Travail Personnel

Minecraft Forgotten Age Update

Beaume Yannick

6C6

2025

Inhalt

1.Einleitung 3
2. Die benutzte Software 3
A. IntelliJ 3
B. Block Bench 4
C. Fabric 5
3.Inhalt 5
A. DataGen 8
ModBlockTagProvider: 8
ModItemTagProvider: 9
ModLootTableProvider: 9
ModModelProvider: 10
ModRecipeProvider: 11
ModRegistryDataGenerator 12
B. Items 12
Custom Items 13
C. Blocks 14
Custom Blocks 14
D. Lang-Files 15
E. „Vorlagen“ 16
4. Schlussfolgerung 16
5. Lexikon 17
6. Quellen 19

1.Einleitung

Ich habe in diesem Jahr als Travail Personnel ein Mod/Update für Minecraft programmiert, um zu zeigen, wie schnell man Updates programmieren kann, da es mich stört, dass die Entwickler von Minecraft sehr lange für neue Erweiterungen brauchen.

Ich habe meine Mod mit Java geschrieben, da auch Minecraft in dieser Sprache geschrieben wurde. Als Mod-Loader habe ich mich für Fabric entschieden, da ich es früher einmal mit Forge probiert habe, jedoch ohne Erfolg. Mein Coding-Programm heißt IntelliJ. Ich habe auch noch Block Bench benutzt. Das ist ein für Minecraft-Development geeignetes 3D Design- und Animierprogramm. Ich werde alle genannten Aspekte im Laufe des Travail Personnel erklären.

2. Die benutzte Software

A. IntelliJ

IntelliJ IDEA ist eine integrierte Entwicklungsumgebung (IDE), die hauptsächlich für Java entwickelt wurde, aber auch viele andere Programmiersprachen unterstützt, wie Kotlin, Scala, Python und JavaScript. Sie bietet intelligente Code-Vervollständigung, effizientes Refactoring und leistungsstarke Debugging-Tools, die das Programmieren erleichtern.

Die Community Edition ist kostenlos und eignet sich für die meisten Entwicklungsbedürfnisse.

Ich benutze dieses Programm, um meinen Code zu schreiben.

