FACHHOCHSCHULE ERFURT UNIVERSITY OF APPLIED SCIENCES

Projektdokumentation

Blacksmith Saga-Simulation

ITIG WS 22/23

Projektteilnehmer:	Nikita Studenikin Justin Noske Adrian Hettstedt
Modul:	Interaktive Technologien / Interaktionsgestaltung
Fachbetreuer:	Prof. Rolf Kruse / Jonathan Müller

26.02.2023

1. Installationshinweise & Systemanforderungen

1.1 Systemanforderungen

	Minimum	Empfohlen
Prozessor	Intel Core i3 2100	Intel Core i3 8300
Arbeitsspeicher	2GB	4GB
Festplatte	250 MB freier Speicherplatz	500 MB freier Speicherplatz
Grafikkarte	Nvidia GT 210	Nvidia RTX 1650

Das Spiel ist für die Full HD Anzeige optimiert worden.

1.2 Installation

Nachdem die Archivdatei entpackt wurde, kann der Spieler die beigefügte Datei “Blacksmitch Saga.exe” öffnen. Anschließend befindet sich der Spieler im Hauptmenü und kann das Spiel starten.

1.3 Spielregeln

Der Spieler hat im Menü die Möglichkeit, das Spiel über 2 Wege zu starten. Er kann sein bereits gespieltes Spiel starten oder ein neues Spiel starten. Im Settings Menü kann sich der Spieler zuvor mit allen wichtigen Tastenkombinationen vertraut machen.

2. Spielkonzept

2.1 Spielidee

Blacksmith Saga beruht auf der Idee, eine Management-Schmiede-Simulation zu erschaffen. Die Top-Down Ansicht sowie der klassische Singleplayer Modus erinnern an klassische Spiele dieses Genres wie Graveyard Kepper oder Legend of Zelda.
Das Spiel soll dem Spieler einen schnellen und direkten Einblick in die Spielwelt geben. Deshalb ist die Steuerung sehr einfach gehalten.

Genre:	Simulation, Abenteuer, RPG, Indie
Zielgruppe:	6+
Spieleranzahl:	Singleplayer
Plattform:	PC (Windows)
Inspiration:	bspw. Graveyard Keeper, Legend of Zelda, Stardew Valley
Grafik:	2D, Pixel Art

2.2 Story

In BlackSmith Saga steuert der Spieler einen Schmied, welcher durch verschiedene Quests auf der Suche nach Ressourcen ist. Er reißt durch die verschiedenen Biome um diese zu sammeln und sich somit Gold und Ruf zu verdienen.

2.3 Spielverlauf

Der Spieler startet Zuhause bei seiner Schmiede und kann sich zunächst umschauen. Mittig ist die Schmiede positioniert, wo der Spieler seine erste Quest annehmen kann. Anschließend muss er diese absolvieren und erhält seinen Ruf und Gold. Fortführend kann der Spieler weitere Quests annehmen und somit seinen Ruf und Gold steigern

2.4 Game Feature & Unique Selling Point

Blacksmith Saga ist ein Top-Down-Management-Spiel, in dem der Spieler einen Schmied steuert, der sich durch verschiedene Quest, Ruf und Gold erarbeitet. Er kämpft im Schnee- und Wüstengebiet gegen Gegner und kann Ressourcen sammeln.

2.4.1 Bedienungsanleitung

Taste	**Funktion**
WASD	Bewegen
Linke Maustaste	Kämpfen
0-9	Items aufnehmen
E	Interagieren, Quest annehmen/ abgeben
R	Quest ablehnen

2.5 Umsetzung

2.5.1 Grafik und Ästhetik

Blacksmith Saga besitzt eine Top-Down-2D-Ansicht und wurde anhand der verwendeten Assets erstellt. Das Spiel wurde in einer PixelArt erstellt.

2.5.2 Level Design

Blacksmith Saga besitzt keine Level, sondern basiert auf dem Erfüllen von verschiedenen Quests. Jedoch steuert der Spieler den Schmied durch 3 verschiedene Biome. Diese sind das Startgebiet, sowie Schnee- und Wüstenbiom. Durch unterschiedliche Kristalle, Ressourcen und Landschaften sind diese klar zu erkennen.

3. Eigene Assets

Nachfolgend werden alle erstellten Spieler-Assets und Komponenten genauer beschrieben.

3.1 Spieler

Der Hauptcharakter des Spiels wird als Schmied dargestellt. Er besitzt eine Axt in seiner Hand, womit er Ressourcen abbauen kann. Diese lagert er anschließend in seinem Rucksack, den er auf dem Rücken trägt.

3.1.1 Hauptkomponenten des Spielers

Der Spieler besitzt wie bereits geschrieben eine Spitzhacke und einen Rucksack, welche elementar wichtig sind, um alle Quests erfolgreich abschließen zu können.

3.2 Funktionen und zugehörige Visuelle Assets

Nachfolgend werden alle erstellten Grafikassets und Komponenten genauer beschrieben.

