Republic Bread

Jan Nothacker

Hochschule Kempten
University of Applied Sciences

Projektdokumentation
Semester: WS 22/23

Entstanden im Rahmen
der Veranstaltung **GE-Lab**
an der Hochschule Kempten
bei Prof. Dr.-Ing. René Bühling

1. Überblick

1.1. Aufgabenverteilung

Während des Projekts haben wir versucht Hierarchien, für ein freundliches Arbeitsumfeld möglichst flach zu halten. Allerdings hat das nicht ganz funktioniert, u. a. auch, weil Ray und ich den Entwurf für das Spiel gemeinsam ausgearbeitet haben und damit auch bei allerhand Fragen Ansprechpartner*innen waren.

	Jan	Ray	Maximilian	Lukas	Alysha
Konzeption					
Idee & Konzept	✓	✓			
Artstyle		✓			
Concept Art		✓			✓
Leveldesign	✓			✓	
Dialog (Writing)	Vereinzelt		Vereinzelt		✓
Story	✓	✓			✓
Modellierung					
Referenz Zeichnungen		✓			
Polizeistation Eingangsraum				✓	
Detektivbüro				✓	
Seitengasse	✓				
Hauptstraße	✓				
Villa	✓				
Sprites		✓		✓	
Charaktermodelle	Brotagonist			Alle anderen	
Programmierung					
Charakter Bewegung			✓		
Inventarsystem			✓		
Gegenstands-interaktionen			✓		
Dialogsystem	(✓)[1]				
Beleuchtung	✓				
Dialog Sounds	✓				
Sound Implementierung	✓				
Menüs	✓				

[1] Beim Dialogsystem handelt es sich um das kostenlose Tool [Yarn Spinner], Jan hat sich um die Implementierung und Modifizierung nach unseren Bedürfnissen gekümmert.

2. Making-Of

2.1. Sounds für den Dialog

2.1.1. Idee

Dialog in Republic Bread wird durch sukzessives Aufdecken einzelner Buchstaben präsentiert. Um das ganze noch interessanter zu gestalten, haben wir uns überlegt während der Präsentation des Dialogs ein paar Sounds abzuspielen, jedoch ohne das gesamte Spiel mit gesprochenem Dialog zu vertonen. Deshalb haben wir uns auf ein Konzept Analog zu den Sounds aus der Spielereihe „Animal Crossing“ geeinigt.

Dabei waren uns folgende Punkte wichtig:

- Vorhersehbarkeit der Sounds
- Eigener Redestil pro Charakter
- Sounds einfach steuerbar machen
    - Für Entwickler und Anwender

Um Sounds vorhersehbar zu machen, muss bei gleichem Text die gleiche Ausgabe erfolgen. Die Idee war also für jeden der 26 Buchstaben aus dem Alphabet einen Sound aufzunehmen und jeweils den korrespondierenden Sound zum aktuell aufgedeckten Buchstaben abzuspielen.

2.1.2. Umsetzung

Die Sounds für die Buchstaben waren schnell aufgenommen. Um den Sounds noch einen etwas einzigartigeren Ton zu geben, habe ich die Geschwindigkeit verdoppelt.

Für das C# Skript, dass den korrespondierenden Sound für einen Buchstaben abspielt, muss man erst wissen, welcher Buchstabe gerade präsentiert wurde. Wir benutzen das kostenlose Tool „[Yarn Spinner]“ für den Dialog im Spiel. Das dort für die Präsentation zuständige C# Skript „LineView.cs“ bietet zwar ein Unity Event `onCharacterTyped()` an, jedoch wird dieses bei jedem aufgedeckten Buchstaben ausgelöst und gibt keinerlei Informationen darüber weiter, welcher Buchstabe gerade präsentiert wurde. Also musste eine modifizierte Version von „LineView.cs“ her. Die naheliegendste Lösung wäre Vererbung gewesen, allerdings beinhaltet das Skript auch eine weitere statische Klasse namens „Effects“. Diese Klasse ist unter anderem für die sukzessive Präsentation der einzelnen Buchstaben zuständig und musste deshalb ebenfalls modifiziert werden. Auch wenn es nicht sonderlich elegant ist, habe ich mich dann letzten Endes dazu entschlossen, das Skript „LineView.cs“ in ein neues Skript namens „LineViewSounds.cs“ zu kopieren und die Klasse „Effects“ in „EffectsAudio“ umzubenennen. Da [Yarn Spinner] unter der MIT Open Source Lizenz steht, ist das auch kein Problem. Das tatsächliche Abspielen der Sounds wird im Skript „CharSoundPlayer.cs“ abgehandelt, welches im LineViewSound Skript referenziert wird und den aktuell dargestellten Buchstaben, die Anzahl an aktuell sichtbaren Buchstaben und ob es sich um den letzen Buchstaben aus der Dialogzeile handelt, übergeben bekommt.