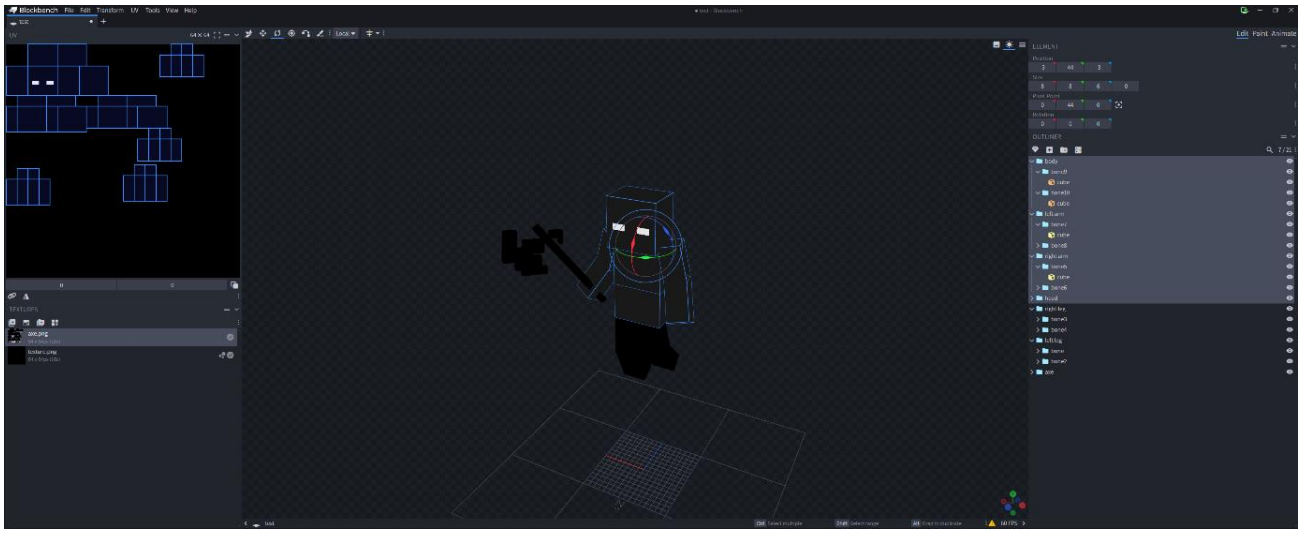
B. Block Bench

Blockbench ist ein kostenloses 3D-Modellierungsprogramm, das speziell für die Erstellung von Block-Modellen und Texturen für Minecraft und andere ähnliche Spiele entwickelt wurde. Es ermöglicht Nutzern, Modelle einfach zu erstellen, zu bearbeiten und in verschiedenen Formaten zu exportieren, die direkt in Spiele integriert werden können.

Die Benutzeroberfläche von Blockbench ist intuitiv und benutzerfreundlich, mit Tools, die speziell für das Modellieren von Blockgrafiken und Charakteren optimiert sind. Zu den Funktionen gehören die Erstellung und Bearbeitung von 3D-Modellen, die Texturierung und das Mapping von Oberflächen sowie die Animation von Modellen.

Blockbench unterstützt verschiedene Exportformate, darunter .json, .obj und .png, und bietet eine nahtlose Integration mit Minecraft, was es besonders für Modder und Entwickler von Minecraft-Ressourcenpackungen attraktiv macht. Das Programm ermöglicht es auch, Modelle für andere Spiele wie BlockCity oder Minecraft Bedrock Edition zu erstellen.

Dieses Programm benutze ich um meine 3D Modelle wie zum Beispiel Entitys, oder auch normale erweiterte Items hinzuzufügen



C. Fabric

Fabric selbst ist ein "Modloader", was bedeutet, dass es die Funktionalität bereitstellt, um Mods in Minecraft zu laden und auszuführen. Es ersetzt den Standard-Modloader von Minecraft, der für Mods verantwortlich ist, aber im Gegensatz zu anderen Modding-Plattformen wie Forge, ist Fabric dafür bekannt, besonders schnell zu sein.

3.Inhalt

In Minecraft gibt es eine Struktur namens ,,Ancient City“. Sie entsteht aus unterirdischen, zerfallenen Häusern. Ich persönlich mag die Struktur, weil es sehr viele interessante und unaufgeklärte Gebäude und Gegenstände gibt. Deshalb habe ich der ganzen Sache etwas Geschichte gegeben.

Ich habe deshalb ein neues Erz namens „Ancient Remains“ hinzugefügt. Es ist ausgesprochen selten, und soll die Überreste eines uralten legendären Schwertes darstellen.

```
public static final Block ANCIENT_REMAINS_BLOCK = registerBlock( name: "ancient_remains_block",  8 usages
        new ExperienceDroppingBlock(UniformIntProvider.create(6, 15),
                AbstractBlock.Settings.create()
                .strength(25f)
                .requiresTool()
                .sounds(BlockSoundGroup.ANCIENT_DEBRIS)
                .resistance(50f)));
```