3.2.1 User Interfaces

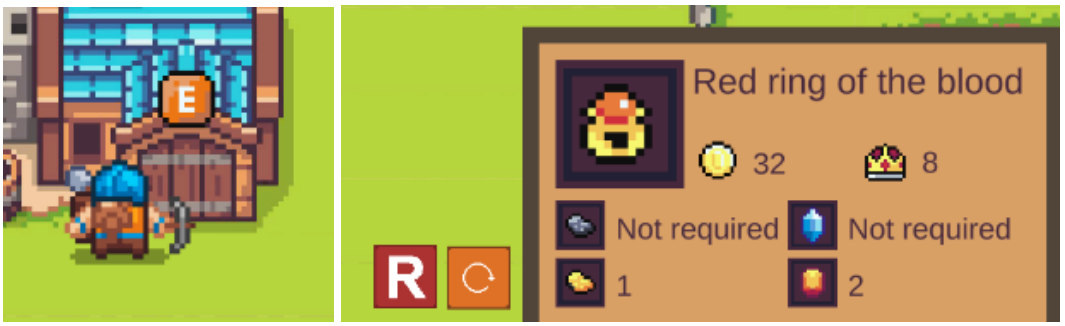
User Interfaces unterteilen sich in 2 Gruppen: Spieler und Interactables

3.2.1.1 Spieler

Spieler Ui setzt sich aus der Lebensanzeige, Gold/Ruhm-Anzeige, Tages- und Inventaranzeige.

3.2.1.2 Interactables

Der Spieler hat die Möglichkeit mit verschiedenen Objekten in der Welt zu interagieren. Dazu gehören: Hinweise, welche Taste man an dem Moment drücken muss, Questfenster und Shopfenster.



3.3 Gegner

Alle Gegner werden als Schleime in den Biomen dargestellt.

3.3.1 Gegner KI

Die Gegner werden durch Spawner in den Biomen erstellt und verfolgen bzw. attackieren den Spieler. Wenn der Spieler sich gegen die Gegner nicht wehrt und diese berührt, verliert er ein Leben und schreckt ein paar Meter zurück.

Kämpft der Spieler gegen diese Gegner, so kann er diese töten und sich somit den Weg zu weiteren Ressourcen freikämpfen.

Im Falle des Todes, wird ein Teil der gesammelten Ressourcen verloren gehen.

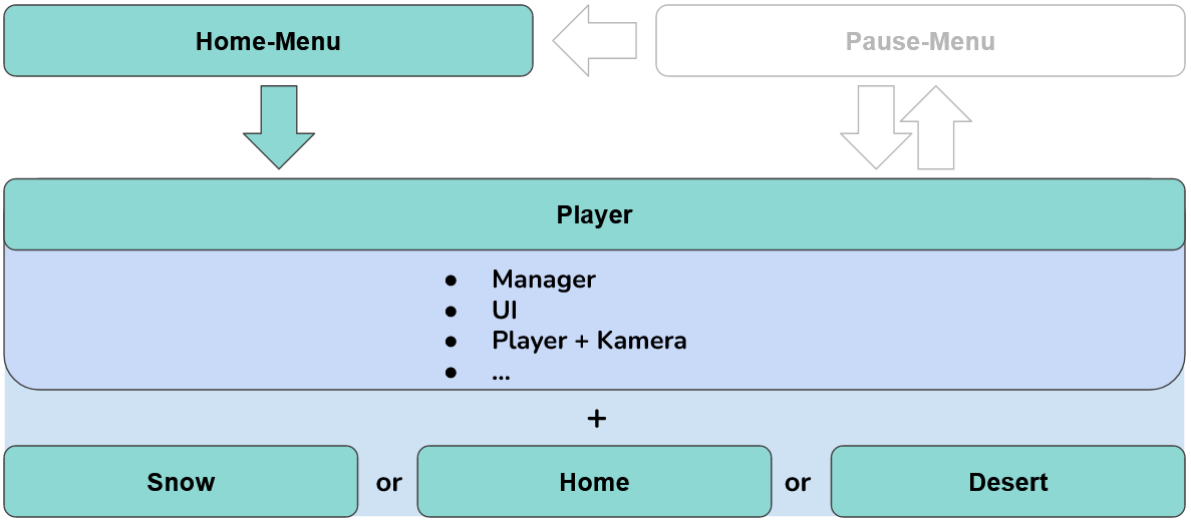
4. Entwicklung

4.1 Aufbau des Projekts

4.1.1. Szenenaufbau

Die Startszene des Projekts ist die “Home”-Szene. Von hier hat der Spieler die Möglichkeit sich in die Spielwelt zu begeben.

