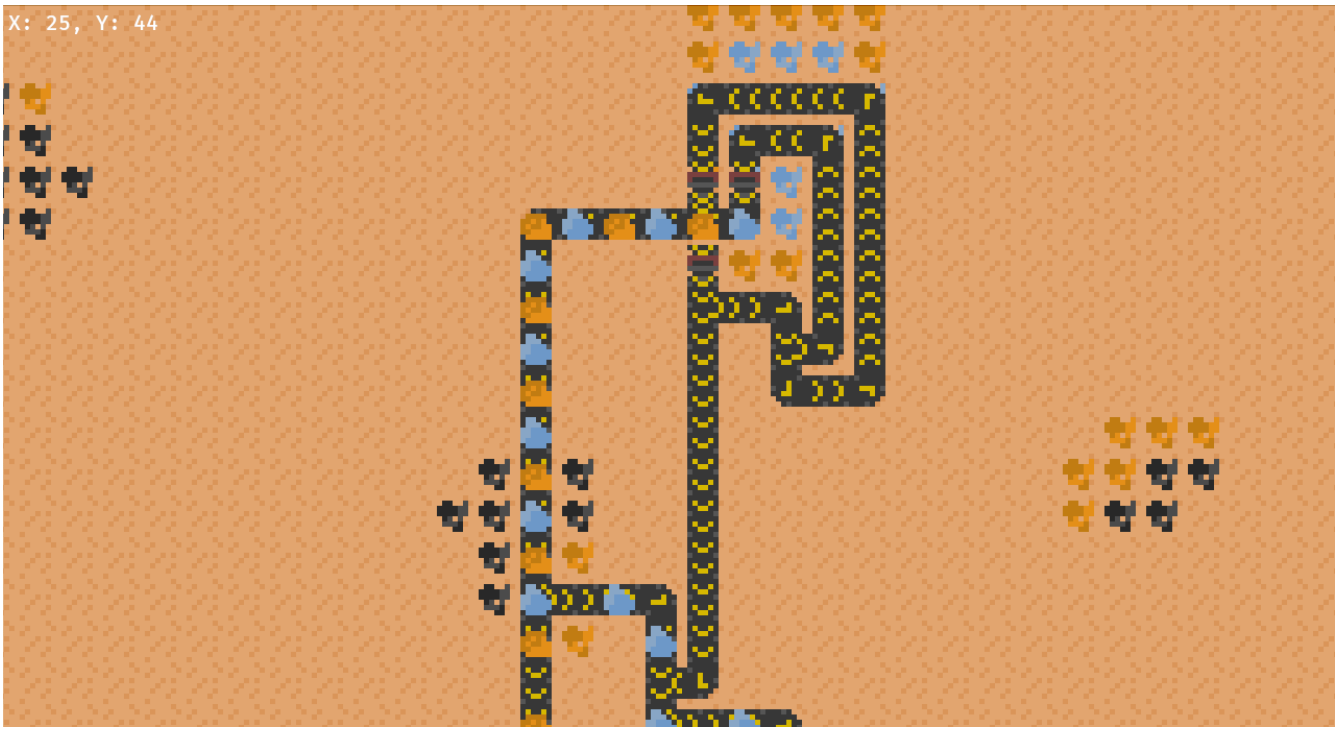

15.06.2025

Spieleprojekt Dokumentation

Planung, Entwicklung und Umsetzung von "Sandy Fact'ry"

Louis Weigel
Klasse: 11BG-PI1
Fach: Praktische Informatik
Lehrer: Danilo Magdeburg

Inhalt

Listings

Tabellen

Abbildungen

1. Einführung und Ziele

1.1. Genre

Das Genre des Spieles ist das eines **Factory Builders**.

In Spielen des Factory Builder-Genres baut man Fabriken auf, um eine geringe Anzahl verschiedener Ressourcen zu komplexeren Gütern zu verarbeiten. Dafür ist effiziente Organisation und Logistik von Förderbändern und oft sogar noch Zügen vonnöten. Ziel ist es, die Fabrik so performant wie möglich zu designen, auch wenn man das eigentliche Spielende schon erreicht hat: *„The Factory must grow!“*[1]

Bekannte Beispiele sind Factorio, das das Genre ins Leben gerufen hat, Satisfactory oder Dyson Sphere Program.

1.2. Ziel des Spiels

Das Ziel des Spiels wird es sein, ein nicht triviales Item/Objekt so effizient wie möglich herzustellen (In diesem Fall wahrscheinlich einen Helikopter).

Dazu wird der Spieler — ähnlich wie in Factorio — Fabriken bauen müssen, um die Herstellung verschiedener Produkte zu automatisieren.

Die Fabriken werden zu Anfang nur aus folgenden Gebäuden gebaut werden, doch es können mehr dazu kommen, wenn die Zeit dafür da ist:

- **Belts:** Bewegen Items hin und her
- **Drills:** Extrahieren Ressourcen aus der Erde
- **Furnaces:** Verarbeiten rohe Ressourcen in Items
- **Crafter:** Verarbeiten Items in andere Items und Objekte

1.3. Schwierigkeiten für den Spieler

Schwierigkeiten für den Spieler werden dabei sein:

- **Logistik:** Wie man Items am besten umherbewegt
- **Effizienz:** Es ist wichtig, Fabriken so zu bauen, dass die Herstellung der benötigten Quantität an Items nicht unnötig viel Zeit in Anspruch nimmt
- (Falls die übriggebliebene Zeit die Erwartung übertrifft) **Gegner:** Feinde, die versuchen, den Fortschritt des Spielers zu verlangsamen

Das Spiel endet, sobald der Spieler einen Helikopter oder ähnliches hergestellt hat.

[1] https://www.reddit.com/r/factorio/comments/ft7jqe/why_do_people_keep_saying_the_factory_must_grow

2. Prototypen

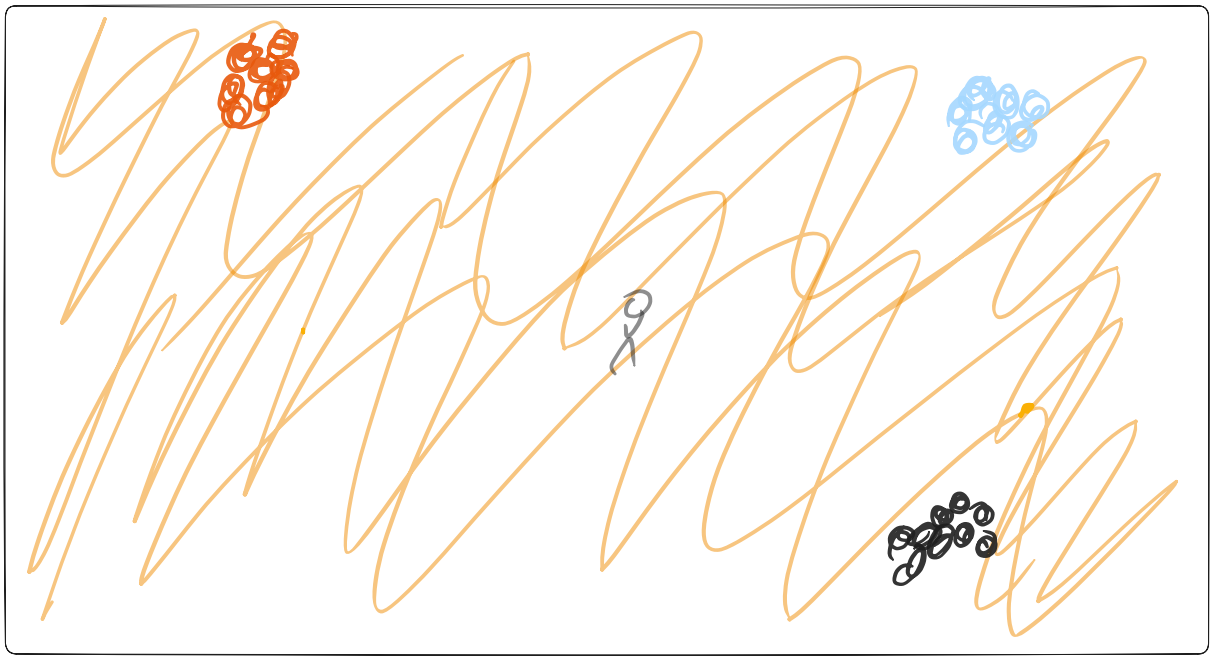

Abbildung 1: Spielstart

Beim Spielstart wird man in einer Wüste mit den drei Ressourcenarten Kohle, Kupfer und Eisen platziert. Zu dem Zeitpunkt kann man nicht mehr machen, als Maschinen zu platzieren, um die Ressourcen abzubauen und weiterzuverarbeiten.