Ich habe versucht das Erz so zu generieren, dass man es aus Zufall findet, das macht das Ganze noch spannender. Mit Hilfe des Erzes kann man sich neue Waffen schmieden, man benötigt dafür jedoch auch eine spezielle Schmiede und Brennstoff. Hier ist der Code der Schmiede:

```java
public class AncientForge extends Block {  6 usages

    public static final BooleanProperty LIT = BooleanProperty.of( name: "lit");  6 usages

    public AncientForge(Settings settings) {  1 usage
        super(settings);
        setDefaultState(this.getDefaultState().with(LIT,  value: true));
    }


    @Override
    protected ItemActionResult onUseWithItem
            (ItemStack stack, BlockState state, World world, BlockPos pos, PlayerEntity player, Hand hand, BlockHitResult hit) {
        if (stack.getItem() == ModItems.ASHES_OF_THE_OLD_DAYS) {
            world.setBlockState(pos, state.cycle(LIT));
            world.playSound(player, pos, SoundEvents.ITEM_FIRECHARGE_USE, SoundCategory.AMBIENT,  volume: 2.5f,  pitch: 1f);
            world.playSound(player, pos, SoundEvents.ENTITY_ENDER_DRAGON_AMBIENT, SoundCategory.AMBIENT,  volume: 0.6f,  pitch: 1f);
        }


        return super.onUseWithItem(stack, state, world, pos, player, hand, hit);
    }
    @Override
    protected void appendProperties(StateManager.Builder<Block, BlockState> builder) {
        builder.add(LIT);
    }

    @Override
    public void onSteppedOn(World world, BlockPos pos, BlockState state, Entity entity) {
        if (entity instanceof ItemEntity itemEntity) {
            if (itemEntity.getStack().getItem() == Items.IRON_INGOT) {
                if (!state.get(AncientForge.LIT)) {
                    world.playSound(entity, pos, SoundEvents.BLOCK_ANVIL_USE, SoundCategory.AMBIENT,  volume: 2f,  pitch: 1f);
                    itemEntity.setStack(new ItemStack(ModItems.STEAL, itemEntity.getStack().getCount()));
                }

            }

        }
    }
}
```

Dieser Code macht, dass der Block 2 verschiede Stadien hat, ,,lit“ und ,,unlit“. Er macht auch, dass der Block, wenn er platziert wird, von Natur aus auf ,,unlit“ geschaltet ist.

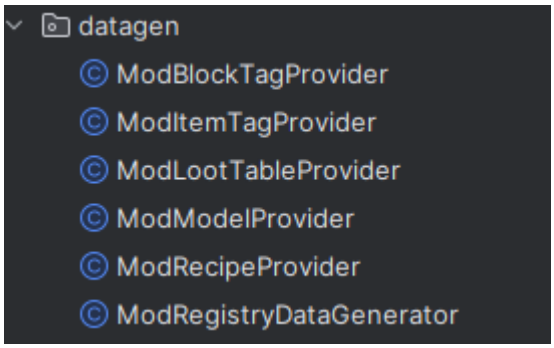
Dieser Code macht, dass, wenn man ihn mit einem bestimmten Gegenstand- in diesem Fall mit ,,Ashes of the old days“ anklickt, dass er dann einen Sound abspielt und seinen Status vom ,,lit“ auf ,,unlit“ oder umgekehrt wechselt.

Dieser Code macht, dass falls der Block den Status ,,lit“ hat, dass man dann im Stande ist, einen Eisenbarren auf ihn zu werfen und er ihn dann in einen Gegenstand namens ,,Steel“ umwandelt. Falls die Schmiede nicht angezündet ist („unlit“), passiert nichts, wenn man Eisen auf die Schmiede legt.

Ich habe auch noch einen anderen Code zu allen Blöcken geschrieben, die ich nicht gezeigt habe, denn in diesem Fall hätte ich nur einen Block, ohne Textur, das bedeutet man würde gar nicht erkennen, ob die Schmiede den Status „lit“ hat. Jedoch würde das Eisen sich trotzdem in „Steel“ umwandeln.

A. DataGen

Diese Klassen, die sich im „datagen“ Ordner befinden, sind dafür zuständig, die JSON-Files zu generieren. Eine Minecraft JSON-Datei ist einfach eine spezielle Textdatei, die Minecraft hilft, Dinge wie Gegenstände, Einstellungen oder bestimmte Informationen im Spiel zu speichern und zu organisieren, damit das Spiel richtig funktioniert.