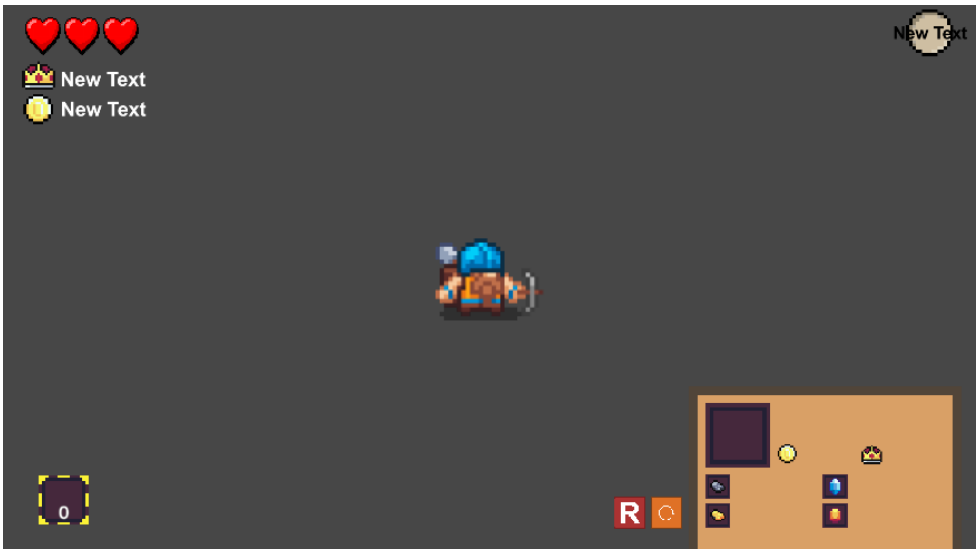
In der Spielwelt sind dem Spieler 3 Szenen (Home, Snow und Desert) während seiner Reise zur Verfügung, wobei die Player-Szene immer geladen wird und für das Managen der Spielabläufe zuständig ist.



Home-Menü

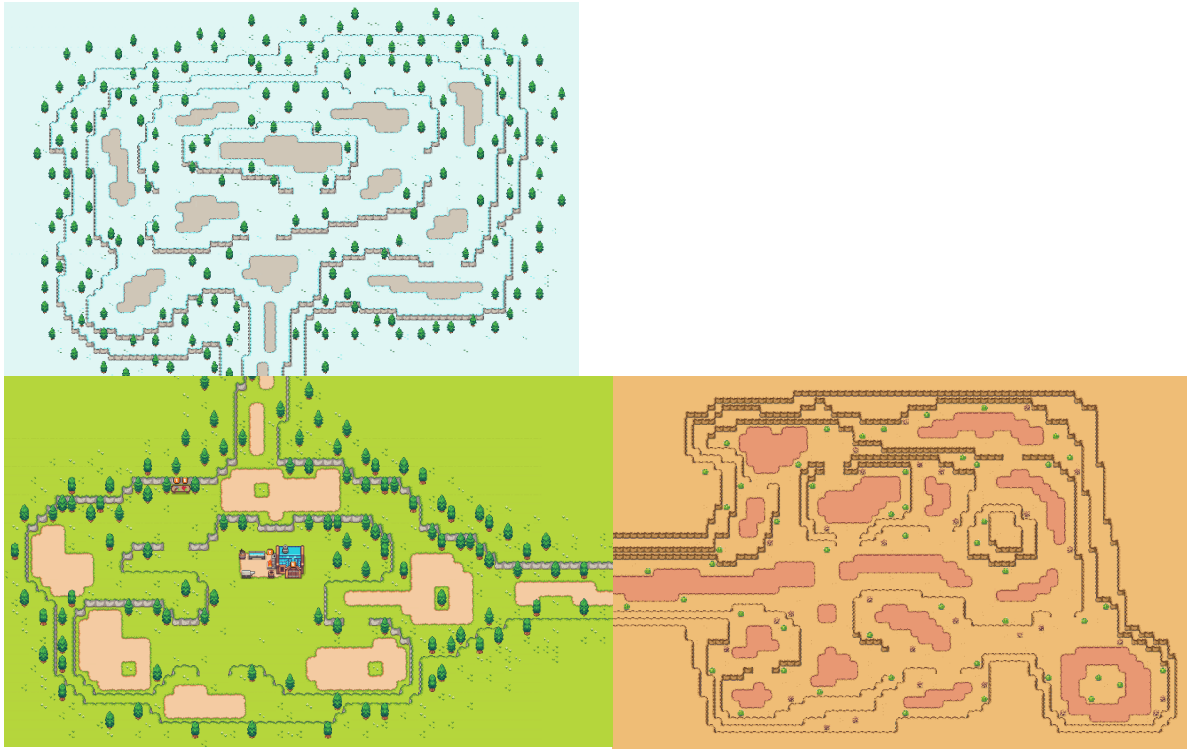
Die Startszene des Projekts ist die “Home”-Szene. Von hier hat der Spieler die Möglichkeit sich in die Spielwelt zu begeben.

Player



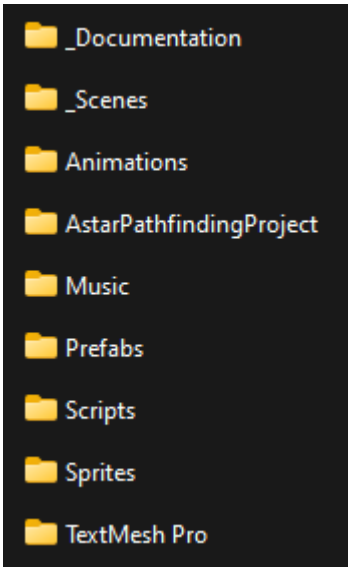
Die Player Szene ist die Grundszene, die für das Managen der Wichtigsten Spielabläufe zuständig ist.

