Elective – DMAI0916

4th Semester project

Group 2

Game development report

24 May 2018

Andreas Richardsen

University College of Northern Denmark

Technology and Business

Computer Science AP Degree Programme

Game development report

Elective DMAI0916

Participants:

Andreas Richardsen

Supervisor: Ronni Hansen

Submission date: 24th May 2018

Contents

Introduction

In this report, I will be talking about the game that I decided to develop for my elective project, there will be three main sections Analysis, Implementation and Process. In analysis, I will be talking about the different choices I found in my research and the reason behind my decisions. In implementation, I will talk about how I decided to implement my decisions. In process, I will be talking about how I worked on this project.

Problem Statement

I want to develop an adventure/RPG were the world gets affected based on what you do and the game changes based on how you interact with NPC's and the decisions you make.

The game should be able to have quests, killing enemies, dialogues and a pause menu.

The player will wake up in an unknown place and will have to decide the outcome of his own fate; does he stay, escape, destroy everything, the choice is in the player's hand. My goal is to make an enjoyable game.

Main areas:

- Dialogue
- Questing
- Character controller
- Player inputs
- Story

Learning Goals

- Game design
- Unity
- Better programming skills
- Environment creation
- Character creation (3D modelling software)

Analysis

Software development methodology

In this part, I will be talking about the differences between agile and plan driven development, show the methodologies I have worked with (strengths and weaknesses), my choice and my reasoning behind it.

Agile vs plan driven development

Agile: Development is iterative and incremental, adaptive so requirements can change if necessary, it doesn't need to know all the requirements, only the ones needed for the next iteration, it embraces and welcomes change, contains multiple learning loops [2], has a balance between predictive work and adaptive unexpected new work, uses small sensible batch sizes which handles small functionalities [3], always expects and considers delay costs, adapts and re-plans instead of staying with one plan, make every iteration with quality. [1]

Plan driven: Development is done in phases and sequential, make decisions based on the phase the project is in, assumes all the requirements and decisions are known from the start, avoids change since it is disruptive and expensive, the whole process is known and predictable, large batches normally 100% is completed before next batch is started, do not consider delay costs, considers that everything will go after the plan, follows the process and expects everything to be done correct the first time, quality is tested at the end of the project. [1]

Unified process

Unified process is a use case driven, object oriented architecture centric and iterative process. UP uses use cases to describe the functionality of the system. Contains four phases:

- Inception: This is where you establish the business case for the system, define the risks, scope, obtain 10% of the requirements and estimate next phase effort
- Elaboration: in elaboration you must develop an understanding of the problem domain, system architecture, address significant risk factors, significant portions may be coded/tested, about 80% of the major requirements should be identified;
- Construction: In this phase, you will do the majority of the system design, programming and testing and build the remaining system in short iterations.
- Transition: In this last phase, you will deploy the system in its operating environment and deliver releases for feedback and deployment.

Iterate while incrementing functionality and architecture:

- Less project failure
- Early mitigation of risks
- Early visible progress
- Early feedback
- User engagement leading to system that meets the needs more appropriately
- Managed complexity
- Lessons learned from an iteration, can be used for later iterations

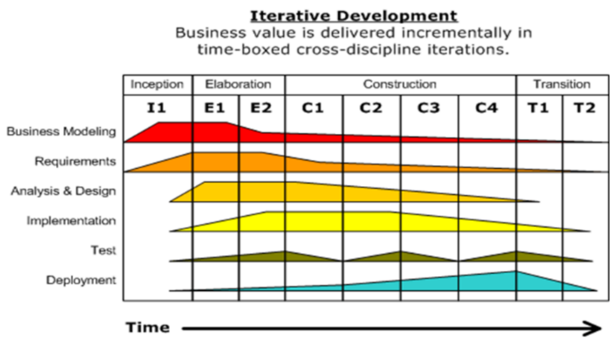


Figure1: Diagram of iterative development

Scrum

Scrum is a development methodology that is based on Scrum principles:

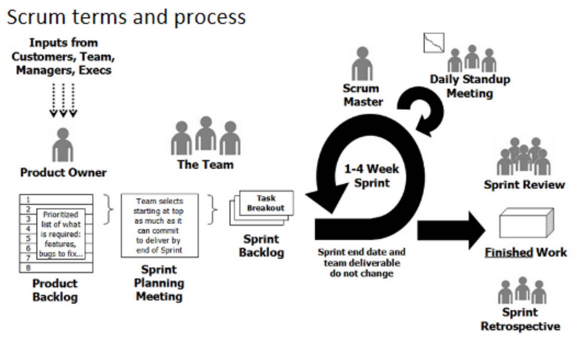

Scrum terms and process
Inputs from Customers, Team, Managers, Execs
Product Owner
The Team
Scrum Master
Daily Standup Meeting
1-4 Week Sprint
Sprint Review
Prioritized list of what is required: features, bugs to fix...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Sprint end date and team deliverable do not change
Finished Work
Sprint Retrospective

Figure2: Diagram of the scrum process

- Self-organising teams
- Stepwise development in sprints (iterations) – duration is normally four weeks
- Requirements are fixed for the sprint
- Requirements are listed in a product backlog
- Review with stakeholders at the end of each sprint
- Does not say anything on how you develop code, so you could use
    - Extreme programming [4]
    - any proven development method and patterns which works

A Scrum team consists of:

- Three roles:
    - Product owner
        - Typically a projects key stakeholder, but can also be someone from the companies marketing team. The only requirement for a product owner is that they have the vision for the system. The product owner's responsibilities is to have a vision of the system he wishes to build and successfully convey that to the scrum team, the product owner does this through the backlog.
    - Scrum master
        - Is the facilitator for the team. The scrum masters job is to manage the process of how information is exchanged, lead the daily stand up meetings, help the team come to consensus, remove obstacles that are damaging to the work, protect the team from outside distractions and help the team stay focused.
    - Team
        - Is the rest of the team. Most effective scrum teams are usually 5 to 7 members, they help and train each other so no one person becomes a bottleneck.
- Three (+1) ceremonies (meetings):
    - Sprint planning
        - The team selects from the product backlog what tasks to be done in this sprint starting from the top.
    - Sprint review
        - A meeting with the team at the end of the sprint were the team review what was done in the sprint
    - Daily stand-up meetings
        - Last, for about 15 minutes and in a meeting you ask three things: what did you do yesterday, what will you do today and are there any hindrances in your way?
    - Sprint retrospective
        - Evaluate the development process: what went well, what went not well, any surprises and where will we improve.
- Three artifacts

- Product backlog
  - Prioritised list of what is required: features, bugs to fix etc.
- Sprint backlog
  - List of tasks to be done in the sprint
- Burndown chart
  - A chart that tracks the tasks done in the sprint.