ModBlockTagProvider:

Diese Klasse benutzte ich, um Gruppen mit den Gemeinsamkeiten der verschiedenen Blöcke zu erstellen. Zum Beispiel, anstatt in jedem Block zu erwähnen, dass man z.B. eine Spitzhacke aus Diamant- oder einem besserem Material benötigt, um ihn abzubauen, kann man einfach ein „Tag“ mit all den Blöcken erstellen.

```java
@Override
protected void configure(RegistryWrapper.WrapperLookup wrapperLookup) {

    getOrCreateTagBuilder(BlockTags.PICKAXE_MINEABLE)
            .add(ModBlocks.ANCIENT_REMAINS_BLOCK)
            .add(ModBlocks.BLOCK_OF_POWER)
            .add(ModBlocks.ASHES_OF_THE_OLD_DAYS)
            .add(ModBlocks.BEACON_OF_HOPE);



    getOrCreateTagBuilder(BlockTags.AXE_MINEABLE)
            .add(ModBlocks.OLD_LOG)
            .add(ModBlocks.OLD_PLANKS);



    getOrCreateTagBuilder(BlockTags.NEEDS_DIAMOND_TOOL)
            .add(ModBlocks.BLOCK_OF_POWER)
            .add(ModBlocks.ANCIENT_REMAINS_BLOCK)
            .add(ModBlocks.BEACON_OF_HOPE);



    getOrCreateTagBuilder(BlockTags.NEEDS_IRON_TOOL)
            .add(ModBlocks.ASHES_OF_THE_OLD_DAYS);
```

Bei diesem Beispiel benötigt man mindestens ein Diamantwerkzeug, um die Blöcke abzubauen.

ModItemTagProvider:

ModItemTagProvider ist exakt dasselbe, allerdings nur mit Gegenständen.

ModLootTableProvider:

ModLootTableProvider benutzt man, um bestimmten Blöcken einen Gegenstand zuzuteilen, der erscheint und eingesammelt werden kann, wenn man ihn abbaut (das sogenannte „droppen“). Falls man einen Block in einem Tag wie zum Beispiel „NEEDS_IRON_TOOL“ erwähnt hat, wird dieser Block nur seinen Gegenstand droppen, wenn diese Kondition erfüllt ist. In den meisten Fällen droppt der Block sich selbst und deshalb ist der Code sehr einfach und kurz.

```
addDrop(ModBlocks.ANCIENT_REMAINS_BLOCK, oreDrops(ModBlocks.ANCIENT_REMAINS_BLOCK, ModItems.ANCIENT_REMAINS));
addDrop(ModBlocks.ASHES_OF_THE_OLD_DAYS, oreDrops(ModBlocks.ASHES_OF_THE_OLD_DAYS, ModItems.ASHES_OF_THE_OLD_DAYS));
```

```
addDrop(ModBlocks.OLD_LOG);
addDrop(ModBlocks.OLD_PLANKS);
```

Diese sind die „Erz-Blöcke“. Diese Blöcke droppen nicht sich selbst, sondern ihr Erz-Gegenstand. Dadurch dass man „oreDrops“ im Code angibt, wird das Spiel, falls man den Block mit einem mit Behutsamkeit verzaubertem Gegenstand abbaut, (solange er dem entsprechendem Block-Tag angehört), den Block selbst droppen, und nicht den Erz-Gegenstand.

Dies sind einfach normale Blöcke, die sich selbst droppen.

ModModelProvider:

Dies ist etwas komplizierter. Das „Model“ eines Blocks oder Gegenstands, ist die Form des Gegenstands, und die Verknüpfung zu dem Texture-File. Dies ist auch die Stelle, wo ich angegeben habe, dass die Textur der Schmiede sich je nach dem Zustand ändert. Theoretisch könnte ich auch die Form je nach Status ändern. Dieser Code ist jedoch schwierig und lang, und deshalb werde ich ein anderes Beispiel nehmen.

```
blockStateModelGenerator.registerSimpleCubeAll(ModBlocks.ANCIENT_REMAINS_BLOCK);
```

