Current entity Index</param>
    /// <param name="_translation">Current Entity Translation Component</param>
    /// <param name="_agentComponent">Current Entity AgentComponent</param>
    /// <param name="_borderComponent">Current Entity BorderComponent</param>
    public void Execute(Entity entity, int index, [ReadOnly] ref Translation _translation, ref AgentComponent
_agentComponent, [ReadOnly] ref BorderComponent _borderComponent)
    {
        // Random initialization
        var randomGenerator = RandomGenerator[threadIndex - 1];

        // This is necessary to update the state of the element inside the array.
        RandomGenerator[threadIndex - 1] = randomGenerator;
        var rnd = RandomGenerator[threadIndex - 1];

        // Generate a new random position on the map
        if (_agentComponent.agentStatus == AgentStatus.Running && !_agentComponent.hasTarget)
        {
            // Set target for enabling Running Job in Running System
            float3 randomGeneratedPanicPosition = float3.zero;
```

```
            float3 tempRandomPosition = float3.zero;
            while (randomGeneratedPanicPosition.Equals(float3.zero))
            {
                // While the result float3 randomGeneratedPanicPosition = float3.zero
                // Generate a random position on the map
                // And calculate if this position is inside the festival Area
                // Set this position to the new target if it's inside
                tempRandomPosition = new float3(
                    rnd.NextFloat(_borderComponent.backRight.x, _borderComponent.backLeft.x),
                    .5f,
                    rnd.NextFloat(_borderComponent.frontLeft.z, _borderComponent.backLeft.z));

                if (CalculateNewRandomPositionSystem.IsInsideFestivalArea(ref _borderComponent,
tempRandomPosition))
                {
                    // If tempRandomPosition is inside the festival Area, set this random Position and trigger the Running
Job in the RunningSystem
                    randomGeneratedPanicPosition = tempRandomPosition;

                    _agentComponent.target = randomGeneratedPanicPosition;
                    _agentComponent.hasTarget = true;
                    _agentComponent.foundTemporaryNewRandomPosition = true;
                }
            }
        }
        // Calculate nearest exit. Look on the exit and check if its overloaded. Only set this exit as target when its
not overloaed
        else if (_agentComponent.agentStatus == AgentStatus.Running && _agentComponent.hasTarget &&
_agentComponent.foundTemporaryNewRandomPosition && !_agentComponent.foundFinalExitPoint &&
!_agentComponent.marked)
        {
            for (int i = 0; i < exitsTranslations.Length; i++)
            {
                // Loop through each exit and calculate if this exit is close to me as an agent
                if (math.distance(_translation.Value, exitsTranslations[i].Value) < 20f)
                {
                    // If there is an exit that is close to me, check if this exit is overloaded
                    if (!exitsExitComponents[i].overloaded)
                    {
                        //If Agent can see an exit and this exit is not overloaded, set this exit as target
                        _agentComponent.target = exitsTranslations[i].Value;
                        _agentComponent.hasTarget = true;
                        _agentComponent.foundFinalExitPoint = true;
                        _agentComponent.foundTemporaryNewRandomPosition = false;
                    }
                    else
                    {
                        // If this exit is overloaded, go to Part 1
                        _agentComponent.hasTarget = false;
                    }
                }
            }
        }
        // Agents notices on the way to the exit that this exis is overloaded now, randomly decide if it shall keep this
exit or to choose another one
        if (_agentComponent.foundFinalExitPoint && _agentComponent.hasTarget)
        {
            // Agent runs to an exit
            for (int i = 0; i < exitsTranslations.Length; i++)
            {
                // Loop through all exits and check if this exit is equals to the agents target
                if (_agentComponent.target.Equals(exitsTranslations[i].Value))
                {
```

```
                // If this exit is overloaded, so the agents current target is overloaded
                if (exitsExitComponents[i].overloaded)
                {
                    // Calculate a random value and compare it with the agents current fleeProbability.
                    // Agents notices on the way to the exit that this exis is overloaded now, randomly decide if it shall keep this exit or to choose another one
                    float dice = rnd.NextFloat(1000f);

                    if (dice <= _agentComponent.fleeProbability)
                    {
                        _agentComponent.hasTarget = false;
                        _agentComponent.foundFinalExitPoint = false;
                        _agentComponent.marked = true;

                        // Take care of the fleeProbability, otherwise it will go negative.
                        // Set it to an static value of 0.11f when its below 0.0f
                        if (_agentComponent.fleeProbability - 5.55f < 0.0f)
                        {
                            _agentComponent.fleeProbability = .11f;
                        }
                        else
                        {
                            _agentComponent.fleeProbability -= 5.55f;
                        }
                    }
                }
            }
        }
    }
}

/// <summary>
/// Job that handles the reaction of an agent when beeing in near of an agent that has panic and is in running mode.
/// </summary>
[BurstCompile]
public struct ReactOnPanicInsideQuadrantJob : IJobForEachWithEntity<Translation, AgentComponent, QuadrantEntity>
{
    // Data from main thread
    // instantiating and deleting of Entitys can only gets done on the main thread, save commands in buffer for main thread
    public EntityCommandBuffer.Concurrent CommandBuffer;

    [ReadOnly] public NativeMultiHashMap<int, QuadrantData> quadrantMultiHashMap; // Quadrant System HashMap
    [ReadOnly] public NativeArray<Translation> exitsTranslations; // Current exit Translation Components
    [ReadOnly] public NativeArray<ExitComponent> exitsExitComponents; // Current exit ExitComponents
    [ReadOnly] public float3 actionPosition; // position where panic appears

    [NativeDisableParallelForRestriction]
    [DeallocateOnJobCompletion]
    public NativeArray<Random> RandomGenerator; // Filled RandomGenerator

    [Unity.Collections.LowLevel.Unsafe.NativeSetThreadIndex]
    [ReadOnly]
    private int threadIndex; // Current thread Index

    /// <summary>
    /// Calculate random quadrants where The Method CalculatePanicReaction searches for matching entities.
    /// </summary>
    /// <param name="entity">Current Entity</param>
```

```
    /// <param name="index">Current entity index</param>
    /// <param name="_translation">Current entity Translation Component</param>
    /// <param name="_agentComponent">Current entity AgentComponent</param>
    /// <param name="_quadrantEntity">Current entity QuadrantEntity</param>
    public void Execute(Entity entity, int index, [ReadOnly] ref Translation _translation, ref AgentComponent
_agentComponent, ref QuadrantEntity _quadrantEntity)
    {
        // Random initialization
        var randomGenerator = RandomGenerator[threadIndex - 1];
        RandomGenerator[threadIndex - 1] = randomGenerator; // This is necessary to update the state of the
element inside the array.
        var rnd = RandomGenerator[threadIndex - 1];

        // Generate different random quadrant índices as index for the HashMapKeys
        int randomQuadrantSearchIndex = 0;
        if (_agentComponent.agentStatus == AgentStatus.Running)
        {
            randomQuadrantSearchIndex = rnd.NextInt(6, 10);
        }
        else
        {
            randomQuadrantSearchIndex = rnd.NextInt(1, 4);
        }

        // Calculate the correct quadrant for this agent
        int hashMapKey = QuadrantSystem.GetPositionHashMapKey(_translation.Value);

        // Call CalculatePanicReaction for each direction
        CalculatePanicReaction(hashMapKey, ref _translation, ref _agentComponent, index, entity); // This quadrant
itself (mid)
        CalculatePanicReaction(hashMapKey + randomQuadrantSearchIndex, ref _translation, ref _agentComponent,
index, entity); // Right quadrant
        CalculatePanicReaction(hashMapKey - randomQuadrantSearchIndex, ref _translation, ref _agentComponent,
index, entity); // Left quadrant
        CalculatePanicReaction(hashMapKey + QuadrantSystem.quadrantYMultiplier, ref _translation, ref
_agentComponent, index, entity); // Above quadrant
        CalculatePanicReaction(hashMapKey - QuadrantSystem.quadrantYMultiplier, ref _translation, ref
_agentComponent, index, entity); // Below quadrant