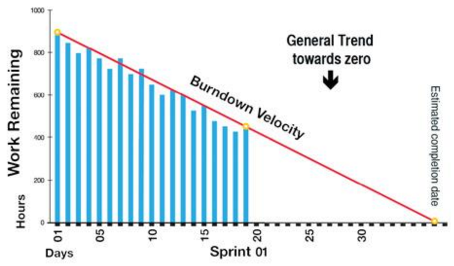


Figure3: Diagram of a burndown chart

Kanban

Is more agile than scrum with fewer rules, higher degree of freedom, but requires developers that are more experienced. It focuses on lean optimisation [5], e.g. removal and limitation of bottlenecks in the development process. Kanban has three simple rules:

1. Visualise the workflow (Kanban board)
2. Limit WIP (Work in progress)
   - Put a limit on the amount of items in the different swimlanes on the Kanban board
3. Measure average lead time (how long it takes to finish an item)

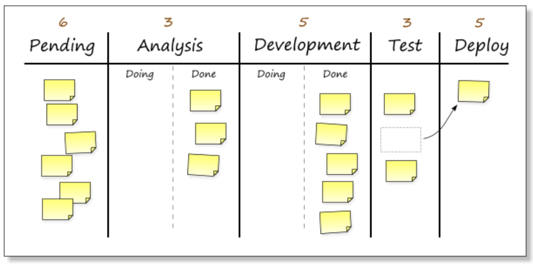


Figure4: Example of a Kanban board

Choice of method

One of my methods of choosing between agile and plan driven was to use Boehm's five forces. The scale of the forces: criticality, size, culture and dynamism show that an agile methodology is the best for me. The only exception is that I am not an experienced developer and since I am working with something that I have not used before it would be best to use a plan driven methodology. Since I am a one-man team, some of the benefits of plan driven are not usable for me e.g. having more experienced developers help in the process and therefore an agile process will fit me more as a developer. In conclusion, I have chosen to use Kanban as my methodology.

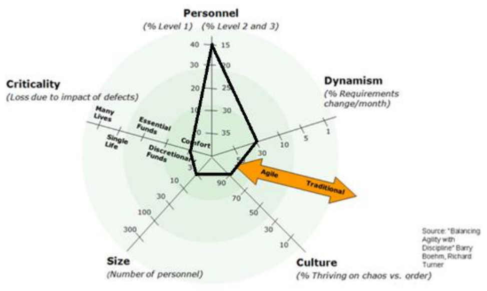


Figure5: My Boehm's five forces diagram

Genre

To find out what type of genre I wanted to make my game in, I thought about what type of game genres I like to play. I decided on making an adventure/RPG because I wanted a game that focuses on story and on building the character. Below you can see the different aspects of the genres I wanted to implement into my game.

Adventure:

- Puzzle solving
- Gathering and using items
- Story
- Dialogue
- Goals/success/failure

RPG:

- Exploration and quests
- Items and inventory
- Character actions and abilities
- Experience and levels
- combat

Story

The player wakes up in an unknown place that he has never seen. The story follows the player, as he is placed in a different dimension by Death himself. The player must find either a way to get back to his home world or a way to survive and live in the new dimension. The dimension he was placed in is unstable and the space around it is being constantly warped, so there is not one theme for the game but multiple e.g. futuristic, prehistoric, etc. therefore, the choice is in the players hand.

Game engines

In this section, I will be talking about using the two most used game engines and a self-created one. In the end I will give the reasoning behind my choice.

Unity

Is a game engine that is programmed in C#, it is well documented [6], royalty free so you only pay for the subscription, is being updated and worked on regularly and has a huge community that contains walkthroughs and help with code [7]. Self-proclaims themselves as the world's leading content- creation engine [8].

Unreal Engine 4

Self-proclaims itself as the most powerful creation engine [9]. Programmed in C++ and considered a more difficult but better performing engine. Unreal engine is used in games that are graphically intensive and important.

Self-created Engine

Creating an engine from scratch is a hard and time-consuming task, which requires that you have a great amount of experience in game design and a good understanding of game engines. You can write the game engine in whatever language you want and the game engine will be made specifically for your game.

Choice of Engine

I chose to use Unity as my game engine because I have experience in C#; Unity has a great documentation and a community that offers help, it's known as the easier engine between Unity and Unreal. I did not decide on making my own game engine because I do not have any experience with game engines and if I were to create a game engine, I would write it in C++, which would require me to learn C++ in addition to creating the game engine.

In the future, I would decide to create my own game engine that is custom made for the type of game I would make and it would be programmed using C++.

Game mechanics

At the start of the planning of the game I needed to figure out what type of mechanics, I wanted there to be in the game. I started by looking at what is common mechanics in games with the same genres as the ones I decided on.

The game mechanics that I chose to implement:

- Quests
- Combat
- Teleporting
- Dialogue

The game mechanics that I would implement in the future:

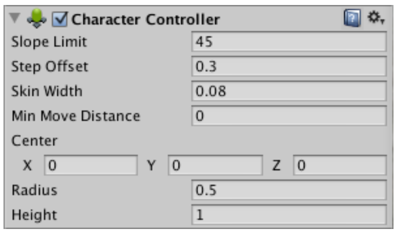
- Trading
- Spells

Character controller

One of the first things I needed to address was how I wanted to control my character. The two different methods I decided between was Unity's character controller or making my own.

Unity Character controller

The Unity character controller is not affected by physics and moves according to the environment (colliders) [10], it will only move when you call its move functions [11]. The character controller will do the basic movements based on the parameters you give it, it will do basic movement but anything advanced must be manually coded e.g. jumping [10].



Character Controller	
Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0
Center	X 0 Y 0 Z 0
Radius	0.5
Height	1

Figure6: Unitys character controller component

Self-created controller

With a self-created controller you can decide on what method you want to use for moving your character, you can use rigidbody, transform or any other way you can think of.

Conclusion

I decided to use my own controller in which I used a rigidbody for the movements, this allows me to use gravity and physics in the way I want, instead of coding all the physics according to Unity's character controller.