registerSimpleCubeAll bedeutet einfach nur, dass jede Seite des Blocks gleich aussieht. Ich werde den Code nicht genau erklären, da es lange dauern würde, aber hier ist der Code der Schmiede. Die Schmiede sieht, wenn sie ausgeschaltet ist, von jeder Seite gleich aus und wenn man sie anzündet, nicht mehr.

```
Identifier forgeOffIdentifier = TexturedModel.CUBE_TOP.upload(ModBlocks.ANCIENT_FORGE, blockStateModelGenerator.modelCollector);
Identifier forgeOnIdentifier = blockStateModelGenerator.createSubModel(ModBlocks.ANCIENT_FORGE, suffix: "_on", Models.CUBE_ALL, TextureMap :: all);
blockStateModelGenerator.blockStateCollector.accept(VariantsBlockStateSupplier.create(ModBlocks.ANCIENT_FORGE)
        .coordinate(BlockStateModelGenerator.createBooleanModelMap(AncientForge.LIT, forgeOnIdentifier, forgeOffIdentifier)));
```

Wie ihr seht, hat die ausgeschaltete Schmiede von allen Seiten die gleiche Textur.

Wie ihr seht, hat die eingeschaltete Schmiede verschiedene Texturen auf verschiedenen Seiten.

ModRecipeProvider:

Erstellt die JSON-Files für die Crafting-Rezepte. Man kann verschiede Vorlagen benutzten, die oft im Spiel vorkommen (dies ist ein gutes Beispiel, um ein „Tag“ zu erklären), wie zum Beispiel *createDoorRecipe*(). Man kann jedoch auch *ShapedRecipeJsonBuilder* benutzten, um sein eigenes Rezept zu erstellen. Um das zu tun, stellt man ein 3x3 Crafting-Menu nach. Anschließend benutzt man irgendwelche beliebige Zeichen, um das Muster anzugeben. Diese Zeichen muss man dann entweder mit einem Vanilla- oder einem eigenem Item Verknüpfen.

```
ShapedRecipeJsonBuilder.create(RecipeCategory.MISC, ModItems.ANCIENT_SWORD)
        .pattern( patternStr: "cbc")
        .pattern( patternStr: "cac")
        .pattern( patternStr: "cdc")
        .input( c: 'a', Items.NETHER_STAR)
        .input( c: 'b', ModItems.STEEL)
        .input( c: 'c', ModItems.SHARD_OF_POWER)
        .input( c: 'd', ModItems.HANDLE)
        .criterion(hasItem(ModItems.SHARD_OF_POWER), conditionsFromItem(ModItems.SHARD_OF_POWER))
        .offerTo(exporter);
```

```
ShapelessRecipeJsonBuilder.create(RecipeCategory.MISC, ModBlocks.OLD_PLANKS,  count: 4)
         .input(ModBlocks.OLD_LOG)
         .criterion(hasItem(ModBlocks.OLD_LOG), conditionsFromItem(ModBlocks.OLD_LOG))
         .offerTo(exporter);
```

ModRegistryDataGenerator

Der DataGenerator ist dafür verantwortlich, dass Klassen und Features wie zum Beispiel Bäume, Terrain, oder andere Weltgenerationen generiert werden und in die Welt eingesetzt werden. Die ,,Configured Features“ ist dafür verantwortlich, wie die Dinge aussehen, und wie groß sie sind. Die ,,Placed Features“ sind dafür verantwortlich, wo die Dinge in der Welt platziert werden.

ModRegistryDataGenerator ist sehr kompliziert zu verstehen und zu erklären. Ich selbst verstehe den Sinn dieser Klasse auch nicht besonders gut. Ich weiß nur, dass sie da sein muss, deshalb mach ich mir keine Gedanken dazu. Man benutzt diese Klasse auch nicht oft, man muss einfach diese paar Codezeilen auswendig lernen.