        CalculatePanicReaction(hashMapKey + randomQuadrantSearchIndex + QuadrantSystem.quadrantYMultiplier,
ref _translation, ref _agentComponent, index, entity); // Corner Top Right
        CalculatePanicReaction(hashMapKey - randomQuadrantSearchIndex + QuadrantSystem.quadrantYMultiplier,
ref _translation, ref _agentComponent, index, entity); // Corner Top Left
        CalculatePanicReaction(hashMapKey + randomQuadrantSearchIndex - QuadrantSystem.quadrantYMultiplier,
ref _translation, ref _agentComponent, index, entity); // Corner Bottom Right
        CalculatePanicReaction(hashMapKey - randomQuadrantSearchIndex - QuadrantSystem.quadrantYMultiplier,
ref _translation, ref _agentComponent, index, entity); // Corner Bottom Left
    }

    /// <summary>
    /// Case 1: Agent sees another agent which has the Running Tag as AgentStatus. Current Agent copy this
behavior and also goes into Panic mode.
    /// Case 2: Agent sees another agent which runs to an exit. Current Agent copy this behavior and also runs to
this exit.
    /// </summary>
    /// <param name="_hashMapKey">Passed calculated hashMap key</param>
    /// <param name="_translation">Passed entity Translation Component</param>
    /// <param name="_agentComponent">Passed entity AgentComponent</param>
    /// <param name="entityIndex">Passed entity Index</param>
    /// <param name="entity">Passed entity</param>
    [BurstCompile]
    private void CalculatePanicReaction([ReadOnly] int _hashMapKey, [ReadOnly] ref Translation _translation, ref
AgentComponent _agentComponent, int entityIndex, Entity entity)
```

```
        {
            // Cycling through all entitys/agents inside this quadrant
            QuadrantData quadrantData;
            NativeMultiHashMapIterator<int> nativeMultiHashMapIterator;
            if (quadrantMultiHashMap.TryGetFirstValue(_hashMapKey, out quadrantData, out
nativeMultiHashMapIterator))
            {
                do
                {
                    // Check if agent sees an agent with Running AgentStatus
                    // If this is the case, copy this Status and set it to Running
                    if (math.distance(_translation.Value, quadrantData.position) < 20
                        && quadrantData.agentComponent.agentStatus == AgentStatus.Running
                        && !quadrantData.agentComponent.exitPointReached // prevent stopping at exit
                        && !_agentComponent.exitPointReached // prevent stopping at exit
                        && _agentComponent.agentStatus != AgentStatus.Running) // prevent from adding Panic Tags again
and again
                    {
                        _agentComponent.agentStatus = AgentStatus.Running;
                    }

                    // Check if agent sees an agent with a target.
                    // If this is the case, copy this target
                    if (quadrantData.agentComponent.foundFinalExitPoint
                        && !quadrantData.agentComponent.exitPointReached
                        && !_agentComponent.exitPointReached
                        && !_agentComponent.marked)
                    {
                        _agentComponent.target = quadrantData.agentComponent.target;
                        _agentComponent.foundFinalExitPoint = true;
                        _agentComponent.hasTarget = true;
                    }
                } while (quadrantMultiHashMap.TryGetNextValue(out quadrantData, ref nativeMultiHashMapIterator));
            }
        }
    }

<...>

/// <summary>
    /// Main Thread section, where Jobs are called and connected.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        JobHandle jobHandle = new JobHandle();

        // Random Generator with a static ProcessorCount length
        // Used to generate individual random values inside a job
        RandomGenerator = new NativeArray<Random>(System.Environment.ProcessorCount, Allocator.TempJob);

        // Filled with random values
        // The access is via Thread Index later
        for (int i = 0; i < RandomGenerator.Length; i++)
        {
            RandomGenerator[i] = new Random((uint)Rnd.NextInt());
        }

        // React on different states of the Radial Menu and start specific jobs
        if (Actions.instance.actionEnabled)
        {
            // Get all exit entities
```

```
        EntityQuery exitQuery = GetEntityQuery(typeof(ExitComponent), ComponentType.ReadOnly<Translation>());

        // Extract Translation and ExitComponent Components
        NativeArray<Translation> exitsTranslations =
exitQuery.ToComponentDataArray<Translation>(Allocator.TempJob);
        NativeArray<ExitComponent> exitsExitComponents =
exitQuery.ToComponentDataArray<ExitComponent>(Allocator.TempJob);

        // If smallGroundExplosion Icon was choosen
        if (Actions.instance.smallGroundExplosion)
        {
            // If User pressed left mouse button down
            if (UnityEngine.Input.GetMouseButtonDown(0))
            {
                // Handle the mouse position and pass it to the EnablePanicModeJob later
                actionPosition = UnityEngine.Input.mousePosition;
                UnityEngine.Ray ray = UnityEngine.Camera.main.ScreenPointToRay(actionPosition);
                if (UnityEngine.Physics.Raycast(ray, out UnityEngine.RaycastHit hit))
                {
                    if (hit.collider != null)
                    {
                        actionPosition = new float3(hit.point.x, .5f, hit.point.z);
                    }
                }

                // Create EnablePanicModeJob which creates a panic reaction around the action effect
                EnablePanicModeJob enablePrePanicJob = new EnablePanicModeJob
                {
                    actionPosition = actionPosition,
                    panicRadius = 5f,
                };

                // Schedule the EnablePanicModeJob Job with starting deps, save results into JobHandle
                jobHandle = enablePrePanicJob.Schedule(this, inputDeps);
            }

            // When choosen, always start a panic job to simulate panic behavior
            // Create Panic Job
            PanicJob panicJob = new PanicJob
            {
                RandomGenerator = RandomGenerator,
                exitsTranslations = exitsTranslations,
                exitsExitComponents = exitsExitComponents,
                quadrantMultiHashMap = QuadrantSystem.quadrantMultiHashMap
            };

            // Schedule Panic Job
            jobHandle = panicJob.Schedule(this, jobHandle);
        }

        // If mediumGroundExplosion Icon was choosen
        else if (Actions.instance.mediumGroundExplosion)
        {
            // If User pressed left mouse button down
            if (UnityEngine.Input.GetMouseButtonDown(0))
            {
                // Handle the mouse position and pass it to the EnablePanicModeJob later
                actionPosition = UnityEngine.Input.mousePosition;
                UnityEngine.Ray ray = UnityEngine.Camera.main.ScreenPointToRay(actionPosition);
                if (UnityEngine.Physics.Raycast(ray, out UnityEngine.RaycastHit hit))
                {
                    if (hit.collider != null)
                    {
```

```
                actionPosition = new float3(hit.point.x, .5f, hit.point.z);
            }
        }

        // Create EnablePanicModeJob which creates a panic reaction around the action effect
        EnablePanicModeJob enablePrePanicJob = new EnablePanicModeJob
        {
            actionPosition = actionPosition,
            panicRadius = 10f,
        };

        // Schedule the EnablePanicModeJob with starting deps, save results into JobHandle
        jobHandle = enablePrePanicJob.Schedule(this, inputDeps);
    }

    // When choosen, always start a panic job to simulate panic behavior
    // Create Panic Job
    PanicJob panicJob = new PanicJob
    {
        RandomGenerator = RandomGenerator,
        exitsTranslations = exitsTranslations,
        exitsExitComponents = exitsExitComponents,
        quadrantMultiHashMap = QuadrantSystem.quadrantMultiHashMap
    };

    // Schedule Panic Job
    jobHandle = panicJob.Schedule(this, jobHandle);
}

// If bigGroundExplosion Icon was choosen
else if (Actions.instance.bigGroundExplosion)
{
    // If User pressed left mouse button down
    if (UnityEngine.Input.GetMouseButtonDown(0))
    {
        // Handle the mouse position and pass it to the EnablePanicModeJob later
        actionPosition = UnityEngine.Input.mousePosition;
        UnityEngine.Ray ray = UnityEngine.Camera.main.ScreenPointToRay(actionPosition);
        if (UnityEngine.Physics.Raycast(ray, out UnityEngine.RaycastHit hit))
        {
            if (hit.collider != null)
            {
                actionPosition = new float3(hit.point.x, .5f, hit.point.z);
            }
        }

        // Create EnablePanicModeJob which creates a panic reaction around the action effect
        EnablePanicModeJob enablePrePanicJob = new EnablePanicModeJob
        {
            actionPosition = actionPosition,
            panicRadius = 15f,
        };

        // Schedule the EnablePanicModeJob Job with starting deps, save results into JobHandle
        jobHandle = enablePrePanicJob.Schedule(this, inputDeps);
    }