Umgebungs-Szenen



In der Spielwelt stehen dem Spieler 3 Szenen (Home, Snow und Desert) während seiner Reise zur Verfügung. Je nach Szene - kann der Spieler unterschiedliche Ressourcen entdecken.

4.2 Ordnerstruktur



_Szenes: Fundort der 5 Szenen wie Menü, Player, Home, …

Animations: Animations-Ordner für Player- und Slime-Animationen

APP (A*): Projekt für die Implementierung von Gegnerlogik zur Zielfindung des Players

Music: Fundort für alle Gamesounds

Prefabs: Prefabordner für unter anderem Player, Slimes, Ressourcen, …

Scripts: Hier sind alle Scripte thematisch in Unterordnern sortiert zu finden

Sprites: Thematisch unterteilte Unterordner mit Sprites und Texturen des Spieles

4.3 Softwarearchitektur

Das Spiel basiert sich auf wichtigen Klassen und Managern, wobei es folgende wichtige gibt:

4.3.1 Game Managing

Im Folgenden werden die relevanten Funktionen zum Managen von Szenen und Spielerdaten als Code dargestellt:

GameManager.cs

```
private void Update()
{
        player = GameObject.FindGameObjectWithTag("Player");

        if (isLoaded == true)
        {
                // When the Savegame was loaded - try to load it
                try { LoadData(); } catch(Exception) { }
        }
}

public void RespawnPlayer()
{
        if(playerRespawn == false)
        {
                // The reset of player is being set to respawn area in case of save or death
                playerRespawn = true;
                Invoke("Restart", restartDelay);
```

```
                // In case of death - the player loses some amount of his resources
                foreach (Item item in Inventory.instance.itemList)
                {
                item.amount = Mathf.RoundToInt(item.amount / 3);
                }
                ResourcesManager.instance.moneyAmount =
                Mathf.RoundToInt(ResourcesManager.instance.moneyAmount / 3);
                ResourcesManager.instance.fameAmount =
                Mathf.RoundToInt(ResourcesManager.instance.fameAmount / 3);
                UI_Inventory.instance.RefreshInventoryItems();
        }
}
```

LevelLoader.cs

```
private void OnTriggerEnter2D(Collider2D collision)
{
        if (collision.tag == "Player")
        {
                // Unsetting the old scene and loading the new one
                gameObjects = GameObject.FindGameObjectsWithTag("Enemy");

                SceneManager.UnloadSceneAsync(unloadScene);
                SceneManager.LoadScene(loadScene, LoadSceneMode.Additive);

                // Destroy all the slime objects in the biome
                for (var i = 0; i < gameObjects.Length; i++)
                Destroy(gameObjects[i]);
        }
}
```

4.3.3 Save/ Load

Im Folgenden wird das Speichern über die Interaktion mit seinem Haus und das Laden der Spieldaten als Code dargestellt:

Save.cs

```
void SaveGame()
{
        // Saving of the money and fame resources
        PlayerPrefs.SetInt("money", ResourcesManager.instance.moneyAmount);
        PlayerPrefs.SetInt("fame", ResourcesManager.instance.fameAmount);

        // Saving o the inventory items
        PlayerPrefs.SetString("itemName0", "");
        [...]
```

```
        if (Inventory.instance.itemList.Count >= 1)
        {
                PlayerPrefs.SetInt("itemAmount0", Inventory.instance.itemList[0].amount);
                PlayerPrefs.SetString("itemName0",
                Inventory.instance.itemList[0].getItemTypeAsString());;

                if (Inventory.instance.itemList.Count >= 2)
                {
                        [...]
        }
        }

        // Respawn of the player
        FindObjectOfType<GameManager>().RespawnPlayer(0f);
        // Increasing of the day count and saving of it
        ResourcesManager.instance.dayAmount = ResourcesManager.instance.dayAmount + 1;
        PlayerPrefs.SetInt("day", ResourcesManager.instance.dayAmount);
}
```

GameManager.cs

```
public void LoadData()
{
        // Loading of the saved data
        ResourcesManager.instance.moneyAmount = PlayerPrefs.GetInt("money");
        ResourcesManager.instance.fameAmount = PlayerPrefs.GetInt("fame");

        if (PlayerPrefs.GetString("itemName0") != "")
        {
                Inventory.instance.AddItemFromString(PlayerPrefs.GetString("itemName0"),
                PlayerPrefs.GetInt("itemAmount0"));

                if (PlayerPrefs.GetString("itemName1") != "")
                {
                        [...]
                }
        }

        ResourcesManager.instance.dayAmount = PlayerPrefs.GetInt("day");

        // Flag to prevent the repeated loading
        isLoaded = false;
}
```

4.3.3 Player/ Slime

Im Folgenden werden die Main-Funktionalitäten von Player und Slime genauer erläutert.