B. Items

Um ein Item hinzuzufügen, muss man eigentlich nur ein paar Zeilen einfachen Code schreiben, jedoch wird dieses Item keine Textur, also kein Aussehen, keinen Namen, genauso wie irgendwelche Funktionen oder Möglichkeiten es im Spiel zu erhalten. Diesen Code schreibt man in die *ModItems* Klasse. Hier ist der Code, um ein einfaches Item ins Spiel zu programmieren.

```
public static final Item EXAMPLE_ITeM = registerItem( name: "example_item", new Item(new Item.Settings())
```

Um ein Item mit einer speziellen Funktion hinzuzufügen, muss man eine eigene Klasse für das Item erstellen. Rein theoretisch könnte man auch für normale Items eine eigene Klasse benutzten, das wäre jedoch ineffizient. Auch hier gibt es natürlich Vanilla-Vorlagen, wenn man zum Beispiel ein Schwert programmieren will. (Solange es ein ganz normales Schwert ist ohne irgendwelche Fähigkeiten). Hier zum Beispiel nehme ich den Code eins normalen Eisenschwertes und ändere den Angriffsschaden + Geschwindigkeit:

```
public static final Item EXAMPLE_SWORD = registerItem( name: "example_sword", new SwordItem(ToolMaterials.IRON, new Item.Settings()
        .attributeModifiers(SwordItem.createAttributeModifiers(ToolMaterials.IRON,  baseAttackDamage: 5,  attackSpeed: -2.4f))));
```

Custom Items

Nehmen wir an, ich möchte ein Schwert programmieren, das verwundete Ziele in Brand setzt. Hier müsste ich eine eigene Klasse erstellen. Da mein Schwert trotzdem noch nach dem Grundprinzip eines normalen Schwertes funktioniert, kann ich auch hier die Basic Vorlage für meine Klasse benutzen. Jedoch muss ich die Klasse dann auch noch erweitern, um mein Feuereffekt zu erhalten. Um dies zu tun, kann man mithilfe von *@Override* wieder eine Vorlage benutzen.

```
package net.cedric.tutorialmod.item.costum;


import net.cedric.tutorialmod.item.ModToolMaterials;
import net.minecraft.entity.LivingEntity;
import net.minecraft.item.HoeItem;
import net.minecraft.item.ItemStack;
import net.minecraft.item.PickaxeItem;
import net.minecraft.item.SwordItem;
import net.minecraft.util.Rarity;

public class AncientSword extends SwordItem {  2 usages  CedricD0812

    public AncientSword(Settings settings) {  1 usage  CedricD0812
        super(ModToolMaterials.ANCIENT, settings
                .attributeModifiers(SwordItem.createAttributeModifiers(ModToolMaterials.ANCIENT, (i
                .rarity(Rarity.EPIC)
                .fireproof());
    }

    @Override  3 usages  CedricD0812
    public boolean postHit(ItemStack stack, LivingEntity target, LivingEntity attacker) {

        if (target instanceof LivingEntity) {
            target.setOnFireFor(5);
        }
        return super.postHit(stack, target, attacker);
    }

    @Override  CedricD0812
    public boolean hasGlint(ItemStack stack) {
        return true;
    }
}
```

C. Blocks

Um einen Block hinzuzufügen, muss man eigentlich nur ein paar Zeilen einfachen Code schreiben, jedoch wird dieser Block keine Textur, also kein Aussehen, keine Form genauso wie irgendwelche Funktionen oder Möglichkeiten es im Spiel zu bekommen. Diesen Code schreibt man in die *ModBlocks* Klasse. Hier ist der Code, um einen einfachen Block ins Spiel zu programmieren:

```
public static final Block EXAMPLE_BLOCK = registerBlock( name: "example_block", new Block(AbstractBlock.Settings.create()));
```