Der einzigartige Redestil pro Charakter wurde mit Scriptable Objects realisiert. Diese beinhalten den Namen des sprechenden Charakters, ob beim Abspielen des nächsten

Sounds der vorherige gestoppt werden soll, den Pitch und die Frequenz der Sounds, also nach wie vielen dargestellten Buchstaben ein Sound abgespielt werden soll *(Abbildung 1)*.

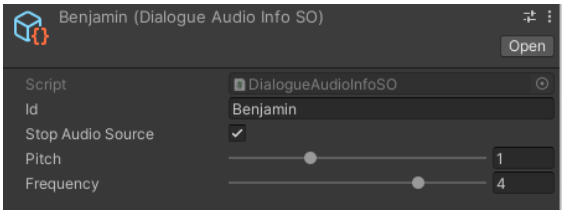

Abbildung 1: ein Scriptable Object mit den Audioinformationen

Das CharSoundPlayer Skript hat ein Array mit den Audio Info Scriptable Obects, welche dann im Skript in ein Dictionary mit der id und den Infos geladen werden *(Abbildung 2)*. Ändert sich der sprechende Charakter bzw. startet ein Dialog, ruft das LineViewSound Skript die Methode `SetCurrentAudioInfo(string _id)` aus CharSoundPlayer.cs auf, um die Audio Konfiguration für den sprechenden Charakter zu laden. Sollte sich nichts finden lassen wird die Default Info aktiviert. Abhängig von den Werten, die in der AudioInfo gesetzt wurden, werden dann die Sounds abgespielt. Die Frequenz ist durch eine einfach Modulo Operation umgesetzt, die Buchstaben 'ä', 'ö' und 'ü' werden durch ähnlich klingende Sounds substituiert. Handelt es sich beim übergebenen Buchstaben um den letzten Buchstaben aus einer Dialogzeile, wird für diesen auch ein Sound abgespielt. Durch Vergleichen des abzuspielenden Buchstaben und dem zuvor abgespielten Buchstaben wird auch vermieden, dass dieselben Sounds mehrmals in Folge abgespielt werden. Stattdessen wird entweder der zuletzt übergebene Buchstabe oder ein zufälliger Buchstabe abgespielt, abhängig davon, ob der zuletzt übergebene Buchstabe auch derselbe ist, wie der zuletzt abgespielte.

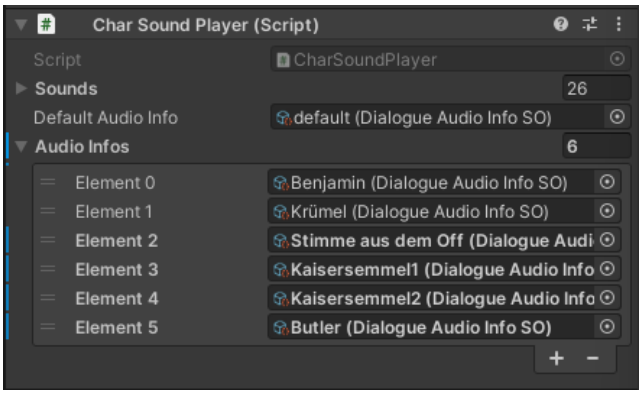

Abbildung 2: Das CharSoundPlayer Skript mit den dazugehörigen Sounds und Audioinformationen