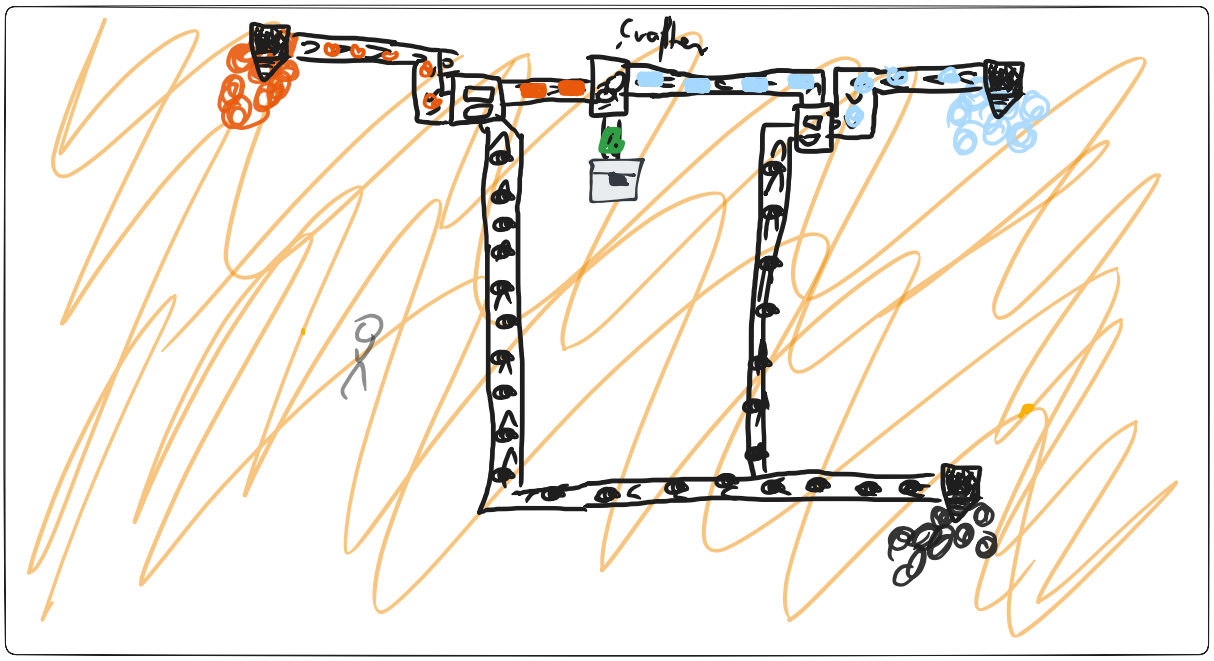

Abbildung 2: Spielmitte

Später im Spiel wird man schon einige Maschinen und Förderbänder gebaut haben, um Ressourcen abzubauen, weiterzuverarbeiten und schließlich in Kisten zu speichern.

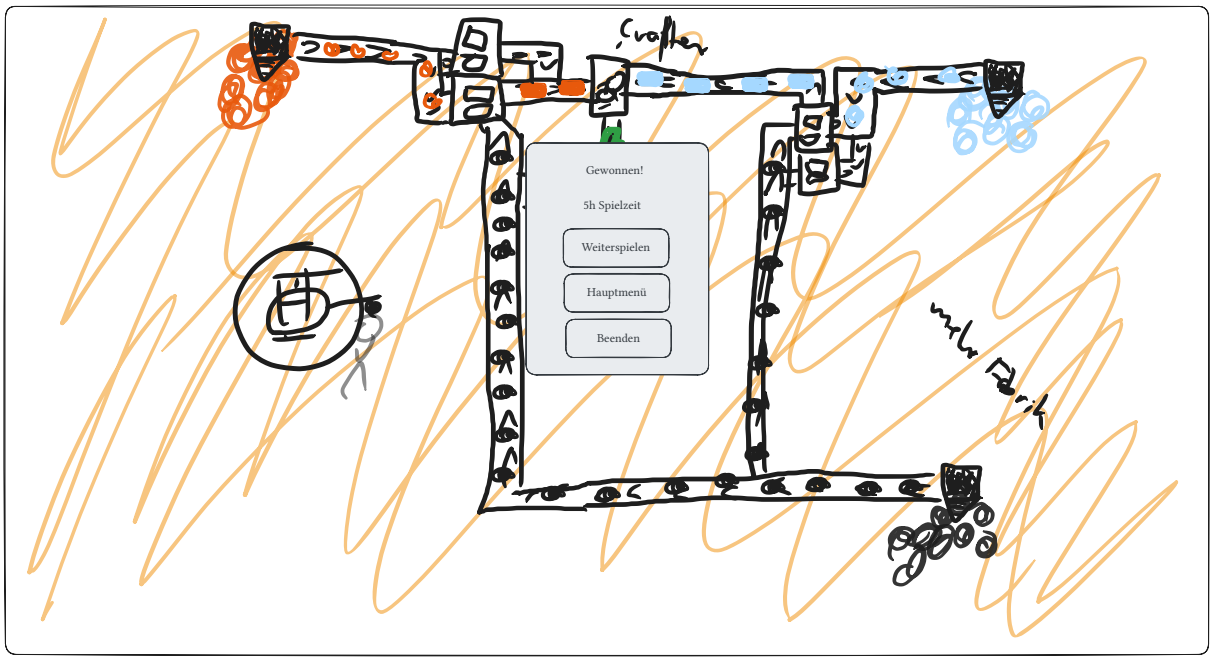

Abbildung 3: Spielende

Am Ende wird man eine noch größere Fabrik gebaut und das finale Item, vorläufig den Helikopter, hergestellt haben, um das Spiel zu beenden. Dazu wird einem noch die gesamte Spielzeit und das Menü angezeigt.

3. Versionierung

Die vorläufige Aufteilung der Aufgaben passiert in einwöchigen Sprints nach einer simplifizierten Variante des SCRUM—Prinzips.

Es gibt insgesamt acht Sprints:

Sprint	Datum
Sprint 1	22.04.-28.04.
Sprint 2	29.04.-05.05
Sprint 3	06.05.-12.05.
Sprint 4	13.05.-19.05.
Sprint 5	20.05.-26.05.
Sprint 6	27.05.-02.06
Sprint 7	03.06.-09.06.
Sprint 8	10.06.-15.06.

Tabelle 1: Sprints

Im Folgenden ist die vorläufige Aufgabenaufteilung (Product Backlog Items (PBIs), gröber noch aufgeteilt in Features, hier nicht erwähnt) aufgelistet:

Sprint	PBIs
Sprint 1	• Gamedesign • Basic Simulation
Sprint 2	• Building System • Art
Sprint 3	• Resource Patches • Miner Building
Sprint 4	• Crafting Recipes • Crafter Building
Sprint 5	• Splitter Building
Sprint 6	• UI • Save System
Sprint 7	• Doku fertigstellen
Sprint 8	• Puffer für Eventualitäten

Tabelle 2: Vorläufige Aufgabenaufteilung

3.1. Abweichungen zur Planung

Am Ende gab es einige Änderungen und es kamen während der Entwicklungen Ziele dazu und andere wurden doch nicht umgesetzt. Grund für neue Ziele war schlicht und einfach, dass am Anfang nicht alles bedacht wurde

Die wirkliche Aufteilung der Aufgaben sah dann so aus:

Sprint	PBIs
Sprint 1	• Gamedesign • Basic Simulation
Sprint 2	• Resource Patches • Miner Building
Sprint 3	• Belt rendering • Crafter Building
Sprint 4	• Combiner Building • Splitter Building Start
Sprint 5	• Splitter Building Bugfixing • Save System • Easier Building • Main Menu
Sprint 6	• Furnace Building
Sprint 7	• UI
Sprint 8	• Crafting Recipes

Tabelle 3: Finale Aufgabenverteilung

Das Building System wurde rausgenommen, weil es sehr komplex gewesen wäre und für einen MVP, beziehungsweise eine technologische Demo, nicht benötigt ist. Später ist das aber doch sehr wichtig für ein gutes Gameplay.