Player inputs

One of the features that is in majority of games is the ability to change the inputs. In Unity by default, you can only change your inputs before you start the game, and every gamer knows you will change inputs in game and sometimes you will have to look at the inputs to know what the button does.

Unity Input Manager

Unity's input manager lets you define the different input axes and game actions, but it does not let you change it inside the game, only before you start the game.

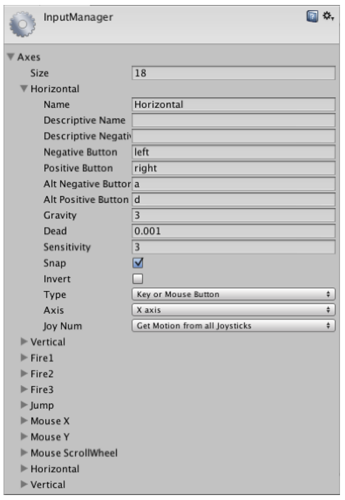
Self-created Input Manager

By creating your own input manager, you can have the player change them inside the game and you will have full control over it.

The only problem is that it requires a lot of time and you will have to create a UI. You can also buy one of the many input managers that are in the asset store.

Choice of input manager

I decided on creating my own input manager because in my opinion being able to configure your inputs mid game is a key feature that I think all games should have.



```
InputManager
▼ Axes
    Size                  18
  ▼ Horizontal
      Name                  Horizontal
      Descriptive Name
      Descriptive Negati
      Negative Button       left
      Positive Button       right
      Alt Negative Buttor   a
      Alt Positive Button   d
      Gravity               3
      Dead                  0.001
      Sensitivity           3
      Snap                  ☑
      Invert                ☐
      Type                  Key or Mouse Button
      Axis                  X axis
      Joy Num               Get Motion from all Joysticks
  ▶ Vertical
  ▶ Fire1
  ▶ Fire2
  ▶ Fire3
  ▶ Jump
  ▶ Mouse X
  ▶ Mouse Y
  ▶ Mouse ScrollWheel
  ▶ Horizontal
  ▶ Vertical
```

Figure7: Unitys input manager

UI

One of the important tasks in game design is to create a UI that is pleasant on the eye and that the player can use with ease. In the start I decided on starting with sketches for the UI elements that I knew I would implement now or in a later iteration so I had a general idea how I wanted the games UI to look.

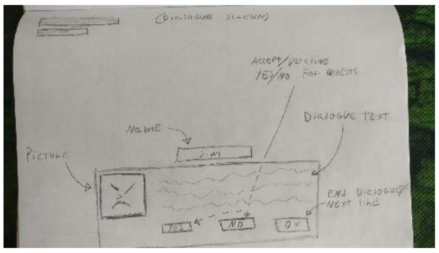


Figure8: Dialogue sketch

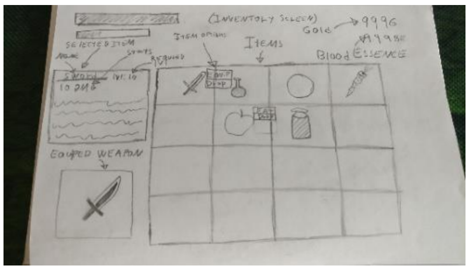


Figure9: Inventory sketch

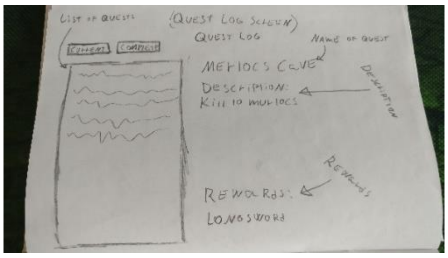


Figure10: Quest log sketch

- ➢ Default UI which contains the players hit points bar, experience bar and level.
- ➢ Dialogue UI that contains a field for the NPC's dialogue, picture, name and buttons that the player could interact with.
- ➢ Merchant UI that contains the merchants name and inventory, the player's inventory, the amount of currency the player has and buttons for buying and selling.
- ➢ Inventory UI that contains the player inventory, item currently equipped with information about it and you could interact with the items by right clicking them.
- ➢ Quest log UI that contains all the quests the player had been assigned, completed and information about the quest that the player highlights.

Dialogue system

For the dialogue system that I deemed to be an important part of the game since I am creating an RPG is something I was thinking about as soon as I started the project. I had the choice between buying one from the asset store that I deemed unnecessary and not worth the money, creating a simple one with the inspector or use JSON.

Simple – Inspector
The simplest and easiest way is to make every quest be in its own script, and make the dialogue in the inspector. Doing this comes with risks, if you change the fields that are in the inspector, all your dialogue will be lost and you must write it again. If you have to recreate your scene, you must rewrite all the dialogue.

Advanced – JSON
By using JSON, you can easily change dialogue, which means that you are safe from losing the dialogue and having to recreate it because of Unity. The problem with this is you must have a good understanding of JSON. You can do more complex dialogue e.g. have dialogue trees that change based on the different actions you have taken in the game.

Choice of dialogue system
My choice of dialogue system landed on the simple inspector, due to my lack of knowledge and experience with JSON and Unity. In the future, I would like to implement JSON so the game could have advanced dialogue trees that would have branches dependent on how you progressed in the game.

Inventory system
When researching how to do the inventory system, I found many different ways. I could either make it myself or buy it from the asset store. The biggest problem was to know what I wanted to put in my inventory system. What I wanted in the start was to have one inventory for my player and one for the proximity of the player that he could pick up from, currently equipped weapon information and to right click items so the different options came up for that item.

I decided in the end to have a player inventory where I could click on the item to get information and stats for the item, an action button on the selected item, a field that displays the players stats and shows them changing when you equip items, currently equipped weapon and the player could pick up items that are on the ground by standing on it and pressing the interact button.

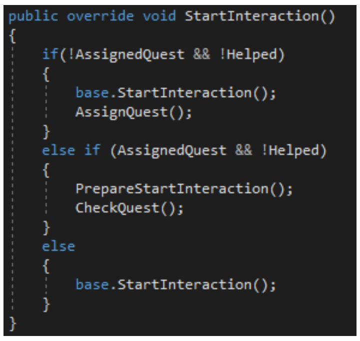
Implementations
In this section I will be talking about my implementations, how I implemented it and show pieces of code of how I implemented them.

Game mechanics

Every game has game mechanics and here I will be showing how I implemented my chosen game mechanics.