PlayerController.cs

```
private void FixedUpdate()
{
        if (canMove == true && movementInput != Vector2.zero)
        {
                rigidbody.velocity =
                Vector2.ClampMagnitude(rigidbody.velocity + (movementInput * moveSpeed *
                Time.deltaTime), maxSpeed);
                // Sprite direction
                if (movementInput.x < 0)
                {
                spriteRenderer.flipX = true;
                }
                else if (movementInput.x > 0)
                {
                spriteRenderer.flipX = false;
                }
                // Animation trigger flag setting
                IsMoving = true;
        }
        else
        {
                IsMoving = false;
        }

        // Animation trigger flag setting for special animations
        if (GameObject.FindGameObjectWithTag("Player").GetComponent<Health>().dead ||
            animator.GetCurrentAnimatorStateInfo(0).IsName("Player_Hurt"))
        {
                canMove = false;
        }
        else
        {
                canMove = true;
        }
}


// Adding of the knockback distance and damage to the player
public void OnHit(float damage, Vector2 knockback)
{
        this.GetComponent<Health>().TakeDamage(damage);
        rigidbody.AddForce(knockback*10);
}
```

Health.cs

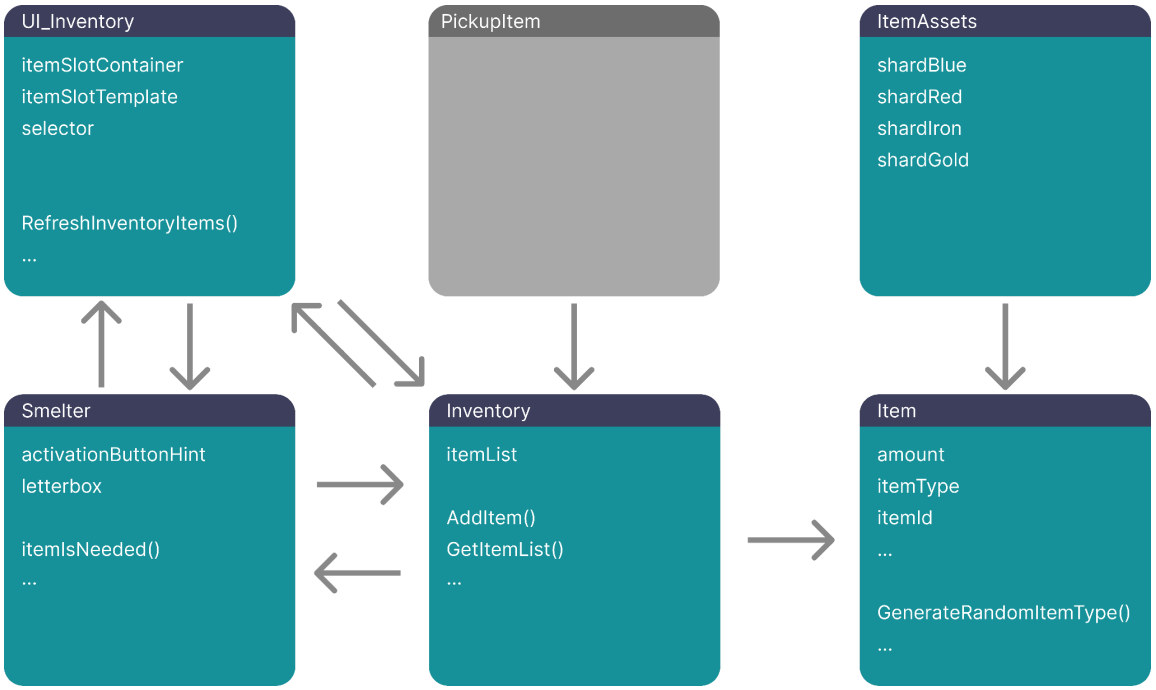
```
public void TakeDamage(float _damage)
{
        currentHealth = Mathf.Clamp(currentHealth - _damage, 0, startingHealth);
         // When the damage is not enough to kill the player - activate the “hurt”-flag trigger
        if (currentHealth > 0)
        {
                anim.SetTrigger("hurt");
        }
        // In the opposite case - activate the death sequence
        else
        {
                if(!dead)
                {
                anim.SetTrigger("die");
                dead = true;
                }
        }
}
```

Slime.cs

```
void FixedUpdate()
{
         // The slime loses its possibility to move when being damaged or after the death
        if (animator.GetCurrentAnimatorStateInfo(0).IsName("Slime_Damage") ||
            animator.GetCurrentAnimatorStateInfo(0).IsName("Slime_Death"))
        {
                this.aiPath.canMove = false;
        }
        else
        {
                this.aiPath.canMove = true;
        }
         // If the calculated path is directed to right - rotate to right
        if (aiPath.desiredVelocity.x >= 0.01f)
        {
                transform.localScale = new Vector3(0.32f, 0.32f, 1f);

                if(this.aiPath.canMove)
                {
                animator.SetBool("isMoving", true);
                }
        }
        // If the calculated path is directed to left - rotate to left
        else if (aiPath.desiredVelocity.x <= -0.01f)
        {
                transform.localScale = new Vector3(-0.32f, 0.32f, 1f);
```

```
                if (this.aiPath.canMove)
                {
                        animator.SetBool("isMoving", true);
                }
        }
        else
        {
                animator.SetBool("isMoving", false);
        }
}


private void OnTriggerEnter2D(Collider2D collision)
{
        if(collision.tag == "Player")
        {
                // If colliding with the player - activate the get-damage-sequence and add knockback
                Vector3 parentPosition =
                gameObject.GetComponentInParent<Transform>().position;

                Vector2 direction =
                (Vector2)(collision.gameObject.transform.position - parentPosition).normalized;
                Vector2 knockback = direction * knockbackForce;

                collision.GetComponent<PlayerController>().OnHit(damage, knockback);
        }
}
```

4.3.4 Inventory



In dem Inventory-System spielt die Inventory Klasse die wichtigste Rolle: Empfang von Items, Signalgabe an das Interface über neue Items und das Tracken der Items werden durch Inventory.cs implementiert.