    // When choosen, always start a panic job to simulate panic behavior
    // Create Panic Job
    PanicJob panicJob = new PanicJob
    {
        RandomGenerator = RandomGenerator,
        exitsTranslations = exitsTranslations,
```

```
            exitsExitComponents = exitsExitComponents,
            quadrantMultiHashMap = QuadrantSystem.quadrantMultiHashMap
        };

        // Schedule Panic Job
        jobHandle = panicJob.Schedule(this, jobHandle);
    }

    // If smallGroundExplosion Icon was choosen
    else if (Actions.instance.fire)
    {
        // If User pressed left mouse button down
        if (UnityEngine.Input.GetMouseButtonDown(0))
        {
            // Handle the mouse position and pass it to the EnablePanicModeJob later
            // Check if the clicked position is a Sound System
            // Disable Information Arrows
            actionPosition = UnityEngine.Input.mousePosition;
            UnityEngine.Ray ray = UnityEngine.Camera.main.ScreenPointToRay(actionPosition);
            if (UnityEngine.Physics.Raycast(ray, out UnityEngine.RaycastHit hit))
            {
                if (hit.collider != null)
                {
                    string hittedGameObjectName = hit.collider.gameObject.name;
                    if (hittedGameObjectName == "Sound System"
                        || hittedGameObjectName == "Sound System_2"
                        || hittedGameObjectName == "Sound System_3"
                        || hittedGameObjectName == "Sound System_4"
                        || hittedGameObjectName == "Sound System_5"
                        || hittedGameObjectName == "Sound System(Clone)")
                    {
                        actionPosition = new float3(hit.point.x, .5f, hit.point.z);
                        hit.collider.gameObject.GetComponent<InformationAnimationSoundSystem>().enabled = false;
                        hit.collider.gameObject.transform.GetChild(0).gameObject.SetActive(false);
                    }
                    hittedGameObjectName = null;
                }
            }

            // Create EnablePanicModeJob which creates a panic reaction around the action effect
            EnablePanicModeJob enablePrePanicJob = new EnablePanicModeJob
            {
                actionPosition = actionPosition,
                panicRadius = 10f,
            };

            // Schedule the EnablePanicModeJob Job with starting deps, save results into JobHandle
            jobHandle = enablePrePanicJob.Schedule(this, inputDeps);
        }

        // When choosen, always start a panic job to simulate panic behavior
        // Create Panic Job
        PanicJob panicJob = new PanicJob
        {
            RandomGenerator = RandomGenerator,
            exitsTranslations = exitsTranslations,
            exitsExitComponents = exitsExitComponents,
            quadrantMultiHashMap = QuadrantSystem.quadrantMultiHashMap
        };

        // Schedule Panic Job
        jobHandle = panicJob.Schedule(this, jobHandle);
    }
```

```
        // If smallGroundExplosion Icon was choosen
        else if (Actions.instance.fallingTruss)
        {
            // If User pressed left mouse button down
            if (UnityEngine.Input.GetMouseButtonDown(0))
            {
                // Handle the mouse position and pass it to the EnablePanicModeJob later
                // Disable Information Arrows
                actionPosition = UnityEngine.Input.mousePosition;
                UnityEngine.Ray ray = UnityEngine.Camera.main.ScreenPointToRay(actionPosition);
                if (UnityEngine.Physics.Raycast(ray, out UnityEngine.RaycastHit hit))
                {
                    if (hit.collider != null)
                    {
                        actionPosition = hit.collider.gameObject.transform.position; // The position of the falling truss
Game Object
                        hit.collider.gameObject.GetComponent<InformationAnimationTruss>().enabled = false;
                        hit.collider.gameObject.transform.GetChild(0).gameObject.SetActive(false);
                    }
                }
            }

            // This bool will be true, when the falling truss animation has completed with Unity events
            // Only create the panic reaction when the truss has fallen
            if (Actions.instance.trussHasFallen)
            {
                // Create EnablePanicModeJob which creates a panic reaction around the action effect
                EnablePanicModeJob enablePrePanicJob = new EnablePanicModeJob
                {
                    actionPosition = actionPosition,
                    panicRadius = 25f,
                };

                // Schedule the EnablePanicModeJob Job with starting deps, save results into JobHandle
                jobHandle = enablePrePanicJob.Schedule(this, inputDeps);
            }

            // When choosen, always start a panic job to simulate panic behavior
            // Create Panic Job
            PanicJob panicJob = new PanicJob
            {
                RandomGenerator = RandomGenerator,
                exitsTranslations = exitsTranslations,
                exitsExitComponents = exitsExitComponents,
                quadrantMultiHashMap = QuadrantSystem.quadrantMultiHashMap
            };

            // Schedule Panic Job
            jobHandle = panicJob.Schedule(this, jobHandle);
        }

        // Case: No Exits, Agents spawned, action placed, 2 pressed to remove all agents, 1 pressed to spawn agents
again -> panic still enabled
        // Only start the ReactOnPanicInsideQuadrantJob when allowed (mouse pressed down)
        if (ManagerSystem.actionUsed)
        {
            // Only React on panic when panic action is enabled
            ReactOnPanicInsideQuadrantJob reactOnPanicInsideQuadrantJob = new ReactOnPanicInsideQuadrantJob
            {
                quadrantMultiHashMap = QuadrantSystem.quadrantMultiHashMap,
                exitsTranslations = exitsTranslations,
                exitsExitComponents = exitsExitComponents,
```

```
                actionPosition = actionPosition,
                RandomGenerator = RandomGenerator,
                CommandBuffer = m_EntityCommandBufferSystem.CreateCommandBuffer().ToConcurrent(), // Create
the commandBuffer
            };

            // Schedule ReactOnPanicInsideQuadrantJob
            jobHandle = reactOnPanicInsideQuadrantJob.Schedule(this, jobHandle);

            // Execute the commandBuffer commands when reactOnPanicInsideQuadrantJob is finished
            m_EntityCommandBufferSystem.AddJobHandleForProducer(jobHandle);
        }

    }
    // For writing on the multiHashMap (Quadrant System)
    jobHandle.Complete();
    return jobHandle;
  }
```

[A36]
Das Quadrant System
20.12.2019

```
/// <summary>
/// Needed for calculating Quadrant Data. Helps to spread panic.
/// </summary>
public struct QuadrantEntity : IComponentData
{
  public TypeEnum typeEnum;

  /// <summary>
  /// Identify which entity is a Agent and which one is a exit.
  /// </summary>
  public enum TypeEnum
  {
    Agent,
    Exit
  }
}

/// <summary>
/// These are saved for each Quadrant.
/// </summary>
public struct QuadrantData
{
  public Entity entity;
  public float3 position;
  public AgentComponent agentComponent;
  public QuadrantEntity quadrantEntity;
}

/// <summary>
/// System that handles the whole Quadrant System.
/// </summary>
public class QuadrantSystem : JobComponentSystem
{
  #region Variables
  public const int quadrantYMultiplier = 1000; // Enough for this level/Simulation size
  private const float quadrantCellSize = 1f; // Cell Size of each quadrant
  public static NativeMultiHashMap<int, QuadrantData> quadrantMultiHashMap; // For accessing the
quadrantMultiHashMap from other Systems and MonoBehavior classes
  #endregion // Variables
```

```
    /// <summary>
    /// Function that takes a position to calculate a individual key with basic math. This way, each position has an individual key to save and load data from.
    /// </summary>
    /// <param name="position">position that is used to calculate an individual key</param>
    /// <returns></returns>
    public static int GetPositionHashMapKey(float3 position)
    {
        return (int)(math.floor(position.x / quadrantCellSize) + (quadrantYMultiplier * math.floor(position.z / quadrantCellSize)));
    }