Der Punkt „Art“ (Texturen und ähnliches) wurde ganz entfernt und neue Texturen wurden immer erst hinzugefügt, wenn sie benötigt wurden. Das hat die Arbeit besser aufgeteilt und man wusste besser, was man genau brauchte, wenn man sich eh schon mit der Funktion gedanklich auseinandergesetzt hat. Außerdem war das immer auch ein guter Einstieg, ein neues Feature zu implementieren.

Dadurch, dass die beiden Punkte weggefallen sind, konnten der Miner und die Generierung von Ressourcen auch nach vorne gezogen werden, was beides definitiv wichtiger war. Damit hat sich die ganze Priorität auch erstmal dahin verschoben, eine funktionale Simulation mit den wichtigsten Maschinen hinzukriegen, anstatt als erstes „nice to have“ Funktionen zu coden. Diese hätten oft auch mehr Zeit in Anspruch genommen, als zur Verfügung stand.

Danach kam das nächste wichtige Gebäude: Der Crafter. Der Grund, warum Crafting Rezepte sich nach hinten verschoben haben, war, weil sie zum Testen und zum Schreiben der Dokumentation noch nicht gebraucht wurden. Erst für die finale Vorstellung sind diese benötigt.

Der Combiner war einer, der Punkte, die nicht von Anfang bedacht wurden, aber offensichtlich mit dem Splitter zu gruppieren waren. Der Vorteil war in dem Moment aber, dass die darunterliegenden Systeme, die für den Combiner benötigt wurden, schon mit dem Crafter implementiert waren. Dadurch hat der Combiner viel weniger Zeit in Anspruch genommen als anfangs vermutet.

Der Splitter hingegen hat viel mehr Zeit gebraucht, als erwartet wurde. Am Ende sind circa anderthalb Wochen für die Implementierung draufgegangen. Der Grund dafür war unter anderem fehlendes Verständnis für die Komplexität, aber vor allem hätte sich vorher nochmal theoretisch damit beschäftigt werden sollen. Dadurch, dass einfach das bestehende System Stück für Stück erweitert wurde, ohne einmal zu definieren, was man da jetzt genau macht, gab es sehr viele Probleme, die sich hätten schneller lösen lassen können.

Dafür konnten während dem Rest der Woche aber viele kleinere Funktionalitäten eingführt werden. Dazu gehörte das Hauptmenü, was innerhalb weniger Stunden fertiggestellt wurde, sowie das Speichersystem, was durch eine Bibliothek stark vereinfacht wurde.

Sprint 6 und 7 sind relativ leer, da die Zeit für die Dokumentation benötigt wurde und zu dem Zeitpunkt die wichtigen System schon da waren. Das UI wurde nur mit dem Furnace (der am Anfang gar nicht bedacht wurde) getauscht, weil in der zweiten Woche mehr Zeit zur Verfügung stand.

Die letzte Woche wurde dann nur noch dazu genutzt, alle Crafting Rezepte und Items hinzuzufügen, die das Spiel überhaupt erst spielbar machen, aber nicht für die Dokumentation benötigt waren.

Abschließend kann man sagen, dass die meisten Änderungen daher kamen, dass die Priorität falsch eingeschätzt wurde und dann sich dadurch alles andere auch verschoben hat.

4. Struktogramm Crafting System

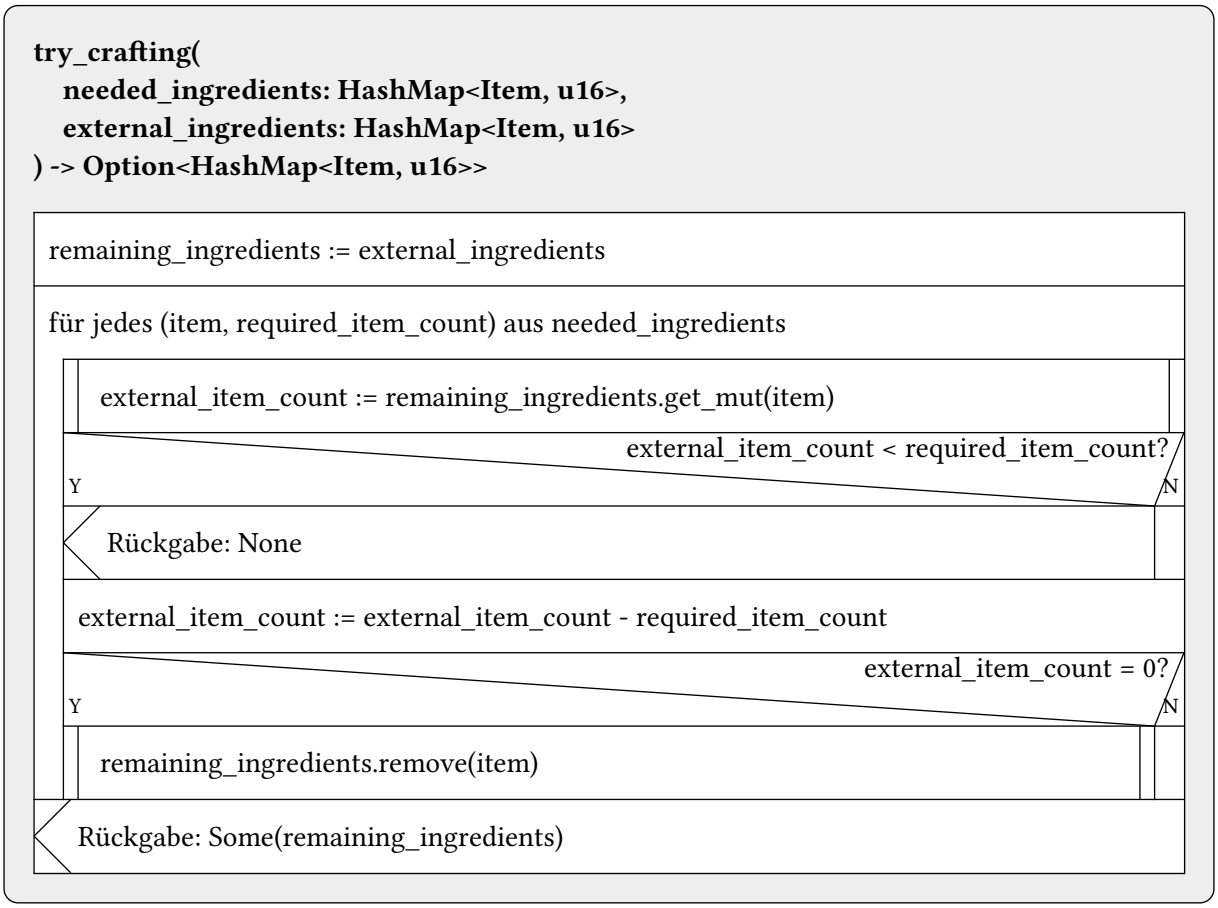

Abbildung 4: Struktogramm für Crafting-Funktion

Das Struktogramm zeigt die Methode, die verwendet wird, um zu testen, ob genügend Items für ein bestimmtes Crafting-Rezept vorhanden sind. Sie wird für jeden Crafter jedes Frame aufgerufen, ist aber kein direktes Performanceproblem, da Hashmaps sehr schnell sind. Aber sie wäre leicht zu optimieren, indem man sie zum Beispiel nur dann aufruft, wenn neue Items im Crafter sind.

5. Entwicklung

Die Entwicklung ist ohne größere Komplikationen verlaufen, auch wenn es manchmal sehr anstrengend war. Auch wenn das ganze Projekt sehr ambitioniert war, war von Anfang an nicht das Ziel, ein gutes Schulprojekt abzugeben, sondern viel dazu zu lernen und einfach Spaß zu haben. Doch hat das am Ende überhaupt funktioniert und ist das was, was man wiederholen sollte?