Custom Blocks

Hier ist es ziemlich dasselbe wie mit den Items, man muss wieder eine eigene Klasse erstellen und wieder mit „*public static class* <name des blocks> *extends Block*“ anschreiben. Dadurch wird diese Klasse automatisch zu den Blöcken dazugezählt, und kann die gemeinsamen Methoden (Vorlagen) benutzen.

```
public class AncientForge extends Block {  6 usages  CedricD0812 *

    public static final BooleanProperty LIT = BooleanProperty.of( name: "lit");  6 usages

    public AncientForge(Settings settings) {  1 usage  CedricD0812
        super(settings);
        setDefaultState(this.getDefaultState().with(LIT,  value: true));
    }

    @Override  CedricD0812
    protected ItemActionResult onUseWithItem
            (ItemStack stack, BlockState state, World world, BlockPos pos, PlayerEntity player, Hand hand, BlockHitResult hit) {
        if (stack.getItem() == ModItems.ASHES_OF_THE_OLD_DAYS) {
            world.setBlockState(pos, state.cycle(LIT));
            world.playSound(player, pos, SoundEvents.ITEM_FIRECHARGE_USE, SoundCategory.AMBIENT,  volume: 2.5f,  pitch: 1f);
            world.playSound(player, pos, SoundEvents.ENTITY_ENDER_DRAGON_AMBIENT, SoundCategory.AMBIENT,  volume: 0.6f,  pitch: 1f);
        }
```

D. Lang-Files

Eine .lang-Datei ist einfach gesagt eine Übersetzungsdatei. Minecraft benutzt sie, um Texte anzuzeigen, z. B. den Namen eines Items, Blocks, Tools, einer GUI-Schaltfläche usw.

Statt den Text direkt in deinem Code zu schreiben, gibst du einen Schlüssel (Key) an, und Minecraft holt sich den passenden Text aus der .lang-Datei.

E. „Vorlagen“

Ich möchte unbedingt erwähnen was es mit diesen ,,Vorlagen“ auf sich hat, da es manchmal so klingt, als ob das Programmieren für Minecraft-Mods gar kein Können benötigt. Wenn ich von Vorlagen spreche, meine ich natürlich nicht wortwörtlich eine Vorlage. Im Projekt gibt es einen sehr großen Ordner mit dem Namen ,,external librarys“. Dies ist im Grunde ein Ordner mit dem gesamten Code von Minecraft. Ihr könnt euch die ,,Vorlage“ eigentlich wie ein *Tag* vorstellen. Zum Beispiel als Mojang das erste Schwert ins Spiel zugefügt hat, gab es bestimmt keine Vorlage, sondern eine spezielle Klasse. Doch als Mojang mit der Zeit immer mehr verschiedene Schwerter oder Gegenstände, die mit demselben Code funktionieren, hinzugefügt hat, haben sie einfach eine ,,Vorlage“ geschrieben, die die *Ultra Basics* in diesem Fall eines Schwertes enthielt. Es ist viel praktischer, weil sie jetzt einfach die ,,Vorlage“ nutzen können, jedes Mal, wenn sie ein Schwert hinzufügen wollen. Natürlich benutzte ich es auch. Und auch wenn es so klingt, als wenn man gar nichts selbst programmieren müsste, ist es trotzdem viel Arbeit. Zum Beispiel die Angriffsgeschwindigkeit, die Haltbarkeit oder den Angriffsschaden muss ich immer noch selbst hinzufügen.

4. Schlussfolgerung

Durch mein Projekt habe ich viel über die Funktionsweise von Minecraft-Mods gelernt und eigene Ideen erfolgreich- und nicht erfolgreich umgesetzt. Das Minecraft modden macht mir viel Spaß, ich habe jedoch noch keine feste Meinung entwickelt, ob ich mich in Zukunft diesem Thema als Hobby oder beruflich weiterhin widmen werde.

Warum hat es mich interessiert: Siehe Einleitung

5. Lexikon

Block: In Minecraft ist ein Block grundsätzlich ein 1x1 Meter Würfel.

Block-Modell: Ein Block-Modell bezieht sich auf das 3D-Modell eines Blocks im Spiel. Dies ist eine detaillierte Darstellung, wie ein Block aussieht und wie er im Spiel dargestellt wird. Modder können Block-Modelle erstellen oder anpassen, um den visuellen Stil des Spiels zu ändern.

Code: In der Minecraft-Modding-Szene bezieht sich "Code" auf den Quellcode, der das Spiel oder Mods steuert. Der Code ist die Programmiersprache, in der die Spielmechanik und die Funktionen programmiert werden. Minecrafts Java Edition basiert auf Java, was bedeutet, dass viele Mods ebenfalls in Java geschrieben sind. Modder schreiben oder bearbeiten Code, um neue Funktionen und Inhalte ins Spiel zu integrieren.