    #region Debug
    /// <summary>
    /// Debug Method that can be used to display a quadrant. The Quadrant can be seen when enabling DebugCube in the Inspector.
    /// A static GameObject cube is more accurate than the current global mouse position.
    /// </summary>
    /// <param name="position">Current position of the DebugCube</param>
    private static void DebugDrawQuadrant(Vector3 position)
    {
        Vector3 lowerLeft = new Vector3(math.floor(position.x / quadrantCellSize) * quadrantCellSize, 0.5f, math.floor(position.z / quadrantCellSize) * quadrantCellSize);
        Debug.DrawLine(lowerLeft, lowerLeft + new Vector3(+1, +0, +0) * quadrantCellSize);
        Debug.DrawLine(lowerLeft, lowerLeft + new Vector3(+0, +0, +1) * quadrantCellSize);
        Debug.DrawLine(lowerLeft + new Vector3(+1, +0, +0) * quadrantCellSize, lowerLeft + new Vector3(+1, +0, +1) * quadrantCellSize);
        Debug.DrawLine(lowerLeft + new Vector3(+0, +0, +1) * quadrantCellSize, lowerLeft + new Vector3(+1, +0, +1) * quadrantCellSize);
        //Debug.Log(GetPositionHashMapKey(position) + " " + position);
    }
    #endregion // Debug
    /// <summary>
    /// Function that calculates how many Entitys are inside a quadrant.
    /// </summary>
    /// <param name="quadrantMultiHashMap">All keys with all QuadrantData</param>
    /// <param name="hashMapKey">A specific key</param>
    /// <returns>entity count inside a quadrant</returns>
    public static int GetEntityCountInHashMap(NativeMultiHashMap<int, QuadrantData> quadrantMultiHashMap, int hashMapKey)
    {
        QuadrantData quadrantData;
        NativeMultiHashMapIterator<int> nativeMultiHashMapIterator;
        int count = 0;
        if (quadrantMultiHashMap.TryGetFirstValue(hashMapKey, out quadrantData, out nativeMultiHashMapIterator))
        {
            do
            {
                count++;
            } while (quadrantMultiHashMap.TryGetNextValue(out quadrantData, ref nativeMultiHashMapIterator));
        }
        return count;
    }

    /// <summary>
    /// Job that runs on each entity with AgentComponent, Translation and QuadrantEntity Component.
    /// Calculates a quadrant key for their actual position.
    /// Use this key to save a quadrant with specific data.
    /// Specific for Agents.
    /// </summary>
    [BurstCompile]
```

```
    private struct SetQuadrantDataHashMapJob : IJobForEachWithEntity<AgentComponent, Translation,
QuadrantEntity>
    {
        public NativeMultiHashMap<int, QuadrantData>.Concurrent quadrantMultiHashMap;

        public void Execute(Entity entity, int index, [ReadOnly] ref AgentComponent agentComponent, [ReadOnly] ref
Translation translation, [ReadOnly] ref QuadrantEntity quadrantEntity)
        {
            int hashMapKey = GetPositionHashMapKey(translation.Value);
            quadrantMultiHashMap.Add(hashMapKey, new QuadrantData
            {
                entity = entity,
                position = translation.Value,
                agentComponent = agentComponent,
                quadrantEntity = quadrantEntity
            });
        }
    }

    /// <summary>
    /// Job that runs on each Entity with ExitComponent, Translation and QuadrantEntity Component.
    /// Calculates a quadrant key for their actual position.
    /// Use this key to save a quadrant with specific data.
    /// Specific for exits.
    /// </summary>
    [BurstCompile]
    private struct SetQuadrantDataHashMapJobForExits : IJobForEachWithEntity<ExitComponent, Translation,
QuadrantEntity>
    {
        public NativeMultiHashMap<int, QuadrantData>.Concurrent quadrantMultiHashMap;

        public void Execute(Entity entity, int index, [ReadOnly] ref ExitComponent exitComponent, [ReadOnly] ref
Translation translation, [ReadOnly] ref QuadrantEntity quadrantEntity)
        {
            int hashMapKey = GetPositionHashMapKey(translation.Value);
            quadrantMultiHashMap.Add(hashMapKey, new QuadrantData
            {
                entity = entity,
                position = translation.Value,
                quadrantEntity = quadrantEntity
            });
        }
    }

    /// <summary>
    /// Creates the main quadrantMultiHashMap for the quadrant System at the beginning.
    /// </summary>
    protected override void OnCreate()
    {
        quadrantMultiHashMap = new NativeMultiHashMap<int, QuadrantData>(0, Allocator.Persistent);
        base.OnCreate();
    }

    /// <summary>
    /// Disposes the main quadrantMultiHashMap for the quadrant System at the end.
    /// </summary>
    protected override void OnDestroy()
    {
        quadrantMultiHashMap.Dispose();
        base.OnDestroy();
    }

    /// <summary>
```

```
    /// Main Thread section, where Jobs are called and connected.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>JobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        // Get All entitys with QuadrantEntity and Translation Component
        EntityQuery entityQuery = GetEntityQuery(typeof(QuadrantEntity), typeof(Translation));

        quadrantMultiHashMap.Clear(); // because of the persistent hasMap
        if (entityQuery.CalculateEntityCount() > quadrantMultiHashMap.Capacity)
        {
            //  prevent hasMap is full error, hashmaps cannot dynamically grow in jobs. need to to this here instead,
before starting the specifify job
            quadrantMultiHashMap.Capacity = entityQuery.CalculateEntityCount();
        }

        // Create SetQuadrantDataHashMapJob to load every Agent Quadrant into the quadrantMultiHashMap
        SetQuadrantDataHashMapJob setQuadrantDataHashMapJob = new SetQuadrantDataHashMapJob
        {
            quadrantMultiHashMap = quadrantMultiHashMap.ToConcurrent()
        };

        // Schedule this job and save deps into jobHandle
        JobHandle jobHandle = setQuadrantDataHashMapJob.Schedule(this, inputDeps);

        // Create SetQuadrantDataHashMapJobForExits to load every Exit Quadrant into the quadrantMultiHashMap
        SetQuadrantDataHashMapJobForExits setQuadrantDataHashMapJobForExits = new
SetQuadrantDataHashMapJobForExits
        {
            quadrantMultiHashMap = quadrantMultiHashMap.ToConcurrent()
        };

        // Schedule this job, use the current jobHandle as deps and save the new deps into jobHandle again
        jobHandle = setQuadrantDataHashMapJobForExits.Schedule(this, jobHandle);
        //jobHandle.Complete();

        #region Debug
        //var mousePosition = Input.mousePosition;
        //mousePosition.z = Mathf.Abs(Camera.main.gameObject.transform.position.z);
        //var worldMousePosition = Camera.main.ScreenToWorldPoint(mousePosition);
        //DebugDrawQuadrant(GameObject.Find("QuadrantDebugCube").transform.position);
        //Debug.Log(GetEntityCountInHashMap(quadrantMultiHashMap,
GetPositionHashMapKey(GameObject.Find("QuadrantDebugCube").transform.position)));
        #endregion // Debug
        return jobHandle;
    }
```

[A37]
Das Mass System
21.12.2019

```
/// <summary>
/// System that handles the overloaded bools for every single exit entity.
/// </summary>
public class MassSystem : JobComponentSystem
{
    /// <summary>
    /// Job that handles the overloaded bools for every single exit entity.
    /// </summary>
```

```
[BurstCompile]
public struct CalculateEntitiesAroundExitJob : IJobForEachWithEntity<ExitComponent, Translation>
{
    // Data from main thread
    [ReadOnly] public NativeMultiHashMap<int, QuadrantData> nativeMultiHashMap;

    /// <summary>
    /// The actual calculation of the Entity amound around an exit entity.
    /// If the calculated amound is greater than x, set the overloaded bool to true.
    /// Other Systems will react to this bool.
    /// </summary>
    /// <param name="entity">Current Entity</param>
    /// <param name="index">Current Entity index</param>
    /// <param name="_exitComponent">Current Entity ExitComponent</param>
    /// <param name="_translation">Current Entity Translation Component</param>
    public void Execute(Entity entity, int index, [WriteOnly] ref ExitComponent _exitComponent, [ReadOnly] ref
Translation _translation)
    {
        int amount = 0;

        // Sum the amound of each quadrant around the exit entity
        amount = QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value)); // This quadrant
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) + 1); // Right quadrant
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) - 1); // Left quadrant
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) + QuadrantSystem.quadrantYMultiplier); // Above
quadrant
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) - QuadrantSystem.quadrantYMultiplier); // Below
quadrant

        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) + 1 + QuadrantSystem.quadrantYMultiplier); // Corner
Top Right
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) - 1 + QuadrantSystem.quadrantYMultiplier); // Corner
Top Left
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) + 1 - QuadrantSystem.quadrantYMultiplier); // Corner
Bottom Right
        amount += QuadrantSystem.GetEntityCountInHashMap(nativeMultiHashMap,
QuadrantSystem.GetPositionHashMapKey(_translation.Value) - 1 - QuadrantSystem.quadrantYMultiplier); // Corner
Bottom Left
        _exitComponent.amount = amount;

        if (amount > 10)
        {
            // Amount is greater than x, set overloaded to true
            _exitComponent.overloaded = true;
        }
        else
        {
            // Amount is less than x, set overloaded to false
            _exitComponent.overloaded = false;
        }
    }
}