5.1. Technische Details

Als Game-Engine wurde sich für Bevy entschieden. Bevy ist eine noch relativ junge Game-Engine, in Rust geschrieben. Sie nutzt ein sogenanntes ECS:

- **Entities:** Representieren Objekte im Spiel und bestehen aus Components. Ein Beispiel wäre ein Gegner
- **Components:** Speichern Daten für ein Entity, zum Beispiel Leben oder Position
- **Systems:** Beinhalten die Logik, die anhand von Entities und Components arbeitet

5.1.1. Warum Bevy?

Bevy ist ja nicht gerade die bekannteste Game-Engine. Warum also nicht Unity oder Godot wie alle anderen?

Als allererstes ist Bevy frei und Open Source, dual-lizensiert unter MIT und Apache-2.0. Das heißt, man bezahlt nichts und keine Firma kann einem einfach so die Lizenz wegnehmen.

Unity war von Anfang an raus, zum einen wegen dem Skandal um die „Runtime Fee“, wegen fehlender Expertise und vor allem, weil sie einfach viel mehr Funktionen hat, als für dieses Projekt benötigt sind. Alles, was man über die Runtime Fee wissen muss, ist, dass Unity vor einigen Monaten eine Entscheidung getroffen hat, wodurch sie sehr viel Vertrauen von ihren Nutzern verloren haben und gezeigt haben, dass ihnen Geld wichtiger ist. Dazu kommt noch, dass Unity nicht Open Source ist.

Godot wäre eine andere Möglichkeit gewesen, doch auch da war das Problem, dass es einfach zu viele Funktionen gibt. Außerdem war das Vorwissen zu Godot begrenzt, was zu anstrengenderer Entwicklung und eventuell weniger Features geführt hätte.

Der Vorteil bei Bevy bestand darin, dass bereits umfangreiche Erfahrung mit Rust bestand und auch hin und wieder mit Bevy gearbeitet wurde. Des Weiteren bestand schon länger der Wunsch, Bevy richtig zu lernen und einmal ein ganzes Projekt darin fertigzustellen. Bevy ist relativ simpel, was den Funktionsumfang angeht, es hat (derzeit[2]) nicht einmal einen Editor. Es wird inklusive der Szenen alles im Code definiert und man hat volle Übersicht über sein Projekt.

Damit war die Entscheidung klar, Bevy zu verwenden, die Einfachheit, bestehende Expertise und dass die Engine Open Source war, hat alles dazu beigetragen. Man konnte recht früh schon sagen, dass das definitiv die richtige Entscheidung war, doch dazu später mehr.

5.1.2. Eventuelle Probleme mit Rust

Die einzige Angst war, dass die Sprache zum Verhängnis werden würde. Rust ist relativ komplex und der Compiler sehr restriktiv. Manchmal sitzt man unnötig lange an Problemen, die man mit anderen Programmiersprachen nicht hätte.

5.2. Entwicklungsablauf

Die Entwicklung des Spieles hat sich nicht immer ganz an die eigentlichen Aufgaben der Woche und das SCRUM-System gehalten, sich aber doch daran orientiert. Praktisch gesehen heißt das, dass zwar jede Woche eine bis drei Hauptaufgaben hatte, aber trotzdem zwischendurch immer mal an kleineren Fehlern oder Funktionen gearbeitet wurde. Für etwas größere Änderungen wurde dann aber ein Product Backlog Item mit Subtasks erstellt, um den Arbeitsaufwand besser einschätzen zu können.

Es gab einige Funktionalitäten, zu denen sich vorher schon viele Gedanken gemacht wurden, die teilweise auch schon in anderen Projekten implementiert worden sind und einfach übernommen werden konnten. Aber über all das, wozu noch kein konkreter Plan bestand, wurde jetzt nicht unbedingt viel nachgedacht, sondern man hat sich direkt an die Umsetzung gemacht.

[2] https://github.com/bevyengine/bevy/issues/85

5.2.1. Umsetzung neuer Funktionalitäten

Unabhängig davon, ob vorher ein Plan bestand oder nicht, war das erste immer, die Typen, Structs (vom Grundkonzept her ähnlich wie Klassen, aber ein bisschen anders, unwichtig im weiteren Verlauf) und Enums definiert.

Rust hat einen der striktesten Compiler von allen Programmiersprachen und das ist nicht anders bei der Typisierung. Mit dem strikten Typensystem und den gut ausgearbeiteten Generics und Traits (*A trait defines the functionality a particular type has and can share with other types*) kann man sich von Anfang an ein gutes Fundament legen, was es einem leichter macht, bei späteren Refactorings, Bugs zu erkennen. Dazu aber später mehr.

Der Vorteil, das so zu definieren, ist, dass man sich schon Gedanken über den Code gemacht hat und von dem Punkt aus gut weiterarbeiten kann. Außerdem erleichtert das spätere Refactorings und verringert die Wahrscheinlichkeit, bei sowas Bugs einzuführen, vor allem durch das strenge Typechecking.

Dann ging es an die praktische Umsetzung, oft viel durch Versuchen und es nicht schaffen. Das war dann der normale Entwicklungszyklus von erst was machen, was funktioniert, es dann richtig machen, die beste Lösung zu verwenden und als letztes wurde darauf geachtet, das ganze zu optimieren. Aber dazu muss man sagen, dass Optimierung hier kein wichtiger Punkt war.

5.2.2. Lösung von Problemen

Leider traten häufiger, als selten Fehler, Bugs und andere Probleme beim Testen auf. Im folgenden wird erklärt, wie damit umgegangen wurde und sie am Ende gelöst wurden.

Dafür wird sich hier beispielhaft der Bug angeguckt, wo man Probleme hatte, das vorherige Gebäude auszuwählen.

Das erste, was immer gemacht wurde, als ein Fehler auftrat, war eine neues Issue mit dem Label „Bug“ zu erstellen und grob zu beschreiben, was passiert ist, damit man sich später besser dran erinnern kann.

Sobald es dann an der Zeit war, den Bug zu bearbeiten, wurde erst einmal versucht, ihn nochmal zu reproduzieren. Das war zum einen, um sicherzustellen, dass er nicht vorher schon durch eine andere Änderung gefixt wurde, und zum anderen, um zu verstehen, was genau passieren muss, damit er auftritt.

5.2.2.1. Fehlercode analysieren

Wenn es einen Fehlercode gab, wurde dieser erstmal analysiert.

Bei dem Beispiel war die Fehlernachricht folgende:

```
index out of bounds: the len is 4 but the index is 7
```

Listing 1: Fehlernachricht

Daran kann man sehen, dass irgendwo ein Index falsch gesetzt wurde. Das Array ist eigentlich nur vier Elemente lang, es wurde aber versucht auf Index sieben zuzugreifen.

5.2.2.2. Hypothese aufstellen und überprufen

Dann wurde eine Hypothese aufgestellt, was der Fehler sein könnte und diese überprüft. In dem Fall war die Hypothese, dass zwei Variable vertauscht wurden, weil der Index sieben bei einem anderen Array in dem Kontext verwendet werden kann.

Die Überprüfung lief hier einfach darüber ab, dass im Code nachgeschaut wurde, ob sich das bestätigen lässt.

Auch wenn in dem Fall die erste Hypothese nicht gestimmt hat, war mit Blick auf den Code sofort klar woran es lag: Es wurde vergessen, eine Variable, die den Index für die derzeitige Maschinenvariante hält, zurückzusetzen. Das hat dann zu dem Problem führen, wenn die neu ausgewählte Maschine weniger Varianten hatte, als die vorheriger.

5.2.2.3. Bug fixen

Als letztes wurde noch eine mögliche Lösung implementiert und überprüft. Wenn es sicher war, dass dann alles funktioniert, wird die Änderung in Git commitet und der Bugreport geschlossen.

Und so lief das immer ab, Fehlercode analysieren, Hypothese aufstellen und überprüfen, Lösung implementieren.