Schlussendlich musste man den Sound für den Dialog noch irgendwie ein- und ausschalten können. Um diese Steuerung über .yarn Skripts, in denen der gesamte Dialog geschrieben wurde, zu ermöglichen, wurde zunächst die in [Yarn Spinner] mitgelieferte Methode `DialogueRunner.AddCommandHandler("Name", Method);` verwendet. Dadurch lassen sich Methoden aus C# über. yarn Skripts aufrufen. Über diesen Weg habe ich eine Methode `EnableSound()` und `DisableSound()` geschrieben, welche eine globale boolsche Variable ändern, die das Abspielen von Sounds steuert. Später fand ich jedoch heraus, dass dies noch einfacher ohne aufrufen von zusätzlichen Methoden geht. Daraufhin habe ich die Implementierung auf sogenannte „Tags“ umgestellt. Nach jeder Dialogzeile können beliebig viele Tags mit dem Präfix „#“ gesetzt werden *(Abbildung 10)* und diese werden dann pro Dialogzeile als Metadaten in Form eines String Arrays mitgeliefert und können ausgelesen werden. Mit einem einfachen `switch case` Konstrukt ließ sich das Ganze beinahe problemlos einbauen.

Den Anwendern ist auch die Möglichkeit gegeben diese Sounds zu umgehen, indem man während eine Dialogzeile präsentiert wird, einen Linksklick ausführt. So wird das

sukzessive Präsentieren der Buchstaben übersprungen und die gesamte Dialogzeile steht da, bis man erneut mit der linken Maustaste klickt.

2.1.3. Schwierigkeiten

Probleme gab beziehungsweise gibt es vor allem bei der Implementierung der Tags in das LineViewSound Skript, denn immer wieder gibt es „Geister-Tags“, also Tags an stellen im Dialog, an denen eigentlich gar keine Tags sein sollten. Im .yarn Skript steht an der korrespondieren Stelle kein Tag und doch, wenn man sich beim Abspielen des Dialogs die Tags über `Debug.Log("Tag :" + tag);` ausgeben lässt, muss man feststellen dass an der Stelle ein Geister-Tag seine Runden macht. Das Ganze ist erst sehr spät in der Entwicklung aufgefallen und es konnte auch keine Ursache gefunden werden. Die von mir programmierten Codezeilen sind simpel und bieten aus meiner Sicht keine Möglichkeit, dass versehentlich Tags aus anderen Zeilen irgendwo hin rutschen. Zumal dieses Problem nur vereinzelt auftritt.

Aktuell ist unser Workaround an den Betreffenden Stellen ein anderes Tag zu setzen. Das scheint die bösen Geister-Tags zu vertreiben.

2.1.4. Ergebnis

Am Ende haben wir eine Umsetzung, die alle unsere Anforderungen erfüllt und dabei auch noch gut klingt:

- Jeder Buchstabe hat einen korrespondieren Sound (Vorhersehbarkeit)
- AudioInfo Scriptable Object pro Charakter (eigener Redestil)
- Tags im Skript (einfach steuerbar).

2.2. Die Villa

2.2.1. Idee

Für eine Szene gegen Ende des Tutorials, in der der *Brotagonist* mit seinem *Brotiskop* über eine Mauer schaut, wurde eine Villa auf einem Hügel mit zwei Broten benötigt *(Abbildung 11)*.

2.2.2. Modellierung und Texturierung

Zunächst musste der Hügel her, dafür habe ich in Blender ein Grid erstellt und mit proportionalen Editing die Steigung nachmodelliert *(Abbildung 3)*.