Droppen: das Erscheinen eines Objekts das eingesammelt werden kann

Entity: In Minecraft ist eine Entity ein allgemeiner Begriff für alles, was sich im Spiel bewegt oder unabhängig vom Spieler interagiert – also Dinge, die keine festen Blöcke sind.

Graphical User Interface (GUI): bedeutet auf Deutsch grafische Benutzeroberfläche. Das ist die Oberfläche, mit der du als Benutzer über grafische Elemente wie Fenster, Schaltflächen, Symbole, Menüs und Textfelder mit einem Programm oder Spiel interagierst.

Java/Bedrock Edition: Diese beiden Begriffe beziehen sich auf die verschiedenen Versionen von Minecraft:

Mod: Ein "Mod" ist eine Modifikation für Minecraft, die das Spiel verändert, indem neue Funktionen, Mechaniken, Blöcke, Kreaturen oder Inhalte hinzugefügt werden.

Mod-Loader: Ein Mod-Loader ist ein Tool, das es ermöglicht, Mods in Minecraft zu installieren und zu verwalten. Er sorgt dafür, dass Mods richtig geladen werden und miteinander kompatibel sind. Beispiele für Mod-Loader sind **Forge** und **Fabric**. Diese Tools erleichtern das Hinzufügen und Verwalten von Mods.

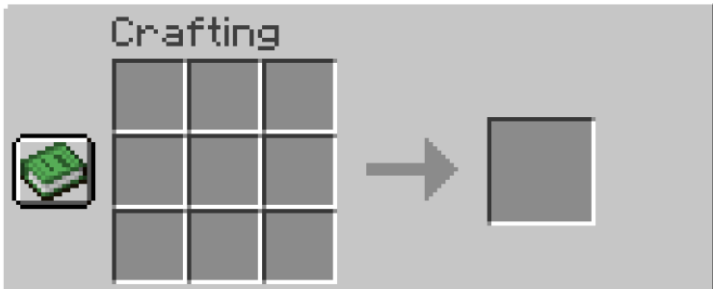
Refactoring: Es ist einfach erklärt ziemlich dasselbe wie ,,umbenennen“

- **Java Edition** ist die ursprüngliche Version von Minecraft, die für PC (Windows, MacOS, Linux) verfügbar ist. Sie unterstützt die meisten Mods und hat eine größere Modding-Community.
- **Bedrock Edition** ist eine Version von Minecraft, die auf vielen Plattformen läuft, darunter Konsolen (PlayStation, Xbox), mobile Geräte (iOS, Android) und Windows 10. Sie hat Einschränkungen bezüglich Modding und unterstützt oft keine Mods, die speziell für die Java Edition entwickelt wurden.

Texture-Mapping: Texture-Mapping ist der Prozess, bei dem die Oberflächen von Blöcken, Objekten oder Charakteren im Spiel visuell gestaltet werden. Texturen sind die 2D-Bilder, die auf 3D-Modelle angewendet werden, um ihnen ein realistisches oder stilisiertes Aussehen zu verleihen. Im Minecraft-Universum können Modder Texturen erstellen oder ändern, um das Aussehen von Blöcken, Items oder Kreaturen zu verändern.

Vanilla: Vanilla Minecraft ist einfach Minecraft, ohne Mods., zb.Vanilla-Gegenstände sind Gegenstände, die im normalen Minecraft existieren.

3x3 Crafting-Menu:



6. Quellen:

Dies ist der Youtuber, von dem ich den größten Teil des Programmierens gelernt habe.

https://www.youtube.com/channel/UCbzPhyLcO8VP25dZ7kaUyAw/videos

Ich habe außerdem auch noch eine Fabric-Dokumentation benutzt.

https://wiki.fabricmc.net/start

Natürlich habe ich auch ChatGPT benutzt, vor allem im schriftlichen Teil, um mir zu helfen, Code in einfachen Wörtern zusammen zu fassen.

https://chatgpt.com/