5.3. Spielablauf an Methoden

Wie oben schon erwähnt, ist die Logik in Systeme aufgeteilt, anhand derer jetzt der Spielablauf erklärt wird.

Das erste, was der Spieler sieht, wenn er das Spiel startet ist ein kurzer Splash Screen mit dem Logo und dann das Hauptmenü. Beide werden in den jeweiligen `setup`-Systemen erstellt und auch wieder mit einem `cleanup`-System entfernt.

Abbildung 5: Hauptmenü

Das Spiel weiß durch den `GameState`, ob man jetzt auf dem Splash Screen, dem Hauptmenü oder im Spiel ist. Gewisse Systeme werden am Anfang oder Ende der jeweiligen Spielstatus ausgeführt, sowie während sie laufen.

Das nächste wichtige System wartet, bis der Nutzer einen der Buttons im Hauptmenü drückt. Sollte er „Quit“ drücken, wird das Spiel einfach direkt beendet, bei „Play“ wird der Spielstatus auf im Spiel gesetzt. Das führt dazu, dass das Hauptmenü entfernt und die Spielwelt aufgesetzt wird.

Es werden durch diverse Setup-Systeme alle benötigten Tilemaps und ähnliches geladen. Das interessante passiert dann in `load_game_save`. Dort wird das vorher gespeicherte Spiel für schnelleren Zugriff in den RAM geladen.

Das `generation`-System wiederum überprüft dann, ob ein Spiel geladen wurde und wenn ja, holt sich den Seed für die Spielwelt. Sollte kein Spiel geladen sein, wird ein neuer zufälliger Seed generiert. Davon ausgehend werden mit Simplex Noise die Ressourcen-Patches generiert.

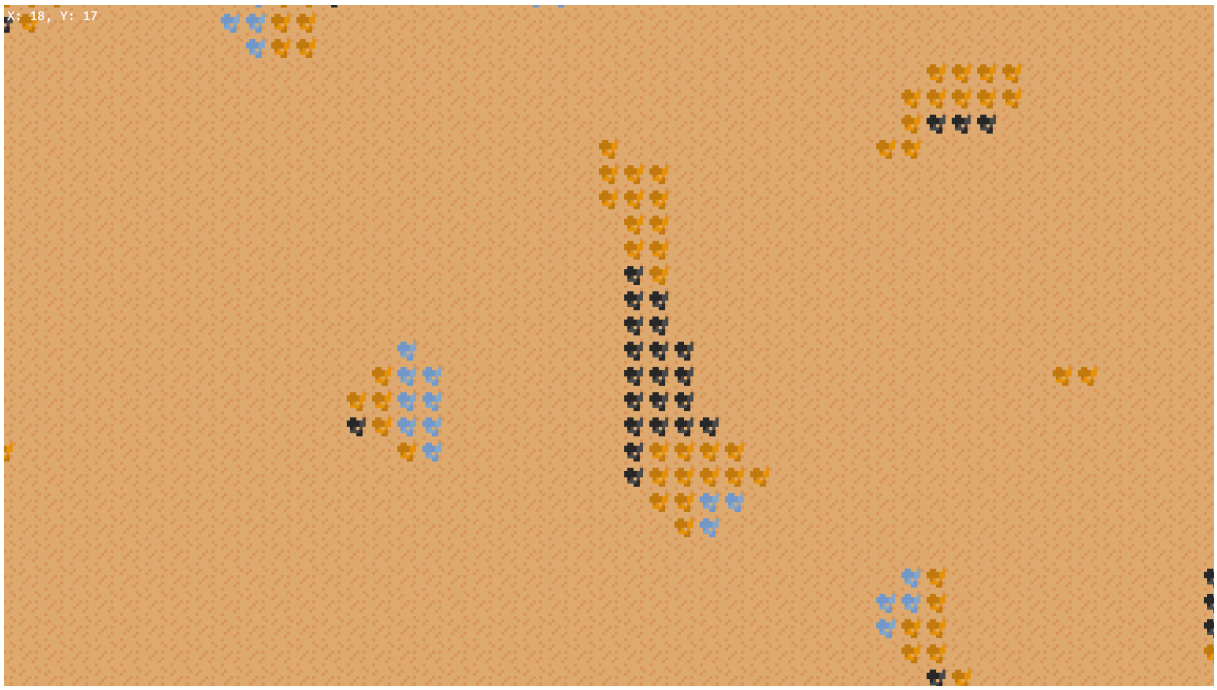

Abbildung 6: Neu generierte Welt

Jetzt kommen alle Systeme ins Spiel, die die eigentliche Logik kontrollieren. Angefangen mit der Kamerasteuerung hat man das System `movement`, das sich darum kümmert, die Kamera zu bewegen, wenn der Spieler die jeweiligen Tasten drückt.

Abbildung 7: Hover beim Bauen

Wenn der Spieler jetzt eine Maschine zum Platzieren auswählen will, gibt es `select_building`, was sich darum kümmert, das richtige Gebäude mit der richtige Variante als aktives zu markieren. `place_building` zeigt dieses daraufhin als Hover an und platziert es, sobald der Spieler linksklickt.

Die nächsten Systeme sind dann die, wo es interessant wird, weil die beinhalten die Logik zum Simulieren der Fabrik. Sollte der Spieler mittlerweile etwas gebaut haben, verwandelt `build_graph` das in eine Graph-Datenstruktur, die die Simulation einfacher und performanter macht, als wenn man direkt anhand der Tiles operieren würde.

Sobald der Graph konstruiert wurde, wird dieser mit `simulate` simuliert. Dieses System kümmert sich darum, dass alle Maschinen ausgeführt und ihre Items weitergegeben werden. Außerdem wird dort darauf geachtet, dass alles in der richtigen Reihenfolge abläuft und kein Chaos entsteht.

Nachdem alles simuliert wurde, muss der Graph anhand von `graph_to_world` wieder zurück in die Spielwelt übertragen werden.

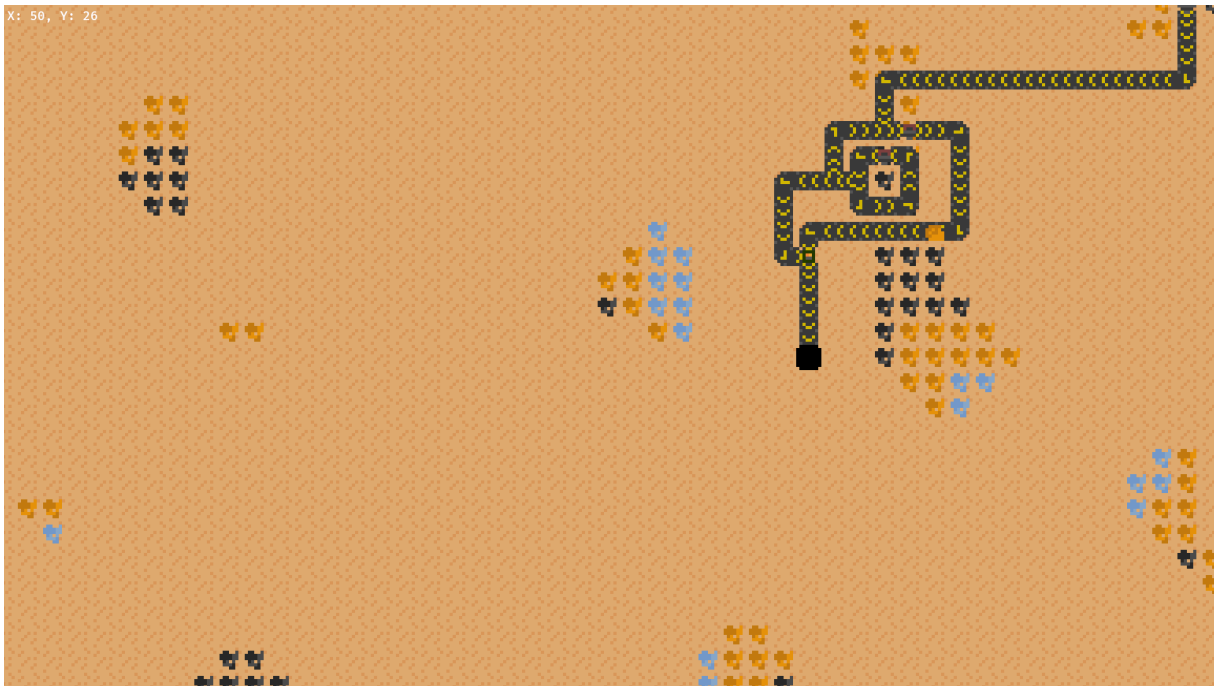

Abbildung 8: Nach ein bisschen Spielzeit

Wenn der Spieler dann fertig ist und das Spiel durch das Menü, beendet, wird die Methode `save_game` aufgerufen, die den Spielstand speichert.