/// <summary>
/// Main Thread section, where Jobs are called and connected.
```

```
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        // Create CalculateEntitysAroundExitJob
        CalculateEntitiesAroundExitJob calculateEntitysAroundExitJob = new CalculateEntitiesAroundExitJob
        {
            nativeMultiHashMap = QuadrantSystem.quadrantMultiHashMap
        };

        // Schedule CalculateEntitysAroundExitJob with starting deps
        JobHandle jobHandle = calculateEntitysAroundExitJob.Schedule(this, inputDeps);

        jobHandle.Complete(); // because other jobs need access to the nativeMultiHashMap
        return jobHandle;
    }
```

[A38]
Das Update Borders System
21.12.2019

```
/// <summary>
/// System that will be activated when border data changes. It will save the current border position of each corner.
/// </summary>
public class UpdateBordersSystem : JobComponentSystem
{
    /// <summary>
    /// Job that will be executed on every entity with BorderComponent.
    /// From main thread this Job gets the actual Border Data from MonoBehavior, this data will be transfered to ECS world.
    /// Save the MonoBehavior Data in BorderComponent of each entity with this Component.
    /// </summary>
    [BurstCompile]
    struct UpdateBordersJob : IJobForEachWithEntity<BorderComponent>
    {
        // Data from main thread
        [ReadOnly] public float3 frontLeft;
        [ReadOnly] public float3 frontRight;
        [ReadOnly] public float3 backLeft;
        [ReadOnly] public float3 backRight;

        /// <summary>
        /// Assign Border data to Border Components.
        /// </summary>
        /// <param name="entity">Current Entity</param>
        /// <param name="index">Current Entity ID</param>
        /// <param name="_borderComponent">BorderComponent of entity x</param>
        public void Execute(Entity entity, int index, [WriteOnly] ref BorderComponent _borderComponent)
        {
            // Assign each value to the borderComponent of each entity with this Component
            _borderComponent.frontLeft = frontLeft;
            _borderComponent.frontRight = frontRight;
            _borderComponent.backLeft = backLeft;
            _borderComponent.backRight = backRight;
        }
    }

    /// <summary>
    /// Job that runs after UpdateBordersJob to disable this Job. This way, [BurstCompile] can be used at UpdateBordersJob.
```

```
    /// The reason for this is that Burst cannot handle World access.
    /// </summary>
    struct DisableUpdateBordersSystemJob : IJobForEach<BorderComponent>
    {
        public void Execute([ReadOnly] ref BorderComponent _borderComponent)
        {
            // Disable this whole System
            World.Active.GetExistingSystem<UpdateBordersSystem>().Enabled = false;
        }
    }

    /// <summary>
    /// Main Thread section, where Jobs are called and connected.
    /// This OnUpdate only runs one time.
    /// Because of the DisableUpdateBordersJob, the whole System will be disabled.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>JobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        JobHandle jobHandle = new JobHandle();

        // Only start UpdateBordersJob when InputField of the Input window is not focused
        if (!InputWindow.instance.inputField.isFocused)
        {
            // Create a UpdateBordersJob and find each Corner GameObject in MonoBehavior
            UpdateBordersJob updateBordersJob = new UpdateBordersJob
            {
                frontLeft = UnityEngine.GameObject.Find("SpawnPoint_Front_Left").transform.position,
                frontRight = UnityEngine.GameObject.Find("SpawnPoint_Front_Right").transform.position,
                backLeft = UnityEngine.GameObject.Find("SpawnPoint_Back_Left_1").transform.position,
                backRight = UnityEngine.GameObject.Find("SpawnPoint_Back_Right_1").transform.position
            };

            // Schedule this Job with starting deps
            jobHandle = updateBordersJob.Schedule(this, inputDeps);

            // After that, create a DisableUpdateBordersJob to disable this whole System when BorderComponents are
updated
            DisableUpdateBordersSystemJob disableUpdateBordersJob = new DisableUpdateBordersSystemJob
            {
            };

            // Schedule this Job with current jobHandle
            jobHandle = disableUpdateBordersJob.Schedule(this, jobHandle);
        }
        return jobHandle;
    }
```

[A39]
Das Remove Agents System
21.12.2019

```
<...>
/// <summary>
    /// Remove all entities with AgentComponent and InputComponent.
    /// </summary>
    [BurstCompile]
    struct RemoveAgentsJob : IJobForEachWithEntity<AgentComponent, InputComponent>
    {
        // Data from main thread
```

```
        // instantiating and deleting of Entitys can only gets done on the main thread, save commands in buffer for
main thread
        public EntityCommandBuffer.Concurrent CommandBuffer;

        public void Execute(Entity entity, int index, [ReadOnly] ref AgentComponent _agentComponent, [ReadOnly] ref
InputComponent _inputComponent)
        {
            CommandBuffer.DestroyEntity(index, entity);
        }
    }

    /// <summary>
    /// Job that handles commands that cannot be done inside a [BurstCompile] Job.
    /// This is the reason why here is no [BurstCompile] Tag.
    /// </summary>
    struct DisableRemoveAgentsSystemJob : IJobForEach<AgentComponent>
    {
        public void Execute(ref AgentComponent _agentComponent)
        {
            // Disable this whole System
            World.Active.GetExistingSystem<RemoveAgentsSystem>().Enabled = false;
        }
    }

    /// <summary>
    /// This OnUpdate only runs one time.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        JobHandle jobHandle = new JobHandle();

        // Creating RemoveAgentsJob
        RemoveAgentsJob removeAgentsJob = new RemoveAgentsJob
        {
            // Create the commandBuffer
            CommandBuffer = m_EntityCommandBufferSystem.CreateCommandBuffer().ToConcurrent(),
        };

        // Schedule this job and save the results into jobHandle
        jobHandle = removeAgentsJob.Schedule(this, inputDeps);

        // Execute the commandBuffer commands when removeAgentsJob is finished
        m_EntityCommandBufferSystem.AddJobHandleForProducer(jobHandle);

        // Mono Behavior stuff
        // If this is not set, all new agents would instantly set to AgentStatus = AgentsStatus.Running
        ManagerSystem.actionUsed = false;

        // Get all children of the Actions GameObject and destroy them.
        // The children are all active Actions (burning fires)
        UnityEngine.GameObject actions = UnityEngine.GameObject.Find("Actions");
        int childs = actions.transform.childCount;
        for (int i = childs - 1; i >= 0; i--)
        {
            UnityEngine.GameObject.Destroy(actions.transform.GetChild(i).gameObject);
        }

        // Creating DisableRemoveAgentsSystemJob to disable this whole System
        DisableRemoveAgentsSystemJob disableRemoveAgentsSystemJob = new DisableRemoveAgentsSystemJob
        {
        };
```

```
        // Schedule this job and save the results into jobHandle
        jobHandle = disableRemoveAgentsSystemJob.Schedule(this, jobHandle);

        return jobHandle;
    }
```

[A40]
Das Remove Exits System
21.12.2019

```
/// <summary>
    /// Deleting all Entities with ExitComponent.
    /// </summary>
    [BurstCompile]
    struct RemoveExitsJob : IJobForEachWithEntity<ExitComponent>
    {
        // Data from main thread
        // Instantiating and deleting of Entitys can only get done on the main thread. Save commands in buffer for main
thread later
        public EntityCommandBuffer.Concurrent CommandBuffer;

        public void Execute(Entity entity, int index, [ReadOnly] ref ExitComponent _exitComponent)
        {
            CommandBuffer.DestroyEntity(index, entity);
        }
    }

    /// <summary>
    /// Job that handles commands that cannot be done inside a [BurstCompile] Job.
    /// This is the reason why here is no [BurstCompile] Tag.
    /// </summary>
    struct DisableRemoveExitsSystemJob : IJobForEach<ExitComponent>
    {
        public void Execute(ref ExitComponent _exitComponent)
        {
            // Disable this whole System
            World.Active.GetExistingSystem<RemoveExitsSystem>().Enabled = false;
        }
    }