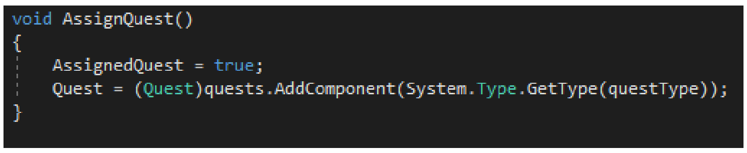
Quest

The player will go to a QuestGiver (NPC), interact with the QuestGiver and obtain the QuestGiver's dialogue. In the dialogue, the QuestGiver will tell the player what it wants and the player must fulfil the quests goals. In order for the player to finish the quest, he must complete all the goals for the quest. There are two types of goals, killing and collecting.



```
public override void StartInteraction()
{
    if(!AssignedQuest && !Helped)
    {
        base.StartInteraction();
        AssignQuest();
    }
    else if (AssignedQuest && !Helped)
    {
        PrepareStartInteraction();
        CheckQuest();
    }
    else
    {
        base.StartInteraction();
    }
}
```

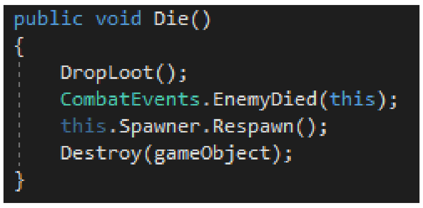
Figure11: Implementation of when the dialogue starts



```
void AssignQuest()
{
    AssignedQuest = true;
    Quest = (Quest)quests.AddComponent(System.Type.GetType(questType));
}
```

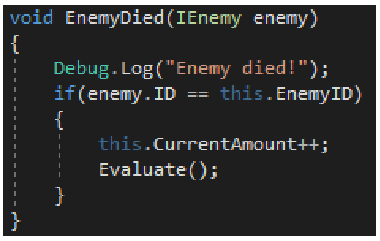
Figure12: Implementation of when the QuestGiver assigns the quest

The killing goal is completed when the player has killed the required amount of the requested character from the point of when the quest started. When the player kills an enemy an event listener is called in the enemies Die() method, that calls the method for the active kill goals to see if the enemy that died has the same ID as the enemy in the quest. If the enemy has the same ID, the method calls another method called Evaluate(). This method evaluates if the current amount is equal to the required amount, if the condition is met, then the Complete() method is called. The Complete() method sets completed to true and calls the method CheckGoals() for the quest for which the goal is attached to. CheckGoals() sets completed to true if all the goals are completed. When the player interacts with the QuestGiver, the QuestGiver will see whether the quest is completed. If the quest is not completed he will just say, "Do the job or no reward!!!", but if it is completed the quest will be set to complete and the player will receive the reward.



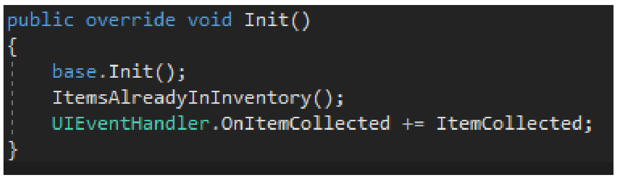
```
public void Die()
{
    DropLoot();
    CombatEvents.EnemyDied(this);
    this.Spawner.Respawn();
    Destroy(gameObject);
}
```

Figure13: Implementation of the enemy's Die() method



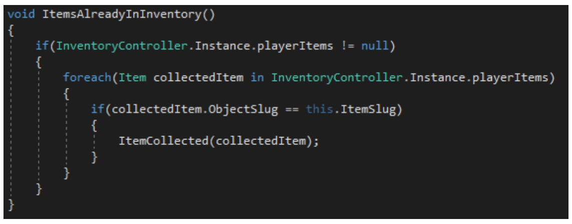
```
void EnemyDied(IEnemy enemy)
{
    Debug.Log("Enemy died!");
    if(enemy.ID == this.EnemyID)
    {
        this.CurrentAmount++;
        Evaluate();
    }
}
```

Figure14: Implementation of the method that checks if the killed enemy is the same as the quest enemy



```
public override void Init()
{
    base.Init();
    ItemsAlreadyInInventory();
    UIEventHandler.OnItemCollected += ItemCollected;
}
```

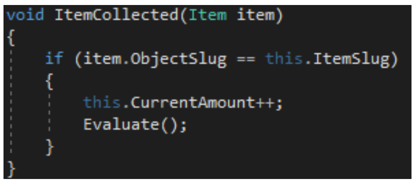
Figure15: The initialiser for CollectGoal



```
void ItemsAlreadyInInventory()
{
    if(InventoryController.Instance.playerItems != null)
    {
        foreach(Item collectedItem in InventoryController.Instance.playerItems)
        {
            if(collectedItem.ObjectSlug == this.ItemSlug)
            {
                ItemCollected(collectedItem);
            }
        }
    }
}
```

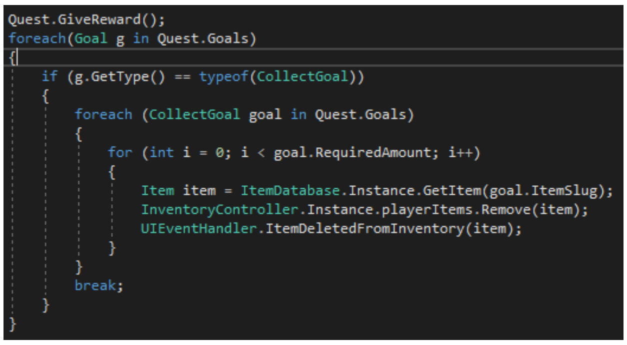
Figure16: Implementation for checking if the player have the quest item(s)

The collecting quest is completed when the player has collected the required items. Contrary to the killing quest the items can be collected before the quest is started. When the player first receives the quest the CollectGoal class calls the method ItemsAlreadyInInventory() that checks if any of the items in the players inventory are the same as the item in the quest. For each item that matches the quest item the ItemCollected(Item item) method is called which increments the current amount with one, then it calls the Evaluate() method which evaluates to see if the current amount matches the required amount. If the required amount is met the Complete() method is called, which sets completed to true, then the CheckGoals() method is called which sets completed to true if all the goals are completed. When the player adds an item to the player inventory, the event handler calls ItemCollected(Item item) and the collect goal listen calls the method ItemCollected(Item item) again and then goes thru the same process. When the player interacts with the QuestGiver after completing the quest, the QuestGiver will go through all of the quest goals to see if any of them are of the class CollectGoal. If goals contains a CollectGoal then it will go



```
void ItemCollected(Item item)
{
    if (item.ObjectSlug == this.ItemSlug)
    {
        this.CurrentAmount++;
        Evaluate();
    }
}
```