5.4. Synchronisierter umgedrehter BFS

Das Herz des Spieles ist ein selbst entwickelter Algorithmus namens Synchronisierter umgedrehter BFS (oder Synchronized Reverse BFS im Englischen), der auf BFS basiert und dessen Aufgabe es ist, die Maschinen in der richtigen Reihenfolge zu simulieren.

5.4.1. Problemstellung

Jeden Tick (zehnmal pro Sekunde) muss die ganze Fabrik simuliert und weitergebracht werden. Doch das führt zu folgenden Schwierigkeiten:

Jede Maschine hat ein oder mehrere Ein- und Ausgänge, wodurch Items geschoben werden können. Außerdem haben alle Förderbänder eine Richtung.

Die Beispiele werden erstmal nur an Förderbänder gezeigt, weil das Prinzip gleichbleibt, aber so einfacher zu erklären ist.

Um die Fabrik zu simulieren, muss jedes volle Förderband überprüfen, ob das nächste Förderband frei ist und es dann sein derzeitiges Item dahinschieben. Man könnte jetzt einfach sagen, dass man sich jedes Förderband nimmt und einfach diese Überprüfung macht. Aber das hätte ein großes Problem: Man würde nichts in der richtigen Reihenfolge updaten. Es könnte sein, dass ein Förderband sein Item nicht weitergeben kann, weil das nächste nicht frei ist, obwohl es zu einem späteren Zeitpunkt im gleichen Simulationsschritt frei wird. Dadurch würden sich die Items sehr unregelmäßig und unvorhersehbar bewegen.

Das heißt, man braucht einen Algorithmus, der von unten startet und die Fabrik nach oben durchgeht, um die Items weiterzuschieben. Dadurch werden alle Items gleichmäßig weitergeschoben, ohne dass irgendetwas Unvorhergesehens passieren kann. Wenn die Items nacheinander von unten aus bewegt werden, dann macht jedes Item den Platz für das nächste frei.

Doch auch der Ansatz hat ein Problem, sobald sich Förderbänder in mehrere Richtungen aufspalten: Sollte von der einen Seite der Splitter erreicht worden sein, kann es aber sein, dass der sein Item auf der anderen Seite rausschiebt. Wenn dann danach die andere Seite wieder zum Splitter kommt, wird das schon ausgegebene Item noch ein Förderband weitergeschoben. Das führt dazu, dass das Item sich dann um zwei Felder bewegt hat.

5.4.2. Lösungsansatz

Die Idee, um das zu lösen, ist an jedem Knotenpunkt, wo sich Förderbänder aufteilen, zu warten, bis alle Pfade bis dahin simuliert wurden. Daher kommt „synchronisiert“ im Name des Algorithmus’.

Doch erstmal sollte überhaupt erklärt werden, wie die Spielwelt intern repräsentiert wird: Wenn der Spieler eine Maschine baut oder zerstört, wird das alles in einen Graphen umgewandelt. (Ein Graph ist eine Datenstruktur, bestehend aus sogenannten Nodes, die mit richtungsangebenden Pfeilen, verbunden sind.) In dem Fall ist jede Node eine Maschine und die Verbindungen zeigen an, wie die verschiedenen Maschinen verbunden sind. Außerdem wird auf jeder Verbindung gespeichert, in welcher Richtung die beiden Gebäude in der Spielwelt verbunden waren.

Das heißt, bei einem solchen Spielaufbau, würde folgender Graph generiert werden:

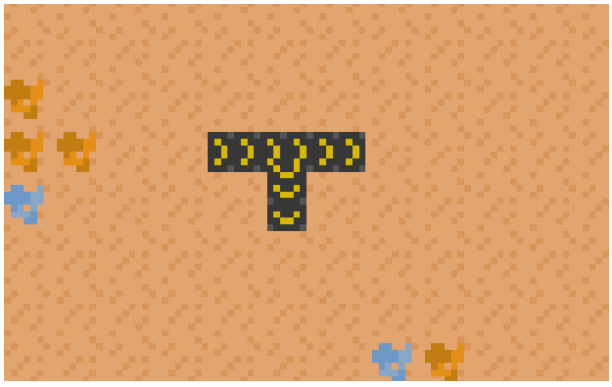

Abbildung 9: Förderband-Beispiel

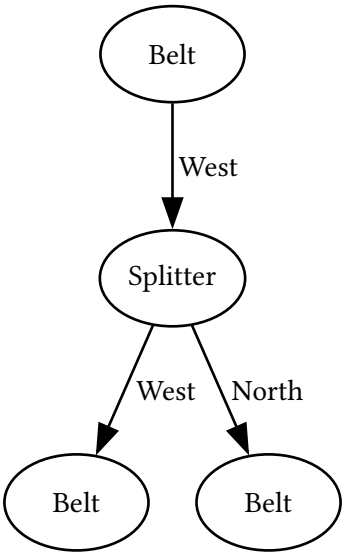

Abbildung 10: Graph zu Beispielaufbau

Also ist die Lösung, mit einem BFS (Breitensuche auf Deutsch, im Grunde genommen geht der Algorithmus einmal durch den ganzen Graphen anhand der Verbindungen durch) von unten anzufangen und jeden Pfad zu simulieren, bis ein Knotenpunkt erreicht wird. Daher kommt auch „umgedrehter

BFS“ im Namen, da man von unten anstatt von oben mit einem BFS den Graphen durchgeht. Dann wird überprüft, das wievielte Mal dieser Punkt jetzt erreicht wurde. Sollte das damit dann so oft passiert sein, wie die Maschine Ausgänge hat, wird diese simuliert und die Simulation wird fortgesetzt darüber fortgesetzt. Wenn das Gebäude noch nicht oft genung erreicht wurde, wird erstmal mit dem nächsten Pfad weitergemacht, bis es wieder erreicht wird.

5.4.3. Quellcode näher beschrieben

Siehe Absatz 7.1 für den ganzen Code mit Kommentaren.

Hier wird der Algorithmus einmal mit mehr technischem Detail beschrieben, gerade was den BFS und die Synchronisierung angeht.

Als allererstes müssen zwei wichtige Variablen deklariert und initialisiert werden: `visited` und `times_machines_hit`. `visited` ist ein Set von allen Nodes, die schonmal besucht und simuliert wurden, damit man keine zweimal versucht zu simulieren. Das könnte zum Beispiel passieren, wenn es einen Kreis in der Fabrik gibt. `times_machines_hit` ist eine Hashmap, auch bekannt als Hash Table oder dictionary. In dieser Hashmap wird gespeichert, wie oft eine Maschine bereits von einem Pfad getroffen wurde.

Danach werden alle stark zusammenhängenden Komponenten (SCCs) des Graphen gesucht. SCCs sind Subgruppen, wo alle Nodes alle anderen Nodes erreichen können. Das ist benötigt, wenn der Graph zyklisch ist, also an irgendeiner Stelle ein Pfad wieder zurück auf sich selbst verweist. Würde man einfach nur von den untersten Nodes aus die Simulation starten, würde man in manche Zyklen gar nicht reinkommen, um sie zu simulieren, weil zum Beispiel ein Splitter, der auf die folgende Weise platziert ist nie von beiden Pfaden gehittet werden würde (Siehe Abbildung 11). Damit der eine Pfad zu ihm kommen kann, müsste ja erstmal über dem Splitter weitergemacht werden, was aber nicht passieren wird, bis der Splitter zweimal erreicht worden ist. SCCs lösen das Problem, da sie den Zyklus in als eigene Komponente definieren. Die verwendete Funktion, um die SCCs zu finden, `tarjan_scc`, ist in der Abhängigkeit, die für Graphen verwendet wird, eingebaut. Diese Funktion findet nicht nur die SCCs anhand von Tarjan’s Algorithmus, sondern führt auch eine topologische Sortierung des Ergebnisses durch.