Im Folgenden wird das Hinzufügen eines Items als Code dargestellt:

Inventory.cs

```
public void AddItem(Item item)
{
    bool itemIsAlreadyInside = false;

    foreach (var itemInList in itemList)
    {
            if (item.itemType == itemInList.itemType)
            {
                itemIsAlreadyInside = true;
                itemInList.amount += 1;
            }
    }

    if (!itemIsAlreadyInside)
    {
        itemList.Add(item);
    }

    UI_Inventory.instance.RefreshInventoryItems();
}
```

Im Folgenden wird das Erneuern der Inventory-Oberfläche als Code dargestellt:

Inventory.cs

```
public void RefreshInventoryItems()
{
    int x = 0;
    int y = 0;
    float itemSlotCellSize = 100f;

    var inventoryList = Inventory.instance.itemList;

     // For all the items in the inventory
    for (int i = 0; i < inventoryList.Count; i++)
    {
            // “Draw” a slot with inventory item 100f further from the previous one
            RectTransform itemSlotRectTransform =
             Instantiate(itemSlotTemplate,itemSlotContainer).GetComponent<RectTransform>();
            itemSlotRectTransform.gameObject.SetActive(true);
```

```
                // Fill it with appropriate image
                itemSlotRectTransform.anchoredPosition =
                 new Vector2(x * itemSlotCellSize, y * itemSlotCellSize);
                Image image = itemSlotRectTransform.Find("image").GetComponent<Image>();
                image.sprite = inventoryList[i].GetSprite();

                // Fill it with appropriate count of the item
                TextMeshProUGUI uiText =
                 itemSlotRectTransform.Find("text").GetComponent<TextMeshProUGUI>();
                 uiText.SetText(inventoryList[i].amount.ToString());
                x++;
        }

        selector.transform.localPosition =
        new Vector2(selectedItemIndex * itemSlotCellSize, y * itemSlotCellSize);
}
```

4.3.5 ResourceManager

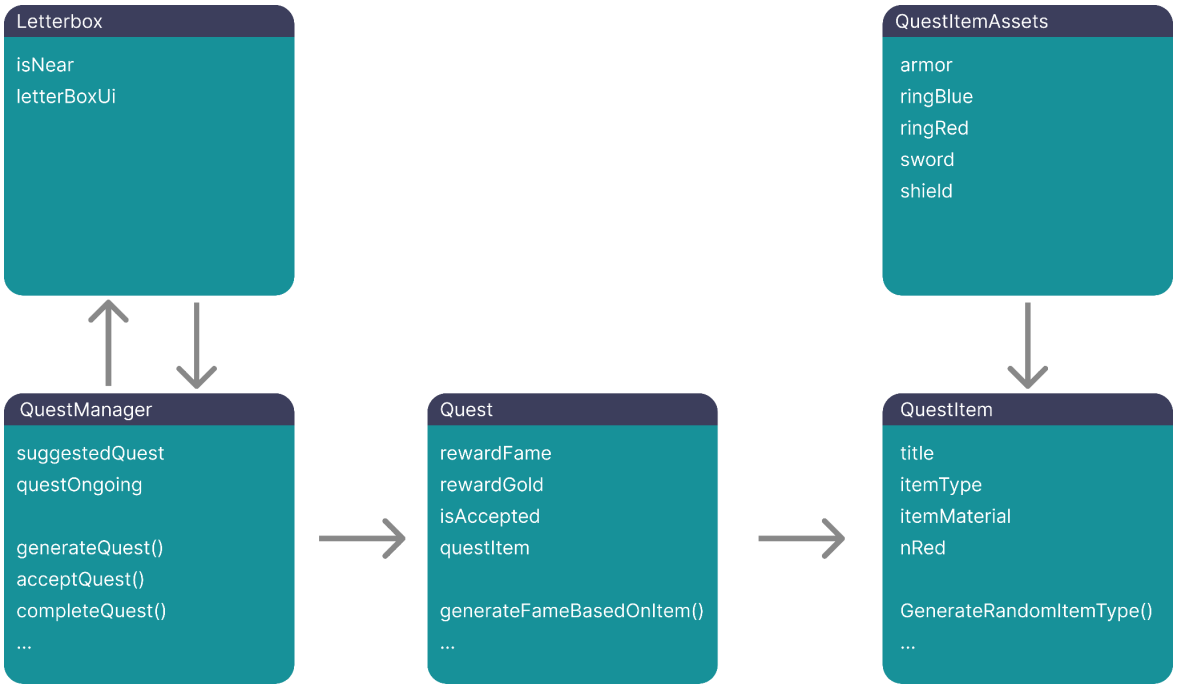
Das Sammeln der Ressourcen ist durch die Skript PickupItem.cs realisiert. Die Anzeige der gesammelten Ressourcen ist dann durch Inventory geregelt.