Figure17: Implementation that check if the collected item is the same as the quest item



```
Quest.GiveReward();
foreach(Goal g in Quest.Goals)
{
    if (g.GetType() == typeof(CollectGoal))
    {
        foreach (CollectGoal goal in Quest.Goals)
        {
            for (int i = 0; i < goal.RequiredAmount; i++)
            {
                Item item = ItemDatabase.Instance.GetItem(goal.ItemSlug);
                InventoryController.Instance.playerItems.Remove(item);
                UIEventHandler.ItemDeletedFromInventory(item);
            }
        }
        break;
    }
}
```

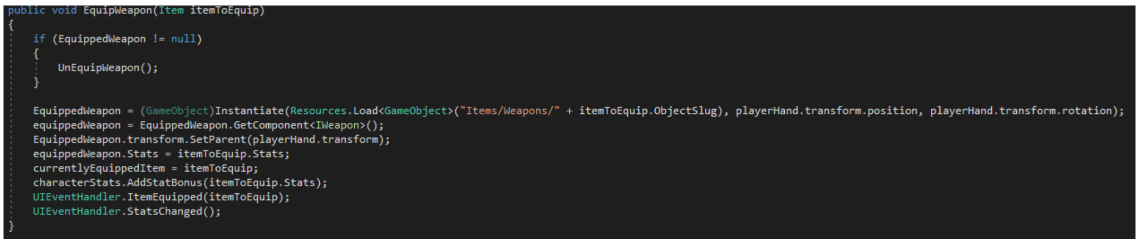
Figure18: implementation of removing CollectGoal item(s) from inventory

through every CollectGoal and remove the required amount of the quest item from the player's inventory.

Combat

The combat system is very simple and not something that I wanted to be complicated in the first iteration. In the future, I would like to add spells and combos.

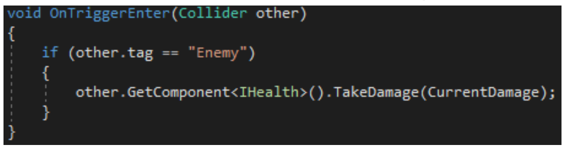
Everything that is able to take damage implements the IHealth interface that contains the methods TakeDamage(int amount), Heal(int amount), Revive(int amount) and Die(). The player can equip and unequip any weapon in his inventory and can only attack while a weapon is equipped. When the player presses the attack button the PerformAttack() method is called which checks if a weapon is equipped, then it calls the weapons IWeapon method PerformAttack(int damage) with the CalculateDamage() method as its parameter. The CalsulateDamage() method gets the player stat AttackDamage and returns it. In the IWeapon's PerformAttack(int damage), the CurrentDamage is set to the damage that is passed from CalculateDamage() and triggers the attack animation.



```
public void EquipWeapon(Item itemToEquip)
{
    if (EquippedWeapon != null)
    {
        UnEquipWeapon();
    }

    EquippedWeapon = (GameObject)Instantiate(Resources.Load<GameObject>("Items/Weapons/" + itemToEquip.ObjectSlug), playerHand.transform.position, playerHand.transform.rotation);
    equippedWeapon = EquippedWeapon.GetComponent<IWeapon>();
    EquippedWeapon.transform.SetParent(playerHand.transform);
    equippedWeapon.Stats = itemToEquip.Stats;
    currentlyEquippedItem = itemToEquip;
    characterStats.AddStatBonus(itemToEquip.Stats);
    UIEventHandler.ItemEquipped(itemToEquip);
    UIEventHandler.StatsChanged();
}
```

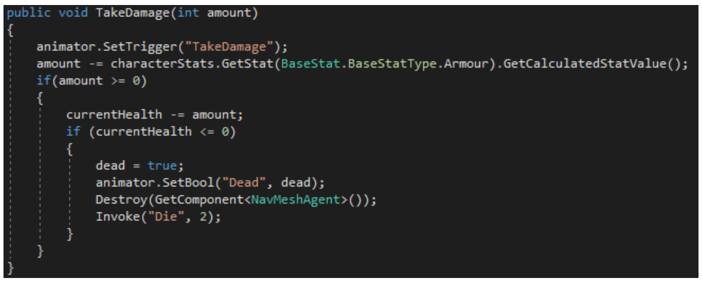
Figure19: Implementation of EquipWeapon(Item itemToEquip)

The players attack uses colliders on the weapon. If the weapons collider hits an enemy that has the tag Enemy then then weapon calls the enemy's TakeDamage(int amount) method which tells the enemy to take damage equal to the players current damage (players AttackDamage). When the enemy takes damage it checks to see if the damage the player did was more than the enemy's armour stat. If the damage was less then he will do no damage, but if it was more, the enemy loses that amount of health from its current health. If its current health drops below zero, dead is set to true, die animation is set to true, navmesh is destroyed and the Die() method is invoked after 2 seconds. The reason for invoking the Die() method after 2 seconds is so the animation clip is seen and the game seem more natural rather than the enemy just disappearing. When the enemy Die() method is invoked, it calls the DropLoot() method, calls the combat events handler, and destroys the game object. DropLoot() gets the loot items from the enemy and instantiates the item on the ground where the enemy was.



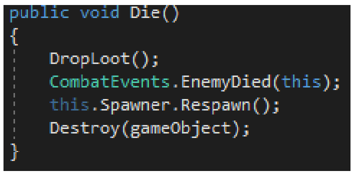
```
void OnTriggerEnter(Collider other)
{
    if (other.tag == "Enemy")
    {
        other.GetComponent<IHealth>().TakeDamage(CurrentDamage);
    }
}
```

Figure20: The weapons method for damaging an enemy



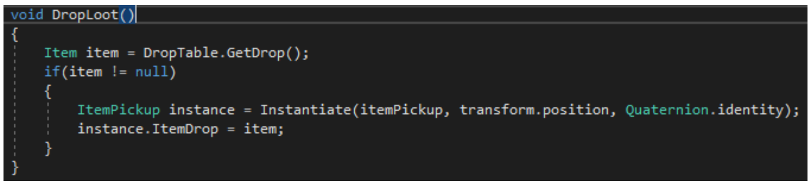
```
public void TakeDamage(int amount)
{
    animator.SetTrigger("TakeDamage");
    amount -= characterStats.GetStat(BaseStat.BaseStatType.Armour).GetCalculatedStatValue();
    if(amount >= 0)
    {
        currentHealth -= amount;
        if (currentHealth <= 0)
        {
            dead = true;
            animator.SetBool("Dead", dead);
            Destroy(GetComponent<NavMeshAgent>());
            Invoke("Die", 2);
        }
    }
}
```