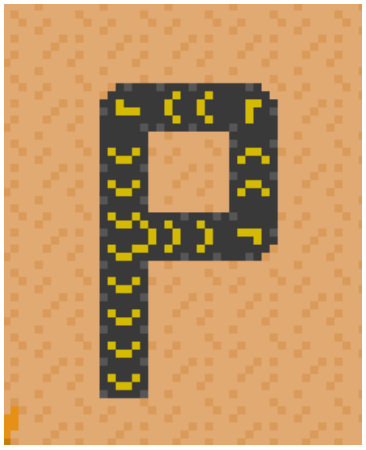

Abbildung 11: Ein Graph mit einem Zyklus

Eine topologische Sortierung, ist eine Sortierung von einer Liste von Nodes, wo am Ende die Nodes, die im Graph zuerst kommen, auch in der Liste zuerst erscheinen. Das garaniert in dem Fall, dass man trotzdem unten mit der Simulation anfängt und nicht irgendwo in der Mitte.

Als nächstes geht man durch jedes SCC und startet bei der jeweils ersten Node einen BFS. Dafür braucht man noch eine Queue, die speichert, welche Nodes man als nächsten simulieren muss. Dann

fängt das Programm an mit einem BFS durch den Graphen durchzugehen. Dazu wird immer der vorderste NodeIndex aus der Queue genommen. Sollte die Queue leer sein, wird mit dem nächsten SCC weitergemacht. Sonst wird überprüft, ob die Node schon in einem vorherigen Durchgang besucht worden ist. Wenn ja, wird mit der nächsten weiter gemacht und sonst werden alle Nodes von dem Graphen extrahiert, wo die derzeitige Maschine Items hinschieben kann.

Sollte das keine Ergebnisse liefern, heißt dass, das Gebäude ist das letzte in einer Reihe und kann einfach simuliert werden. Dann werden in dem Fall alle Maschinen, die in die derzeitige reinführen und noch nicht besucht wurden, der Queue der nächsten zu besuchenden Nodes angehängt. Als letztes wird die derzeitige Maschine als besucht markiert.

Wenn es aber ausgehende Verbindungen gibt, wird die Anzahl, die die derzeitige Maschine erreicht wurde um eins erhöht. Wie oben schon erwähnt, darf ein Gebäude erst simuliert werden, wenn alle ausgehenden Pfade simuliert wurden. Heißt, es wird der derzeitige Wert aus `times_machines_hit` abgerufen und mit der Anzahl der Ausgänge verglichen. Sollte die Anzahl der Ausgänge größer sein, wird mit dem nächsten Durchgang des BFS weitergemacht. Sonst werden auch hier alle Maschinen, die in die derzeitige reinführen und noch nicht besucht wurden, der Queue der nächsten zu besuchenden Nodes angehangen. Endlich wird an der Stelle auch die Maschine simuliert.

Als letztes wird jetzt versucht, für jeden Ausgang, ein Item, sofern vorhanden, zum nächsten Gebäude zu schieben. Dafür wird überprüft, ob es in die Richtung ein Item gibt, das weitergegeben werden kann. Dann wird noch das nächste Gebäude „gefragt“, ob es diese Art von Item überhaupt annehmen kann. Wenn das alles zutrifft, wird das Item weitergegeben und alles fängt wieder von vorne an.

6. Fazit

Im Rahmen dieses Projektes wurde ein Spiel entwickelt, wo der Spieler Fabriken baut, um verschieden Produkte herzustellen. Dafür musst ein eigener Algorithmus entwickelt und eine gute Aufgabenverteilung etabliert werden, um alles zu schaffen.

6.1. Reflexion

Das meiste, was geplant war, ist auch implementiert worden. Alles, was es nicht in die finale Version geschafft hat, ist auch nicht unbedingt benötigt für ein Schulprojekt.

Generell kann man sagen, dass das Projekt viel zu ambitioniert war, dafür dass es nur ein Schulprojekt war. Wäre das Ziel nur eine gute Note gewesen, hätte ein viel simpleres Spiel schon ausgereicht. Aber da kommt der Hauptgrund für dieses Projekt her: Es wurde nicht gemacht, um eine gute Note zu bekommen, sondern um etwas zu lernen und, vor allem, um Spaß zu haben. Und das ist beides genauso eingetreten.

6.1.1. Was ist gut gelaufen?

Sehr gut gelaufen ist die ganze Aufteilung der Aufgaben. Es hat sich definitiv gelohnt, sich am Anfang mal ein paar Stunden hinzusetzen und einfach nur alles, was zu machen ist, aufzuschreiben.

Außerdem war es überraschenderweise kein Problem, die Motivation zu halten. Bei früheren Projekten vergleichbarer Größe war nach sehr kurzer Zeit immer schon die Motivation und damit aller Grund weiterzuarbeiten, weg. Dabei hat definitiv auch der Druck der Schule geholfen, denn die Option, aufzugeben, gab es nicht.

Eine Befürchtung am Anfang, die sich nicht bestätigt hat, war, dass die Programmiersprache zum Verhängnis werden würde. Wie schon oben (Absatz 5.1.2) erwähnt, ist Rust sehr komplex und man braucht für manche Dinge viel mehr Zeit, als bei anderen Programmiersprachen. Auf der anderen Seite hat man aber zum Beispiel die Vorteile des strikten Type-Systems, das Refactorings viel einfacher gemacht hat.

6.1.2. Was sollte man anders machen?

Insgesamt kann man sagen, dass das Projekt generell sehr reibungslos verlaufen ist, gerade was die technische Implementation angeht.

Eine der Sachen, die gar nicht gemacht wurde, war automatisiertes Testen. Klar, das ist bei Spielen nochmal viel schwieriger, als bei anderen Programmen, aber der Vorteil von testgetriebener Entwicklung (erst Tests schreiben, dann den eigentlichen Code)[3] wäre gewesen, dass man sich vor der Implementation eines Features nochmal besser hätte Gedanken machen können, was jetzt genau das Ziel der Funktion ist. Hier war aber auch der Zeitdruck das Problem und dass nur das wichtigste gemacht wurde.

Damit wird auch direkt der letzte Punkt angesprochen: Das Projekt hat einem sehr viel abverlangt. Da das alles während der Klausurenphase stattfand, mussten die Prioritäten richtig gesetzt werden, um einer noch stärkeren Überarbeitung vorzubeugen. So viel Spaß die Arbeit auch gemacht hat, es wäre vielleicht doch besser gewesen, ein Projekt zu wählen, was für die Schule ausgereicht hätte, anstatt zu versuchen, eine so große Idee in so kurzer Zeit umzusetzen. Es gab Wochen, in denen man nach Hause kam und dann direkt mehrere Stunden lang bis in die Nacht hinein programmiert hat.

6.2. Blick in die Zukunft

Blickt man in die Zukunft, gibt es einige Pläne und teilweise schon konkrete Ideen, wie das Spiel erweitert und verbessert werden kann.

Da sind zum einen die Tunnelförderbänder, die eigentlich für gutes Gameplay unbedingt vonnöten sind. Das würde viel komplexere Fabriken erlauben, ohne die Schwierigkeiten, die man derzeit hat.

Ein weiteres System, was zwar nicht etwas ist, was für einen MVP benötigt ist, aber, wie vieles in diesem Projekt, eher eine Lernerfahrung wäre, ist eine Modding-API. Die internen Systeme sind von Anfang an so gebaut worden, dass Modunterstützung nicht schwierig wäre zu implementieren. Der Vorteil ist, dass man dadurch das Spiel leichter erweitern kann und auch intern schneller arbeiten würde. Die Idee ist, dass Mods zu Webassembly kompiliert würden, damit sie nicht für jede Platform einzeln zur Verfügung gestellt werden müssten.