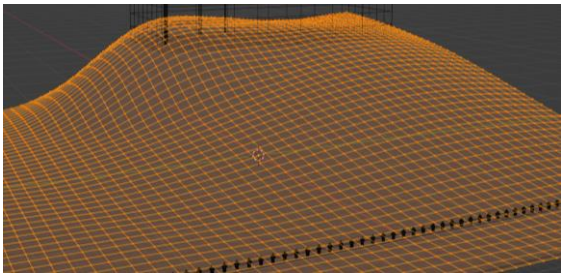

Abbildung 3: Das Grid, welches den Hügel bildet

Anschließend habe ich ein simples Haus mit Türen, Fenstern, einem Kamin und einer Regenrinne auf dem Hügel erstellt und am Fuße des Hügels eine einfache Mauer mit Stacheln platziert. Anschließend haben Dach, Fassade, Tür, Tür- & Fensterrahmen, Mauer und Stacheln einfache Texturen bekommen, welche mit [Stable Diffusion] und [Dream Textures] generiert wurden *(Abbildung 4)*.

Abbildung 4: Die Villa Szene mit Texturen

2.2.3. Importieren zu Unity

Nachdem alles fertig modelliert war, habe ich die Szene zu Unity importiert. Da das Level am Abend spielen sollte, wurde eine Abendliche Palette mit vielen Lila tönen für die Skybox gewählt und mithilfe des [Quibly] Skybox Shaders für die Szene umgesetzt *(Abbildung 5)*.

Abbildung 5: Die Skybox, beziehungsweise der dazugehörige Farbverlauf

Anschließend habe ich bei allen Objekten die Materialien für ein einheitliches Aussehen auf den [Quibly] Stylized Lit Shader umgestellt und die Einstellungen angepasst, so dass sie zur Szene passen. Dem Boden wurde auch mit Hilfe von [Stable Diffusion] eine Gras Textur gegeben, welche dann mit der Vertex Paint Funktion aus Polybrush[2] noch mit etwas mehr grün und dem braunen Trampelpfad versehen wurde *(Abbildung 6)*.

Abbildung 6: Die Villa Szene mit den Finalen Shadern und dem texturierten Hügel

2.2.4. Vegetation und Dekoration

Anschließend wurden die Wolken und die Rauchsäule aus dem Kamin mit dem [Quibly] Foliage Generator & Cloud3D Shader erstellt und platziert. Nun fehlte nur noch etwas an Natur. Dafür wurde mithilfe des [Quibly] Grass Generator und Grass Shader ein paar Gras Prefabs erstellt, die Büsche und Blumen Prefabs aus dem [Toon] Fantasy Nature Assets in die Scatter Brush von Polybrush geladen und sinnvoll platziert *(Abbildung 7)*. So hat es auf dem Trampelpfad z. B. keine Pflanzen, dafür wurden die Büsche gezielt um den Trampelpfad herum platziert. Abschließend wurden noch ein paar Bäume von Hand platziert, ein Bordstein und der Gehweg vor die Mauer gesetzt und der Sprite der sprechenden Brote in der Szene platziert und schon war alles Sichtbare fertig!

Abbildung 7: Die Villa Szene mit Wolken und Vegetation

2.2.5. Einbindung in die Unity Szene

Ein Großteil des Tutorial Levels befindet sich in einer Unity Szene namens „Main Scene“. Dazu gehört auch die Villa. Um nun den Anwendern zu ermöglichen mit dem Brotagonisten zwischen den verschiedenen Orten innerhalb der Unity Szene zu reisen, hat Maximilian ein Skript namens „SceneDoor.cs“ geschrieben. Mit diesem muss man einem GameObject nur den Unity Tag „SceneDoor“ zuordnen, das

[2] Polybrush ist ein von Unity zur Verfügung gestelltes Package

SceneDoor Skript als Komponente, einen Collider und ein Empty als Child hinzufügen. Das Empty dient als Spawnpunkt des Brotagonisten nach traversieren der "Tür".

In diesem Fall haben wir uns einfach dazu entschlossen ein Auto als "Tür" zu verwenden. Allerdings sollten die Anwender erst in der Lage sein, diesen Abschnitt im Spiel zu erreichen, wenn mit einem NPC eine bestimmte Stufe im Dialog erreicht wurde. Wenn diese Stelle im Dialog erreicht wurde, wird eine Variable im Dialogskript geändert. In der Umsetzung habe ich eine bei [Yarn Spinner] mitgelieferte Methode genutzt, mit der sich Variablen aus .yarn Skripts in C# Skripts laden lassen. Anschließend habe ich das SceneDoor Skript auf eine Wiese erweitert, mit der anhand einer Dialog Variablen die “Tür“ gesperrt werden kann und stattdessen ein Dialog abgespielt wird *(Abbildung 8)*.