Figure21: Implementation of the enemys TakeDamage(int amount)



```
public void Die()
{
    DropLoot();
    CombatEvents.EnemyDied(this);
    this.Spawner.Respawn();
    Destroy(gameObject);
}
```

Figure22: The enemys Die() method



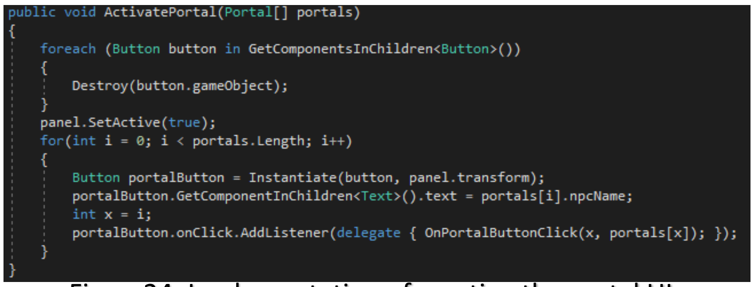
```
void DropLoot()
{
    Item item = DropTable.GetDrop();
    if(item != null)
    {
        ItemPickup instance = Instantiate(itemPickup, transform.position, Quaternion.identity);
        instance.ItemDrop = item;
    }
}
```

Figure23: The enemys DropLoot() method

Teleporting

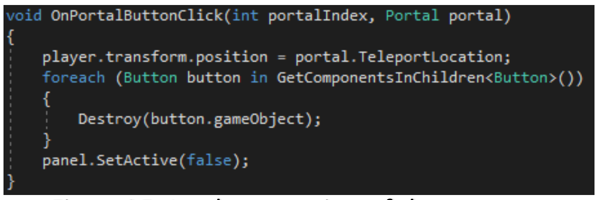
To make the player able to get to one location fast from another I implemented teleportation, in the future the player will only be able to teleport to locations that he has already discovered. There are two types of portals implemented, the portal that connects to all the other portals and a passage portal that just connect to one other portal that works as a road.

The portals that are connected to each other uses a portal controller. When the player stands next to the portal and presses the interact button, the portal controllers ActivatePortal(Portal[] portals) method is called. When the ActivatePortal(Portal[] portals) method is called it sets up the UI and instantiates a button for each of the portals that are connected to the portal the player interacted with. When the player presses the button to teleport to a portal, the player teleports to that portals location and all the buttons are destroyed.



```
public void ActivatePortal(Portal[] portals)
{
    foreach (Button button in GetComponentsInChildren<Button>())
    {
        Destroy(button.gameObject);
    }
    panel.SetActive(true);
    for(int i = 0; i < portals.Length; i++)
    {
        Button portalButton = Instantiate(button, panel.transform);
        portalButton.GetComponentInChildren<Text>().text = portals[i].npcName;
        int x = i;
        portalButton.onClick.AddListener(delegate { OnPortalButtonClick(x, portals[x]); });
    }
}
```

Figure24: Implementation of creating the portal UI



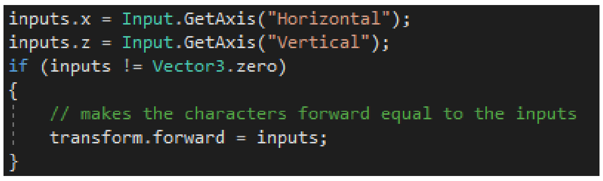
```
void OnPortalButtonClick(int portalIndex, Portal portal)
{
    player.transform.position = portal.TeleportLocation;
    foreach (Button button in GetComponentsInChildren<Button>())
    {
        Destroy(button.gameObject);
    }
    panel.SetActive(false);
}
```

Figure25: Implementation of the portals teleport button

The passages that are only connected to one other use a collider. When the player enters the collider the Passage class checks the players tag, if the tag equals Player it teleports the player to the location of the other gate.

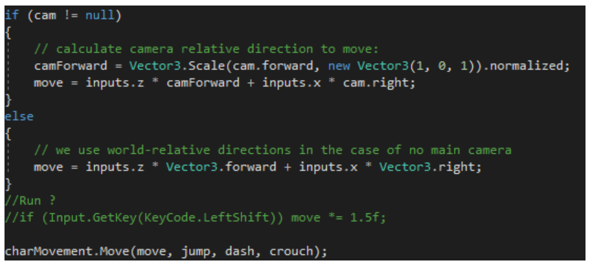
Character controller

Since I decided on making my own character controller I had to decide what I would allow the character to do, the character can walk, jump and crouch.



```
inputs.x = Input.GetAxis("Horizontal");
inputs.z = Input.GetAxis("Vertical");
if (inputs != Vector3.zero)
{
    // makes the characters forward equal to the inputs
    transform.forward = inputs;
}
```

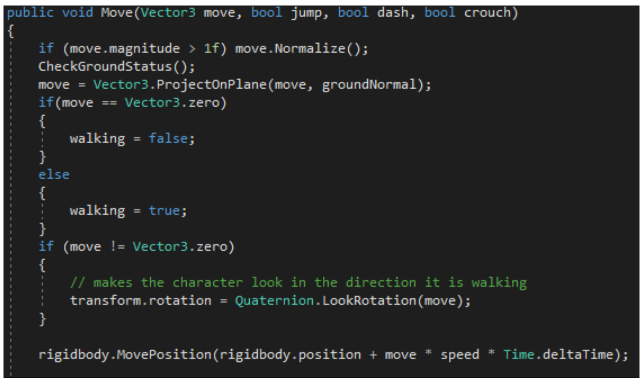
Figure26: The players x and y inputs



```
if (cam != null)
{
    // calculate camera relative direction to move:
    camForward = Vector3.Scale(cam.forward, new Vector3(1, 0, 1)).normalized;
    move = inputs.z * camForward + inputs.x * cam.right;
}
else
{
    // we use world-relative directions in the case of no main camera
    move = inputs.z * Vector3.forward + inputs.x * Vector3.right;
}
//Run ?
//if (Input.GetKey(KeyCode.LeftShift)) move *= 1.5f;

charMovement.Move(move, jump, dash, crouch);
```