Die nächste Frage ist, in welche Richtung das Spiel weitergehen soll: Soll es eher Richtung Survival-Spiel gehen, sodass man nur begrenzt Ressourcen hat und diese verwenden muss, um Maschinen herzustellen oder ob es eher um reine Effizienz geht. Das würde heißen, man hat unbegrenzt Ressourcen und Maschinen und müsste sich nur auf Effizienz und Logistik konzentrieren.

Schön wäre es, wenn die unterliegenden System so erweitern werden würden, dass man mehr mit den Maschinen machen kann. Das heißt, dass Maschinen möglich sein sollten, die über mehrere Tiles gehen und die Texturen variiert und animiert werden können.

In dem Zuge wären auch neue Texturen vonnöten, da die derzeitigen nur ihren Job erfüllen, aber nicht gut aussehen.

Das letzte große Feature, was in der nicht allzu fernen Zukunft dazukommen könnte, sind Flüssigkeiten. Das würde eine ganz neue Ebene an Schwierigkeit hinzufügen. Es könnte Öl geben, was man mit Raffinerien aufbereiten muss, um damit weiterarbeiten zu können. Außerdem wären Farben eine Möglichkeit, zur Personalisierung und um völlig neue Kombinationen von Items herzustellen.

Einige der Ideen sind einfacher zu implementieren als andere. Um da die Prioritäten weiterhin richtig zu setzen, sollte überlegt werden, was unbedingt benötigt wird, um das Spiel in einen spielbaren Status zu bringen und was noch warten kann.spielbaren Status zu bringen und was noch warten kann.

[3] https://en.wikipedia.org/wiki/Test-driven_development

7. Anhang

7.1. Quellcode Simulationsfunktion

```
/// Do a single simulation step of the world based on the `SimulationGraph`
pub fn simulate(
  mut simulation_graph: ResMut<SimulationGraph>,
  tile_query: Query<(&TilePos, &TileTextureIndex), With<Middleground>>,
  mut simulation_timer: ResMut<SimulationTimer>,
  time: Res<Time>,
) {
  // Check if this tick even is a simulation tick
  if !simulation_timer.tick(time.delta()).just_finished() {
    return;
  }

  // Return if the simulation graph is empty
  // aka there are no machines in the world
  if simulation_graph.node_count() == 0 {
    return;
  }

  // Get all the SCCs (Strongly Connected Components)
  // using Tarjan's algorithm.
  // This function also performs a topological sort on the result
  let scc = tarjan_scc(&**simulation_graph);

  let mut visited = HashSet::new();
  let mut times_machines_hit: HashMap<NodeIndex, u32> = HashMap::new();

  // Loop through all the first nodes of the SCCs
  for scc_start_node in scc.iter().map(|component| component[0]) {
    let mut next_nodes = VecDeque::from([scc_start_node]);

    // Run the BFS while there are nodes in the queue
    while let Some(node_index) = next_nodes.pop_front() {
      if visited.contains(&node_index) {
        continue;
      }

      // Get all the indices of the machines,
      // we could theoretically push to
      let next_machine_indices: Vec<(NodeIndex, Side)> = simulation_graph
        .edges_directed(node_index, Direction::Outgoing)
        .map(|next_machine_edge| (
          next_machine_edge.target(),
          *next_machine_edge.weight()
        ))
        .collect();

      // The value needs to be copied, because
```

```
// else the borrow checker would complain
let next_machine_indices_len = next_machine_indices.len();

// Check whether there are even any machines to try to push to
if !next_machine_indices.is_empty() {
  // Either increase the amount of times this
  // machine has been hit or insert one
  times_machines_hit
    .entry(node_index)
    .and_modify(|count| *count += 1)
    .or_insert(1);

  let times_machine_hit = times_machines_hit
    .get(&node_index)
    .expect("This was just inserted/updated, so it should exist");

  // If the machine hasn't been hit the amount of outputs it has,
  // continue, because we don't want to process it before it has
  // been hit by all its outputs
  if *times_machine_hit != next_machine_indices_len as u32 {
    continue;
  }

  // Insert all neighbors we want to visit into the queue
  for adjacent_node in
    simulation_graph.neighbors_directed(
      node_index,
      Direction::Incoming
    )
  {
    if !visited.contains(&adjacent_node) {
      next_nodes.push_back(adjacent_node);
    }
  }

  // Go through all machines the current machine could try to push to
  for (i, (next_machine_index, input_side)) in
    next_machine_indices.into_iter().enumerate()
  {
    // The output side of the connected machine is the opposite
    // of the current machine's input side
    let output_side = input_side.get_opposite();

    // Retrieve the nodes of the current and connected machine
    // This can't be done earlier, because of the borrow checker
    let ((machine, machine_tile_pos), (next_machine, _)) =
      simulation_graph.index_twice_mut(
        node_index,
        next_machine_index
      );
```

```
// Check whether this is the first time this loop is being run
// If this check wasn't made, the machine's action would be
// performed multiple times per frame. The check has to be done
// in the loop, because of borrow checker rules
// (`index_twice_mut` is the problem). But this is the same as if
// it was done right before the loop
if i == 0 {
  visited.insert(node_index);

  // Perform the machine's action
  machine
    .perform_action(get_middleground_object(
      &tile_query,
      machine_tile_pos
    ));
}

// Get the output items of the side being currently checked
let output_items = machine
  .output_items
  .get_side_mut(&output_side)
  .unwrap_or_else(|| {
    panic!(
      "The side {output_side:?} should exist on this machine"
    )
  });

// Get the frontmost output item, if it exists
let Some(item) = output_items.front() else {
  continue;
};

// If the connected machine can accept the item,
// push it to the correct input side
if next_machine.machine_type.can_accept(
  item,
  &next_machine.input_items,
  &next_machine.output_items,
  &input_side,
) {
    let item = output_items
      .pop_front()
      .expect("There should be an item in `output_items`");

    next_machine
      .input_items
      .get_side_mut(&input_side)
      .expect(
        "The input side should be set;
```

```
                  it's connected in the graph"
                )
                .push_back(item);
            }
          }
        } else {
          // ... because if not, all the additional steps for
          // trying to push items can be skipped

          // Insert all neighbors we want to visit into the queue
          for adjacent_node in
            simulation_graph.neighbors_directed(
              node_index,
              Direction::Incoming
            )
          {
            if !visited.contains(&adjacent_node) {
              next_nodes.push_back(adjacent_node);
            }
          }

          // Always mark this node as visited
          visited.insert(node_index);
          let (machine, machine_tile_pos) =
            &mut simulation_graph[node_index];

          // Perform the machine's action
          machine.perform_action(get_middleground_object(
            &tile_query,
            machine_tile_pos
          ));
        }
      }
    }
  }
}
```

Listing 2: Quellcode Simulation

Der gesamte Quellcode ist auch auf Github verfügbar:
https://github.com/TheBlckbird/sandy-factry

Die Revision bei Abgabe der Dokumentation ist:
https://github.com/TheBlckbird/sandy-factry/tree/eaf09d0

7.2. Wörteranzahl

Die Dokumentation hat **4796 Wörter**, ausgenommen sind Code und automatisch generierte Abschnitte, wie das Inhaltsverzeichnis, nicht aber der Anhang.

7.3. Quellen

- https://en.wikipedia.org/wiki/Breadth-first_search
- https://en.wikipedia.org/wiki/Strongly_connected_component
- https://en.wikipedia.org/wiki/Topological_sorting
- https://docs.rs/petgraph/latest/petgraph
- https://www.factorio.com

7.4. Andere Hilfsmittel

ChatGPT wurde hin und wieder verwendet, allerdings eher, um bestehenden Code zu verbessern (idiomatischeres Rust), als zur Ideenfindung oder für Logik. Ein Beispielprompt wäre folgendes:

Could this code be written better or shorter with Rust's declarative functions:
// code...

Die Male, wo versucht wurde, ein LLM für Code zu verwenden ist ganz oft nur Unsinn rausgekommen, unter anderem, weil Bevy sich sehr schnell verändert (Breaking Changes alle drei Monate).

Ansonsten wurde natürlich viel die Dokumentation von Bevy und petgraph, den zwei Hauptabhängigkeiten, verwendet.