    /// <summary>
    /// This OnUpdate only runs one time, so GameObject.Find will be no problem here.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        JobHandle jobHandle = new JobHandle();

        if (!InputWindow.instance.inputField.isFocused)
        {
            // If InputField is not focused
            // Get all pole GameObjects
            var poleClones = UnityEngine.Resources.FindObjectsOfTypeAll<UnityEngine.GameObject>().Where(obj =>
obj.name == "GroundPoleA(Clone)");

            foreach (UnityEngine.GameObject pole in poleClones)
            {
                if (pole != null)
                {
```

```
                // The poles are childs from a parent GameObject.
                // Get this GameObject, get the first child from this GameObject which holds a MeshRenderer, which
displays the whole GameObject
                // Enable this MeshRenderer.
                // Not the best implementation but ECS/Jobs do not offer a better solution.
                pole.transform.parent.GetChild(0).GetComponent<UnityEngine.MeshRenderer>().enabled = true;
                UnityEngine.Object.Destroy(pole);
            }
        }

        // Creating RemoveExitsJob
        RemoveExitsJob removeExitsJob = new RemoveExitsJob
        {
            // Create the commandBuffer
            CommandBuffer = m_EntityCommandBufferSystem.CreateCommandBuffer().ToConcurrent(),
        };

        // Scheduling RemoveExitsJob and save the results into jobHandle
        jobHandle = removeExitsJob.Schedule(this, inputDeps);

        // Load CommandBuffercommands into main thread
        // Execute the commandBuffer commands when spawnJob is finished
        m_EntityCommandBufferSystem.AddJobHandleForProducer(jobHandle);

        // Create DisableRemoveExitsSystemJob to disable this whole System
        DisableRemoveExitsSystemJob disableRemoveExitsSystemJob = new DisableRemoveExitsSystemJob
        {
        };

        // Schedule this job and save the results into jobHandle
        jobHandle = disableRemoveExitsSystemJob.Schedule(this, jobHandle);
      }
      return jobHandle;
    }
```

[A41]
Das Manager System
21.12.2019

```
/// <summary>
/// Helper System for the other Systems.
/// </summary>
[UpdateBefore(typeof(UnitSpawnerSystem))]
public class ManagerSystem : JobComponentSystem
{
    // For checking if an action was used
    // Can be used in other Systems
    public static bool actionUsed = false;

    /// <summary>
    /// Job that runs on every single agent to calculate human behavior while beeing not in panic mode.
    /// </summary>
    [BurstCompile]
    struct ManagerJob : IJobForEachWithEntity<AgentComponent, Translation>
    {
        // Data from main thread
        [NativeDisableParallelForRestriction] // Enables writing to any index of RandomGenerator
        [DeallocateOnJobCompletion]
        public NativeArray<Random> RandomGenerator; // For generating random values inside this job

        [Unity.Collections.LowLevel.Unsafe.NativeSetThreadIndex]
        [ReadOnly]
```

```
    private int threadIndex; // For generating individual random values inside this job

    public bool actionUsed; // To check if an action has been used. So this System stops working on. (It's still online but it cannot pass the if case)

    /// <summary>
    /// Generate random values and assign them to different random behavior for the agents.
    /// </summary>
    /// <param name="entity">Current Entity</param>
    /// <param name="index">Current Entity index</param>
    /// <param name="_agentComponent">Current Entity AgentComponent</param>
    /// <param name="_translation">Current Entity Translation Component</param>
    public void Execute(Entity entity, int index, ref AgentComponent _agentComponent, ref Translation _translation)
    {
        if (!actionUsed)
        {
            // Only Start the random schedule process when the agent is on the festival area and it did not pass an exit yet
            var rnd = RandomGenerator[threadIndex - 1]; // Use the current threadIndex-1 to access a random Random value out of the RandomGenerator Native Array
            var dice = rnd.NextFloat(1000); // Use this random Random value to calculate a real random value for this job: value 0||1||2||...||1000

            // 30% Idle, 50% Jumping, 20% Moving
            if (dice >= 0f && dice <= 3f)
            {
                // Stay where you are
                // Idle mode
                _agentComponent.target = _translation.Value;
                _agentComponent.agentStatus = AgentStatus.Idle;
                _agentComponent.hasTarget = false;
            }
            else if (dice >= 3f && dice <= 8f)
            {
                // Trigger Jumping System to start operating on this agent
                _agentComponent.target = _translation.Value;
                _agentComponent.agentStatus = AgentStatus.Dancing;
                _agentComponent.hasTarget = false;
            }
            else if (dice >= 8 && dice <= 10)
            {
                // Trigger MovingSystem to start operating on this agent
                _agentComponent.target = _translation.Value;
                _agentComponent.agentStatus = AgentStatus.Moving;
                _agentComponent.hasTarget = false;
            }
            RandomGenerator[threadIndex - 1] = rnd; // This is necessary to update the state of the element inside the array.
        }
    }
}

/// <summary>
/// Open/Close different Systems to save performance.
/// No Burst here because of the World.Active access.
/// </summary>
struct ManagerInputJob : IJobForEachWithEntity<InputComponent>
{
    /// <summary>
    /// Enable/Disable different Systems.
    /// </summary>
    /// <param name="entity">Current Entity</param>
```

```
        /// <param name="index">Current Entity index</param>
        /// <param name="_inputComponent">Current Entity Input Component</param>
        public void Execute(Entity entity, int index, ref InputComponent _inputComponent)
        {
            if (_inputComponent.keyOnePressedDown)
            {
                // Enable/Disable the RemoveAgentsSystem and the UnitSpawnerSystem when agents are spawned with
key 1
                World.Active.GetExistingSystem<RemoveAgentsSystem>().Enabled = false;
                World.Active.GetExistingSystem<UnitSpawnerSystem>().Enabled = true;
            }
            if (_inputComponent.keyFivePressedUp || _inputComponent.keySixPressedUp ||
_inputComponent.keyThreePressedUp || _inputComponent.keyFourPressedUp)
            {
                //Add/Remove/Change Barriers
                //Enable System that updates the spawn objects from the border component
                World.Active.GetExistingSystem<UpdateBordersSystem>().Enabled = true;
            }
            if (_inputComponent.keyTwoPressedUp)
            {
                // Enable the RemoveAgentsSystem to remove all agents
                World.Active.GetExistingSystem<RemoveAgentsSystem>().Enabled = true;
            }
            if (_inputComponent.keySevenPressedUp)
            {
                // Enable the RemoveExitsSystem to remove all exits
                World.Active.GetExistingSystem<RemoveExitsSystem>().Enabled = true;
            }
        }
    }

    /// <summary>
    /// Job that updates the user created entity amount in DOTS world.
    /// </summary>
    [BurstCompile]
    struct UpdateEntityAmount : IJobForEachWithEntity<UnitSpawnerComponent>
    {
        // Data from main thread
        public int newAmountToSpawn;

        /// <summary>
        /// Update the current Entity amount.
        /// </summary>
        /// <param name="entity">Current Entity</param>
        /// <param name="index">Current Entity index</param>
        /// <param name="_unitSpawnerComponent">Current Entity unitSpawnerComponent</param>
        public void Execute(Entity entity, int index, ref UnitSpawnerComponent _unitSpawnerComponent)
        {
            _unitSpawnerComponent.AmountToSpawn = newAmountToSpawn;
        }
    }

    // Variables for not creating new ones each time OnUpdate restarts
    #region Variables
    Random Rnd = new Random(1);
    NativeArray<Random> RandomGenerator;
    #endregion // Variables

    /// <summary>
    /// Main Thread section, where Jobs are called and connected.
    /// </summary>
    /// <param name="inputDeps">starting deps</param>
    /// <returns>jobHandle</returns>
```

```
protected override JobHandle OnUpdate(JobHandle inputDeps)
{
    // Initialize Native Array with the processorCount length and the TempJob Allocator tag
    RandomGenerator = new NativeArray<Random>(System.Environment.ProcessorCount, Allocator.TempJob);

    // Set actionUsed to true when an action was used
    if (Actions.instance.actionEnabled)
    {
        if (UnityEngine.Input.GetMouseButtonDown(0))
        {
            actionUsed = true;
        }
    }

    // Fill the RandomGenerator with random Random objects
    for (int i = 0; i < RandomGenerator.Length; i++)
    {
        RandomGenerator[i] = new Random((uint)Rnd.NextInt());
    }