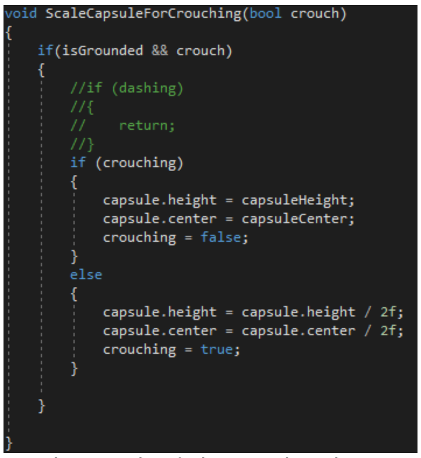
Figure27: implementation of the player moving in the direction of the camera

The character is controlled by the Player class, which sends the characters vector movement, if the jump/crouch button was pressed to the PlayerMovement class. In the PlayerMovement class all the movement is then decided, based on whether the player is jumping, crouching, etc. the PlayerMovement class must also measure if the player is standing on ground which it does by sending a ray cast a set distance and checking if it hits the layer mask that that is set in the inspector.



```
public void Move(Vector3 move, bool jump, bool dash, bool crouch)
{
    if (move.magnitude > 1f) move.Normalize();
    CheckGroundStatus();
    move = Vector3.ProjectOnPlane(move, groundNormal);
    if(move == Vector3.zero)
    {
        walking = false;
    }
    else
    {
        walking = true;
    }
    if (move != Vector3.zero)
    {
        // makes the character look in the direction it is walking
        transform.rotation = Quaternion.LookRotation(move);
    }

    rigidbody.MovePosition(rigidbody.position + move * speed * Time.deltaTime);
```

Figure28: The move method that is called from the player



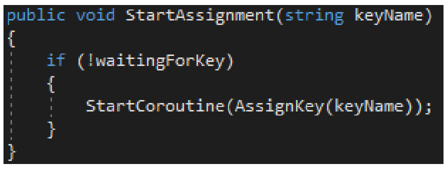
```
void ScaleCapsuleForCrouching(bool crouch)
{
    if(isGrounded && crouch)
    {
        //if (dashing)
        //{
        //    return;
        //}
        if (crouching)
        {
            capsule.height = capsuleHeight;
            capsule.center = capsuleCenter;
            crouching = false;
        }
        else
        {
            capsule.height = capsule.height / 2f;
            capsule.center = capsule.center / 2f;
            crouching = true;
        }

    }
}
```

Figure29: The method that scales the capsule down when the player is crouching

Input Manager

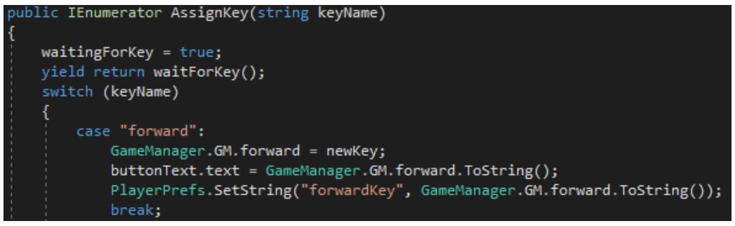
The custom input manager is using two classes, one for the GameManager, which have KeyCode properties for each of the actions e.g. forward, backward, interact etc. and that parses the KeyCode properties to the PlayerPrefs. The other is the InputsUI, which controls the inputs.

When you press a button in the inputs UI, it will call two methods in the InputsUI class. It will first call the method SendText(Text text) which will get the keys button text and set the buttonText to the text of the pressed button. Then it calls the StartAssignment(string keyName) method which starts a coroutine with AssignKey(string keyName). If waitingForKey is false, this sets waitingForKey true so you cannot start a new assignment. Then it goes to a yield that calls the method waitForKey() which will yield null because it is waiting for the user to press a key for the keyEvent. When the player have pressed a button on the keyboard the event calls the monobehaviour method OnGUI() with makes the field newKey into the keyEvent that was pressed and changes the waitingForKey to false. Then the yield stops returning null and keeps going through the code. It goes into a switch statement where it goes to the case that has the same name as the button that was pressed. When it hits the case with the same name it will make the game manager KeyCode property become the newKey, change the button text to the new buttons name, change the PlayerPrefs to the new button and then break out of the case.



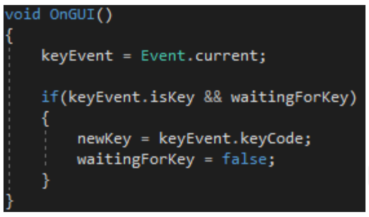
```
public void StartAssignment(string keyName)
{
    if (!waitingForKey)
    {
        StartCoroutine(AssignKey(keyName));
    }
}
```

Figure30: Implementation of StartAssignment(string keyName)



```
public IEnumerator AssignKey(string keyName)
{
    waitingForKey = true;
    yield return waitForKey();
    switch (keyName)
    {
        case "forward":
            GameManager.GM.forward = newKey;
            buttonText.text = GameManager.GM.forward.ToString();
            PlayerPrefs.SetString("forwardKey", GameManager.GM.forward.ToString());
            break;
```

Figure31: Implementation of AssignKey(satring keyName)



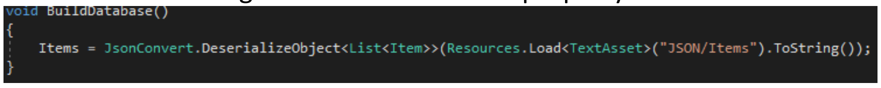
```
void OnGUI()
{
    keyEvent = Event.current;

    if(keyEvent.isKey && waitingForKey)
    {
        newKey = keyEvent.keyCode;
        waitingForKey = false;
    }
}
```

Figure32: Implementation of OnGUI()

Inventory system

The inventory system consists off an ItemDatabase, InventoryUI, JSON file that contains the items and an InventoryController.

The ItemDatabase which at start calls the BuildDatabase() method which deserialises the Items.json, makes it into a string and saves it into a list property of Items.