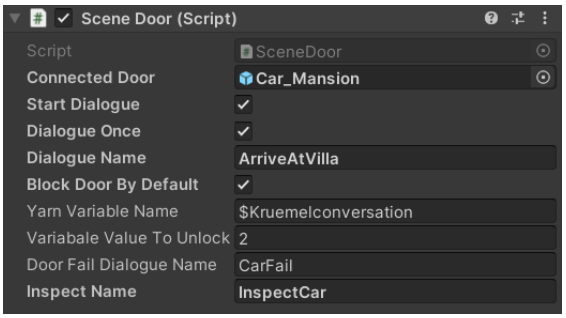

Abbildung 8: Das SceneDoor Skript mit Eingaben zum Blockieren von Türen

Damit der Brotagonist auch mit dem Brotiskop über die Mauer schauen kann, musste ein Weg her, mit dem wir den Brotagonist an eine bestimmte Stelle laufen lassen können. Dafür habe ich eine Methode geschrieben, die der NavMeshAgent Komponente ein empty GameObject in der Szene als neues Ziel setzt und den Dialog pausiert, bis der Brotagonist an der Position angekommen ist *(Abbildung 12)*. Durch Hinzufügen des `[YarnCommand("CommandName")]` Attributs konnte ich die Methode im .yarn Skript mit Parametern aufrufen, den Brotagonisten an die gewünschte Stelle laufen lassen und die aktive Kamera mit einer weiteren simplen Methode ändern *(Abbildung 13)*. Um den Anwendern das Gefühl zu geben, wirklich durch ein Brotiskop (Periskop) zu schauen, habe ich der betreffenden Kamera mit lokalen Postprocessing Effekten einen Fisheye- und Vignette-Effekt gegeben *(Abbildung 9)*.

Abbildung 9: Die Sprechenden Brote, wie man sie durch das Brotiskop sieht

In dieser Kameraeinstellung wird ein Dialog abgespielt, an dessen Ende die Anwender das Tutorial verlasen und in einer anderen (Unity) Szene landen. Und damit war die Villa Szene auch komplett fertig ausgearbeitet!

3. Best-Of-Bilder

```
183   title: BrotiskopCrafted
184   ---
185   <<set $HasBrotiskop to true>>
186   <<set $PickedUpBaguette to false>>
187   Benjamin: Hey, damit könnte ich vielleicht um die Ecke schauen. #enablesound
188   Benjamin: Damit habe ich quasi ein Brotiskop.
189   Benjamin: ... #tripledot
190   Benjamin: Oder vielleicht doch Perisbrot? #afterdot #audiooneline
191   ===
```

Abbildung 10: Eine Dialognode mit Variablen und Tags

Abbildung 11: Die Villa Szene, wie sie im finalen Spiel ist

```
[YarnCommand("WalkMarkerAndWait")]
0 Verweise
public IEnumerator WalkMarkerAndWait(GameObject destination)
{
    agent.destination = destination.transform.position;
    yield return WaitForNavMeshAgentToReachTarget(agent);
}
```

Abbildung 12: Eine Methode mit dem Yarn Command Attribut

```
title: TalkingBois
---
Benjamin: Da kommt mir mein Perisbrot ja wie gerufen. Soo wer bist du nun? #enableaudio
<<WalkMarkerAndWait brotagonist ZaunMarker>>
<<Spy>>
Butler: Die Bewohner sind inzwischen ziemlich aufgebracht Sir.
```

Abbildung 13: Die Methode aus Abb. 11 als Yarn Command

4. Probleme

4.1. Sperren von Interaktionen, während Dialog läuft oder das Inventar offen ist

Während der Entwicklung war schnell klar, dass eine Methode her muss, mit der man Interaktionen mit dem Spiel sperren kann, während die Anwender in einem Dialog oder im Inventar sind. Denn das Fortsetzen des Dialogs, die Interaktion mit Items im Inventar und die Steuerung des Brotagonisten finden alle über die linke Maustaste statt.