PickupItem.cs

```
private void Update()
{
        despawnTime -= Time.deltaTime;
        if (despawnTime < 0)
        {
            Destroy(gameObject);
        }

        float distance = UnityEngine.Vector3.Distance(transform.position, player.position);

        Debug.Log(distance);
        Debug.Log(distance > pickupDistance);

        // When the item is laying out of the player radius - just return the function
        if (distance > pickupDistance)
        {
            return;
        }

        // In opposite case - start pulling the item in the direction of the player
        transform.position =
        UnityEngine.Vector3.MoveTowards(transform.position, player.position, speed * Time.deltaTime);
        if (distance < 0.1f)
        {

            // When the thereached the player - create new Item object based on the id
            Item newItem = new Item();
```

```
            switch (itemId)
            {
                case 1:
                    newItem.itemType = Item.ItemType.BlueShard;
                    break;
                case 2:
                    newItem.itemType = Item.ItemType.RedShard;
                    break;
                case 3:
                    newItem.itemType = Item.ItemType.GoldShard;
                    break;
                case 4:
                    newItem.itemType = Item.ItemType.IronShard;
                    break;
            }

            // Initialise the new item through inventory, so it can be displayed in UI
            Inventory.instance.AddItem(new Item { itemType = newItem.itemType, amount = 1 });
            Letterbox.instance.RefreshQuestUi();
            Destroy(gameObject);

        }
    }
```

4.3.6 QuestManager



In dem Quest-System spielt die Quest-Manager-Klasse die wichtigste Rolle: Generieren von Quests, Signalgabe an das Interface über neue Quests und das Tracken der Quests werden durch QuestManager.cs implementiert. Im Folgenden wird die Abschluss einer Quest als Code dargestellt:

QuestManager.cs

```
public bool completeQuest()
{
	if (questOngoing)
	{
		var reqsFulfilled = false;

		// When enough required crystals are in inventory - set reqsFulfilled flag true
		if (suggestedQuest.questItem._nBlue == -1 ||
		suggestedQuest.questItem._nBlue ==  0)
		{
			reqsFulfilled = true;
		}
		else
		{
			return false;
		}

		[...]

		// When all requirements are fulfilled to this point - the quest is completable
		if (reqsFulfilled)
		{
			ResourcesManager.instance.moneyAmount +=
			suggestedQuest._rewardGold;
			ResourcesManager.instance.fameAmount +=
		suggestedQuest._rewardFame;

			questOngoing = false;
			QuestManager.Instance.playSuccessSound();
			this.suggestedQuest = null;
			generateQuest();
			return true;
	}
	}
	return false;
}
```

Im Folgenden wird die Generierung eines Namens eines Questitems dargestellt:

QuestItem.cs

```
public string GenerateItemName()
{
	var first = "";
	var second = "";
```

```
        switch (this.itemType)
        {
                default:
                case ItemType.ringBlue:
                    first = "Blue ring of ";
                    break;

                case ItemType.ringRed:
                    first = "Red ring of ";
                    break;

                case ItemType.armor:
                    first = "Armor of ";
                    break;

                case ItemType.sword:
                    first = "Sword of ";
                    break;

                case ItemType.shield:
                    first = "Shield of ";
                    break;
        }

        switch (this.itemMaterial)
        {
                default:
                case ItemMaterial.blue:
                    second = "the sky";
                    break;

                case ItemMaterial.red:
                    second = "the blood";
                    break;

                case ItemMaterial.iron:
                    second = "the might";
                    break;

                case ItemMaterial.gold:
                    second = "the sun";
                    break;
        }

        return first + second;
    }
```

4.3.7 ShopManager

Das Shopsystem ist sehr ähnlich dem Quest-System aufgebaut.
Die ShopManager-Klasse hat die wichtigste Rolle in dem System: Signalgabe an den Interface, Abfrage der Werte des Spielers (Gold/Ruhm) und die tatsächliche transaktion von den, sowohl auch das Prüfen ob der Spieler genug Ressourcen zur Verfügung hat.
Im Folgenden wird beispielhaft der Kauf eines Lebensbuffs als Code dargestellt:

ShopManager.cs

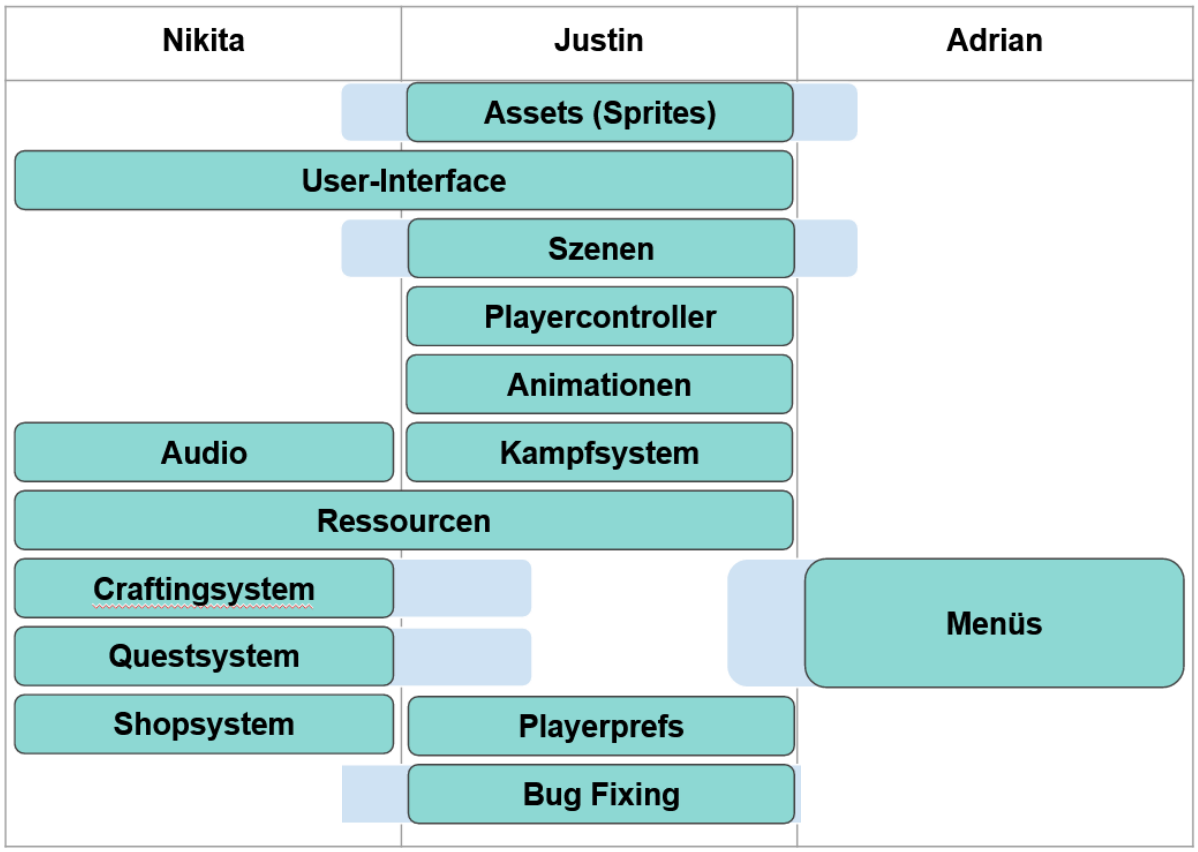
```
public void BuyHealth()
{
        // If player has enough money
        if (ResourcesManager.instance.moneyAmount >= 10)
        {
                // If player has not yet bought all the updates
                if(player.GetComponent<Health>().startingHealth < 6)
                {
                        // The transaction is being completed - player lose 10 gold and
                        receives the buff
                        ResourcesManager.instance.moneyAmount -= 10;
                        player.GetComponent<Health>().buyNewHealth();
                }
        }
}
```

5. Projektmanagement

5.1 Vorgehensweise

Das Projekt wurde begleitend mit dem Modul ITIG erstellt. Dabei wurden die in den Vorlesungen und Seminaren gezeigten Kenntnisse in dem Projekt eingearbeitet und versucht, umzusetzen.
Außerdem wurden Tutorials und Foren dazu genutzt, um Recherche zu betreiben und weitere Herangehensweisen für uns zu übernehmen.

5.2 Arbeitsaufteilung



6. Offene ToDo-s/Ideen

6.1. Mögliche Zukünftige Entwicklungen

- ein Tutorial einbauen
- mehr Nutzerfeedback / Sounds
- Pausenmenü
- optisch fehlende Sprites ergänzen/ Sprites verbessern
- mehr Gegnertypen (Bossgegner)
- Lagerung von Ressourcen
- Ausbau von Schmiede und Umgebung
- NPCs
- mehr Vielfalt (Ressourcen, Rezepte, Biome, usw.)

6.2. bekannte Bugs

- Möglichkeit, dass man durch den Rückstoß der Gegner in Wänden hängen bleibt
- Teilweise Überlappung von Sprites

7. Reaktionen von Testern

“Interaktion mit dem Nutzer ist etwas, woran man noch weiter arbeiten kann.”

Anonym

“Die visuelle Darstellung des Spiels ist sehr ansprechend, wenn auch nicht final.”

Anonym

“Es gibt leider nicht viele Möglichkeiten, das erworbene Geld auszugeben.”

Anonym

“Sehr cool, was ihr da geschaffen habt. Wenn ihr das weiter entwickeln würdet, könnte sicher noch ein vollwertiges Spiel daraus werden.”

Anonym

“Für euer erstes Spiel ist das schon echt gut geworden.”

Anonym

8. Anhang

8.1. Werkzeuge

- Projekterstellung: Unity - https://unity.com/de
- IDE: Visual Studio - https://visualstudio.microsoft.com/de/
- Soundbearbeitung: Audacity - https://www.audacityteam.org/
- Grafik:
    - Adobe Photoshop - https://www.adobe.com/de/products/photoshop.html
    - PixelArt - https://www.pixilart.com/
    - Asprite - https://www.aseprite.org/