    // Create ManagerInputJob
    ManagerInputJob managerInputJob = new ManagerInputJob
    {
    };

    // Schedule ManagerInputJob with starting deps, save in jobHandle
    JobHandle jobHandle = managerInputJob.Schedule(this, inputDeps);

    // Create ManagerJob
    ManagerJob managerJob = new ManagerJob
    {
        RandomGenerator = RandomGenerator,
        actionUsed = actionUsed
    };

    // Schedule ManagerJob with jobHandle, save in jobHandle
    jobHandle = managerJob.Schedule(this, jobHandle);

    // Create UpdateEntityAmount Job
    UpdateEntityAmount updateEntityAmount = new UpdateEntityAmount
    {
        newAmountToSpawn = UnitSpawnerProxy.instance.AmountToSpawn
    };

    // Schedule UpdateEntityAmount Job with current jobHandle, save in jobHandle
    jobHandle = updateEntityAmount.Schedule(this, jobHandle);

    return jobHandle;
}
```

[A42]
Das Information Animation Truss Skript
23.12.2019

```
/// <summary>
/// Functionality script for Falling Trusses.
/// </summary>
public class InformationAnimationTruss : MonoBehaviour
{
    #region Variables
    // GameObjects
```

```
    private GameObject informationArrow; // The Child GameObject that contains the Information Arrow
    #endregion // Variables

    void Start()
    {
        // Get whole arrow Game Object (first in hierarchy)
        informationArrow = transform.GetChild(0).gameObject;
    }

    void Update()
    {
        EnableDisableArrows();
    }

    /// <summary>
    /// React on enableTrussArrows, set whole arrow Game Object On/Off.
    /// </summary>
    private void EnableDisableArrows()
    {
        if (UIHandler.instance.enableTrussArrows)
        {
            // If this bool is true, enable the actual animation and enable the whole child GameObject
            informationArrow.GetComponent<Animator>().SetBool("InformationArrowTruss", true);
            informationArrow.SetActive(true);
        }
        else
        {
            // If this bool is false, disable the actual animation and disable the whole Child GameObject
            informationArrow.GetComponent<Animator>().SetBool("InformationArrowTruss", false);
            informationArrow.SetActive(false);
        }
    }
```

[A43]
Das Information Animation Sound System Skript
23.12.2019

```
/// <summary>
/// Functionality script for Sound Systems.
/// </summary>
public class InformationAnimationSoundSystem : MonoBehaviour
{
    #region Variables
    // GameObjects
    private GameObject informationArrow; // The Child GameObject that contains the Information Arrow
    #endregion // Variables

    void Start()
    {
        // Get whole arrow Game Object (first in hierarchy)
        informationArrow = transform.GetChild(0).gameObject;
    }

    void Update()
    {
        EnableDisableArrows();
    }

    /// <summary>
    /// React on enableSoundSystemArrows, set whole arrow Game Object On/Off.
    /// </summary>
    private void EnableDisableArrows()
```

```
    {
        if (UIHandler.instance.enableSoundSystemArrows)
        {
            // If this bool is true, enable the actual animation and enable the whole child GameObject
            informationArrow.GetComponent<Animator>().SetBool("InformationArrowSoundSystem", true);
            informationArrow.SetActive(true);
        }
        else
        {
            // If this bool is false, disable the actual animation and disable the whole Child GameObject
            informationArrow.GetComponent<Animator>().SetBool("InformationArrowSoundSystem", false);
            informationArrow.SetActive(false);
        }
    }
```

[A44]
Das Action After Falling Skript
23.12.2019

```
/// <summary>
/// Class that holds methods that will be called when Unity event in animation is triggered.
/// </summary>
public class ActionAfterFalling : MonoBehaviour
{
    #region Variables
    // Prefab GameObjects
    [SerializeField] private GameObject explosionPrefab;
    [SerializeField] private GameObject firePrefab;
    private GameObject actionsGameObject;
    #endregion // Variables

    private void Awake()
    {
        actionsGameObject = GameObject.Find("Actions");
    }

    /// <summary>
    /// Instantaite Explosion and fire effects when falling.
    /// </summary>
    /// <returns></returns>
    private IEnumerator ExplodeAfterFallingForeground()
    {
        GameObject explosionOne = Instantiate(explosionPrefab, new Vector3(transform.position.x, 0.5f,
transform.position.z + 6f), transform.rotation);
        yield return new WaitForSeconds(0.1f);
        GameObject explosionTwo = Instantiate(explosionPrefab, new Vector3(transform.position.x, 0.5f,
transform.position.z + 12f), transform.rotation);
        yield return new WaitForSeconds(0.1f);
        GameObject explosionThree = Instantiate(explosionPrefab, new Vector3(transform.position.x, 0.5f,
transform.position.z + 18f), transform.rotation);

        yield return new WaitForSeconds(1f);

        GameObject fireOne = Instantiate(firePrefab, new Vector3(transform.position.x, 0.5f, transform.position.z +
6f), transform.rotation);
        GameObject fireTwo = Instantiate(firePrefab, new Vector3(transform.position.x, 0.5f, transform.position.z +
12f), transform.rotation);
        GameObject fireThree = Instantiate(firePrefab, new Vector3(transform.position.x, 0.5f, transform.position.z +
18f), transform.rotation);

        fireOne.transform.SetParent(actionsGameObject.transform);
```

```
        fireTwo.transform.SetParent(actionsGameObject.transform);
        fireThree.transform.SetParent(actionsGameObject.transform);

        Destroy(explosionOne, 3f);
        Destroy(explosionTwo, 3f);
        Destroy(explosionThree, 3f);

        Destroy(fireOne, 300f);
        Destroy(fireTwo, 300f);
        Destroy(fireThree, 300f);
    }

    /// <summary>
    /// Instantaite Explosion and fire effects when falling.
    /// </summary>
    /// <returns></returns>
    private IEnumerator ExplodeAfterFallingLeftSide()
    {
        GameObject explosionOne = Instantiate(explosionPrefab, new Vector3(transform.position.x - 6f, 0.5f,
transform.position.z), transform.rotation);
        yield return new WaitForSeconds(0.1f);
        GameObject explosionTwo = Instantiate(explosionPrefab, new Vector3(transform.position.x - 12f, 0.5f,
transform.position.z), transform.rotation);
        yield return new WaitForSeconds(0.1f);
        GameObject explosionThree = Instantiate(explosionPrefab, new Vector3(transform.position.x - 18f, 0.5f,
transform.position.z), transform.rotation);

        yield return new WaitForSeconds(1f);

        GameObject fireOne = Instantiate(firePrefab, new Vector3(transform.position.x - 6f, 0.5f,
transform.position.z), transform.rotation);
        GameObject fireTwo = Instantiate(firePrefab, new Vector3(transform.position.x - 12f, 0.5f,
transform.position.z), transform.rotation);
        GameObject fireThree = Instantiate(firePrefab, new Vector3(transform.position.x - 18f, 0.5f,
transform.position.z), transform.rotation);

        fireOne.transform.SetParent(actionsGameObject.transform);
        fireTwo.transform.SetParent(actionsGameObject.transform);
        fireThree.transform.SetParent(actionsGameObject.transform);

        Destroy(explosionOne, 3f);
        Destroy(explosionTwo, 3f);
        Destroy(explosionThree, 3f);

        Destroy(fireOne, 300f);
        Destroy(fireTwo, 300f);
        Destroy(fireThree, 300f);
    }

    /// <summary>
    /// Instantaite Explosion and fire effects when falling.
    /// </summary>
    /// <returns></returns>
    private IEnumerator ExplodeAfterFallingRightSide()
    {
        GameObject explosionOne = Instantiate(explosionPrefab, new Vector3(transform.position.x + 6f, 0.5f,
transform.position.z), transform.rotation);
        yield return new WaitForSeconds(0.1f);
        GameObject explosionTwo = Instantiate(explosionPrefab, new Vector3(transform.position.x + 12f, 0.5f,
transform.position.z), transform.rotation);
        yield return new WaitForSeconds(0.1f);
        GameObject explosionThree = Instantiate(explosionPrefab, new Vector3(transform.position.x + 18f, 0.5f,
transform.position.z), transform.rotation);
```

```
        yield return new WaitForSeconds(1f);