Also habe ich im Skript für die Steuerung des Brotagonisten eine private Membervariable namens „`m_bisActive`“ deklariert, welche steuern kann ob Interaktionen in der Szene gesperrt sind. Wenn nun ein Dialog startet oder das Inventar geöffnet wird, wird die Variable über eine Methode auf *`false`* gesetzt und umgekehrt.

Diese Implementierung kam jedoch mit zwei Problemen:

1. Sollte ein Dialog in einer `Awake()` Methode gestartet werden, wird die Interaktion nicht gesperrt.
2. Wenn ein Dialog läuft und man währenddessen das Inventar schließt, wurde die Interaktion wieder freigegeben

Die Lösung für diese Probleme war letzten Endes dem Inventar eine Membervariable für den offen/geschlossen Status „`m_isOpen`“ zu geben und auch die Variable „`IsDialogueRunning`“ aus dem Dialogue Runner auszulesen und beide mit einer AND-Operation zum Sperren von Interaktionen verwenden.

4.2. Skalierung

Ein weiteres Problem war, dass beim ursprünglichen Erstellen der Hauptszene der Brotagonist als erstes „hineingestellt“ wurde und danach alle anderen Modelle anhand des Brotagonisten hoch skaliert wurden. Das hatte zur Folge, dass UI-Elemente im Editor viel zu klein und damit kaum sichtbar waren *(Abbildung 15)*. So ging im Vergleich zu Szenen in „normaler“ Skalierung die Übersicht ein wenig verloren *(Abbildung 16)*. Darüber hinaus musste deshalb bei allen Einstellungen, bei denen es um Entfernungen geht, mit absurd hohen Werten umgegangen werden, wodurch jegliche Granularität verloren gegangen ist *(Abbildung 14)*.

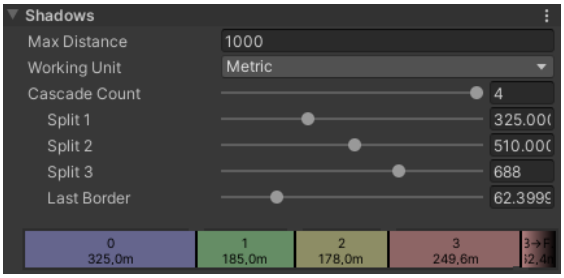

Abbildung 14: Extrem hohe Werte bei den Schatten im URP Asset

5. Problem-Bilder

Abbildung 15: Eine kaum sichtbare Lichtquelle aufgrund der groß geratenen Objekte

Abbildung 16: Gut sichtbare UI-Elemente dank "richtiger" Skalierung

6. Quellenverzeichnis

Stand der Weblinks: 23.01.2023

6.1. Assets und Inhalte

- Packages:
  - [Yarn Spinner] Secret Lab, „Yarn Spinner“, MIT Lizenz, https://yarnspinner.dev/
  - [Quibly] Dustyroom, „Quibly: Anime Shaders and Tools“, https://assetstore.unity.com/packages/vfx/quibli-anime-shaders-and-tools-203178
  - [Toon] SICS Games, „Toon Fantasy Nature“, https://assetstore.unity.com/packages/3d/environments/landscapes/toon-fantasy-nature-215197
- Schriftarten:
  - Dieter Steffmann „Marker Felt“, https://www.1001freefonts.com/marker-felt.font
  - Daniel Midgley, „Daniel“, https://www.1001freefonts.com/daniel.font
- Texturen:
  - Alle von mir verwendeten Texturen, außer der Brotagonist (erstellt von Hand), wurden mithilfe von Stable Diffusion erstellt (Modelle v1.5, v2.1-base und 2-depth).
  - Verwendete Tools:
    - [Stable Diffusion] n00mkrad, „NMKD Stable Diffusion GUI – AI Image Generator“, https://nmkd.itch.io/t2i-gui
    - [Dream Textures] Carson Katri, „Dream Textures – Stable Diffusion built into Blender“, https://github.com/carson-katri/dream-textures