        GameObject fireOne = Instantiate(firePrefab, new Vector3(transform.position.x + 6f, 0.5f,
transform.position.z), transform.rotation);
        GameObject fireTwo = Instantiate(firePrefab, new Vector3(transform.position.x + 12f, 0.5f,
transform.position.z), transform.rotation);
        GameObject fireThree = Instantiate(firePrefab, new Vector3(transform.position.x + 18f, 0.5f,
transform.position.z), transform.rotation);

        fireOne.transform.SetParent(actionsGameObject.transform);
        fireTwo.transform.SetParent(actionsGameObject.transform);
        fireThree.transform.SetParent(actionsGameObject.transform);

        Destroy(explosionOne, 3f);
        Destroy(explosionTwo, 3f);
        Destroy(explosionThree, 3f);

        Destroy(fireOne, 300f);
        Destroy(fireTwo, 300f);
        Destroy(fireThree, 300f);
    }

    /// <summary>
    /// Helper IEnumerator that calls when truss has fallen, needed for other methods.
    /// </summary>
    /// <returns></returns>
    private IEnumerator EnableTrussHasFallenBool()
    {
        yield return new WaitForSeconds(.2f);
        Actions.instance.trussHasFallen = true;
        Actions.instance.actionPlaced = true;
    }
```

[A45]
Das Actions Skript
23.12.2019

```
 /// <summary>
    /// React on action bool changes, when enabled, check for left mouse input. -> Instantiate action/effect when
clicked.
    /// </summary>
    private void CheckActionBools()
    {
        if (smallGroundExplosion)
        {
            // Small Explosions in Radial menu was selected
            var mousePosition = Input.mousePosition;
            Ray ray = Camera.main.ScreenPointToRay(mousePosition);
            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                if (hit.collider != null)
                {
                    // Save current mouse position
                    mousePosition = new Vector3(hit.point.x, 0.5f, hit.point.z);
                }
            }

            if (Input.GetMouseButtonDown(0))
            {
                // Instantiate action when left mouse button was pressed
                GameObject explosion = Instantiate(smallExplosionParticleEffect, mousePosition, Quaternion.identity);
```

```
            actionPlaced = true;
            Destroy(explosion, 3f);
        }
    }
    else if (mediumGroundExplosion)
    {
        // Medium Explosions in Radial menu was selected
        var mousePosition = Input.mousePosition;
        Ray ray = Camera.main.ScreenPointToRay(mousePosition);
        if (Physics.Raycast(ray, out RaycastHit hit))
        {
            if (hit.collider != null)
            {
                // Save current mouse position
                mousePosition = new Vector3(hit.point.x, 0.5f, hit.point.z);
            }
        }

        if (Input.GetMouseButtonDown(0))
        {
            // Instantiate action when left mouse button was pressed
            GameObject explosion = Instantiate(mediumExplosionParticleEffect, mousePosition, Quaternion.identity);
            actionPlaced = true;
            Destroy(explosion, 3f);
        }
    }
    else if (bigGroundExplosion)
    {
        // Medium Explosions in Radial menu was selected
        var mousePosition = Input.mousePosition;
        Ray ray = Camera.main.ScreenPointToRay(mousePosition);
        if (Physics.Raycast(ray, out RaycastHit hit))
        {
            if (hit.collider != null)
            {
                // Save current mouse position
                mousePosition = new Vector3(hit.point.x, 0.5f, hit.point.z);
            }
        }

        if (Input.GetMouseButtonDown(0))
        {
            // Instantiate action when left mouse button was pressed
            GameObject explosion = Instantiate(bigExplosionParticleEffect, mousePosition, Quaternion.identity);
            actionPlaced = true;
            Destroy(explosion, 3f);
        }
    }
    else if (createExits)
    {
        // Exit creater in Radial menu was selected
        if (Input.GetMouseButtonDown(0))
        {
            var mousePosition = Input.mousePosition;
            Ray ray = Camera.main.ScreenPointToRay(mousePosition);
            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                if (hit.collider != null && hit.collider.gameObject.name != "ColliderGround" &&
hit.collider.gameObject.tag != "Truss")
                {
                    // Spawn pin Game Object and set the position to the barrier position
                    GameObject pinGo = Instantiate(pinPrefab, hit.collider.gameObject.transform.position,
Quaternion.identity);
```

```
            pinGo.transform.SetParent(hit.transform.parent);
            Vector3 newPinPosition = new Vector3(
               hit.collider.gameObject.transform.position.x,
               0.0f,
               hit.collider.gameObject.transform.position.z);
            pinGo.transform.position = newPinPosition;

            // Disable barrier GameObject
            // <moved to Input System>

            // Increase Exits Amount
            UIHandler.instance.IncreaseExitsAmount(); // +1 exits amount
         }
      }
   }
}
else if (fallingTruss)
{
   // Falling Truss in Radial menu was selected
   if (Input.GetMouseButtonDown(0))
   {
      var mousePosition = Input.mousePosition;
      Ray ray = Camera.main.ScreenPointToRay(mousePosition);
      if (Physics.Raycast(ray, out RaycastHit hit))
      {
         if (hit.collider.gameObject.name != "ColliderGround")
         {
            // Prevent from getting Animator Component from the ground
            hit.collider.gameObject.GetComponent<Animator>().SetTrigger("isFalling");
         }
      }
   }
}
else if (dropSoundSystem)
{
   // Create Sound System object on mouse position
   if (Input.GetMouseButtonDown(0))
   {
      var mousePosition = Input.mousePosition;
      Ray ray = Camera.main.ScreenPointToRay(mousePosition);
      if (Physics.Raycast(ray, out RaycastHit hit))
      {
         if (hit.collider != null)
         {
            // Instantiate SOund System and set position to mouse position
            GameObject soundSystemObject = Instantiate(soundSystemPrefab);
            UIHandler.instance.userCreatedSoundSystems.Add(soundSystemObject);
            Vector3 newSoundSystemPosition = new Vector3(
               hit.point.x,
               0.5f,
               hit.point.z);
            soundSystemObject.transform.position = newSoundSystemPosition;
            soundSystemPrefab.transform.rotation = Quaternion.identity;

            // Enable lights when they are enabled
            if (!UIHandler.instance.effectsEnabled)
            {
               soundSystemObject.transform.Find("Lights").gameObject.SetActive(false);
            }
            else
            {
               soundSystemObject.transform.Find("Lights").gameObject.SetActive(true);
            }
```

```
            }
        }
    }
}
else if (fire)
{
    // Falling Truss in Radial menu was selected
    if (Input.GetMouseButtonDown(0))
    {
        var mousePosition = Input.mousePosition;
        Ray ray = Camera.main.ScreenPointToRay(mousePosition);
        if (Physics.Raycast(ray, out RaycastHit hit))
        {
            // Only Instantiate fire on Sound Systems
            string hittedGameObjectName = hit.collider.gameObject.name;
            if (hittedGameObjectName == "Sound System"
                || hittedGameObjectName == "Sound System_2"
                || hittedGameObjectName == "Sound System_3"
                || hittedGameObjectName == "Sound System_4"
                || hittedGameObjectName == "Sound System_5"
                || hittedGameObjectName == "Sound System(Clone)")
            {
                Fire(hit.collider.gameObject.transform.position);
                actionPlaced = true;
            }
        }
    }
}
}

/// <summary>
/// Instantiate Fire GameObject and spawns it at parameters position.
/// </summary>
/// <param name="SoundSystemPosition">Spawn position for Sound System</param>
private void Fire(Vector3 SoundSystemPosition)
{
    GameObject fire = Instantiate(firePrefab);
    fire.transform.SetParent(actionsGameObject.transform); // Set the "Actions" GameObject as parent, to have
the option to delete all childs of this parent when deleting all agents later with key 2

    Vector3 newFirePosition = new Vector3(
        SoundSystemPosition.x,
        0.0f,
        SoundSystemPosition.z);
    fire.transform.position = newFirePosition;
}
```

---

github Link zum *Panic-Simulator*
https://github.com/MHallweger/Panic-Simulator

Massenpaniksimulation - Fallender Mast - 3.000 Konzertbesucher
https://www.youtube.com/watch?v=KOZDrBfupnc

Massenpaniksimulation - Feuer - 10.000 Konzertbesucher
https://www.youtube.com/watch?v=7Jj8uNKVCNw

Massenpaniksimulation - 100.000 Konzertbesucher
https://www.youtube.com/watch?v=PhAIUZVdZg4