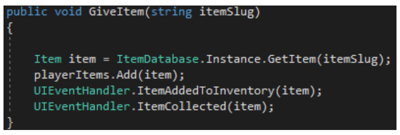


```
void BuildDatabase()
{
    Items = JsonConvert.DeserializeObject<List<Item>>(Resources.Load<TextAsset>("JSON/Items").ToString());
}
```

Figure33: Implementation of BuildDatabase()

The InventoryUI which creates an itemContainer from the resources, adds two events listeners (OnItemAddedToInventory and OnItemDeletedFromInventory) and deactivates the game object on the Start() method. When the player picks up an item or is given an item from a quest, the methods ItemAdded(Item item) and ItemDeleted(Item item) are called through the event handler whenever an item is added or deleted, it is then removed or added to the inventory UI as an itemContainer.

The InventoryController contains the methods for giving items to the players inventory, setting the items details, equipping a weapon (calls the EquipWeapon(Item itemToEquip) from the WeaponController) and consuming an item. When the



```
public void GiveItem(string itemSlug)
{

    Item item = ItemDatabase.Instance.GetItem(itemSlug);
    playerItems.Add(item);
    UIEventHandler.ItemAddedToInventory(item);
    UIEventHandler.ItemCollected(item);
}
```

Figure34: Implementation of GiveItem(string itemSlug)

either of the GiveItem() methods are called they invoke the two events ItemAddedToInventory(Item item) and ItemCollected(Item item).

UI

Pause menu: The pause menu is brought up by clicking the escape button. It then enables the PauseMenu, disables the camera, unlocks the cursor, sets the cursor to visible and pauses the game. There are 3 things you can do in the menu, resume is either done by pressing the resume button or pressing escape again. You can go into options by pressing the options button which disables the PauseMenu and enables the OptionsMenu. The last button is the quit button which quits the game.

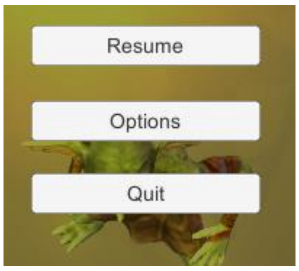
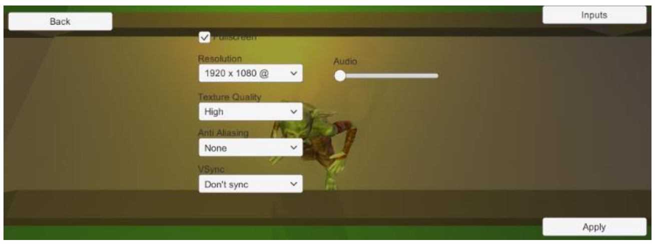


Figure35: Pause UI

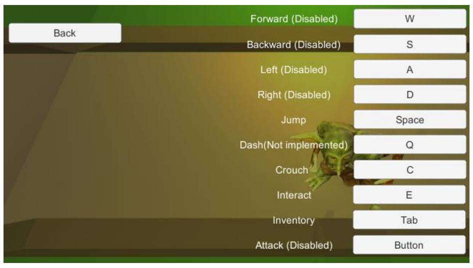
Options menu: In the OptionsMenu, you can go back to the PauseMenu, go to InputsMenu or change the options. It has two classes, one called GameSettings, which contains the fields for the settings, and the SettingManager, which controls the settings. Every time the game object is enabled it adds listeners to the buttons so when they are pressed it will call the method to change the value then it calls the LoadSettings() method which loads all the previous GameSettings to the UI from the gamesettings.json. Then when a setting is changed the delegate will call the method that it is connected to and change the Unity setting to be what the gamesettings is that is changed to be what the new setting is in the field.



[✓] Fullscreen
Resolution: 1920 x 1080 @
Texture Quality: High
Anti Aliasing: None
VSync: Don't sync

Figure36: Options UI

Inputs menu: The inputs menu is accessed from the OptionsMenu and contains the current buttons that are implemented. Some of the buttons are disabled due to the time constraint on the project and I would not have been able to do much else if I wanted to perfect the InputsMenu. To read more about the inputs go to



Forward (Disabled)	W
Backward (Disabled)	S
Left (Disabled)	A
Right (Disabled)	D
Jump	Space
Dash(Not implemented)	Q
Crouch	C
Interact	E
Inventory	Tab
Attack (Disabled)	Button

Figure37: Inputs UI

Input Manager.

Inventory: The inventory is brought up by pressing the inventory button. It contains a list of the players inventory, a field for the selected item which contains the name of the item, a description of the item, the stats of the item and the action button (name changes depending on what type of item it is) for the item, the players stats and the currently selected weapon with the stats for the weapon, an image of the weapon and an unequip button for the weapon. To read more about the inventory go to Inventory system.

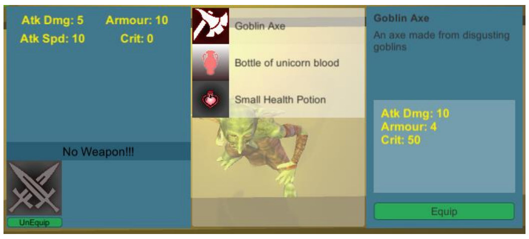


Figure38: Inventory UI

Portal menu: The portal menu is a simple menu that is always active and when the player interact with a portal it sets a panel active and fills it with buttons for each other portal that is connected to that portal. To read more about the Portal go to Teleporting.

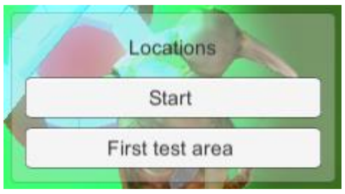


Figure39: Portals UI

Dialogue: When the player interacts with a NPC or a QuestGiver it disables the camera, player script, CanvasPause, pauses the game, enables the dialogueUI, unlocks the cursor, makes the cursor visible and makes interacting true. Then it will play the first or second dialogue depending on the conditions that are met. The reasoning behind having 2 dialogues is so after the player have talked to an NPC and they have told the player their first dialogue they should talk to the player different (like they have already met the player). When the dialogue is over it will close the dialogue window. The QuestGiver derives from the NPC so it uses the NPC's dialogue, to read more go to Quest.

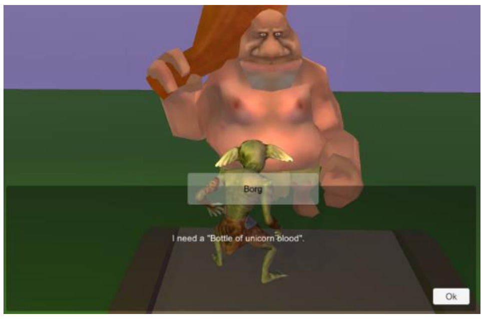
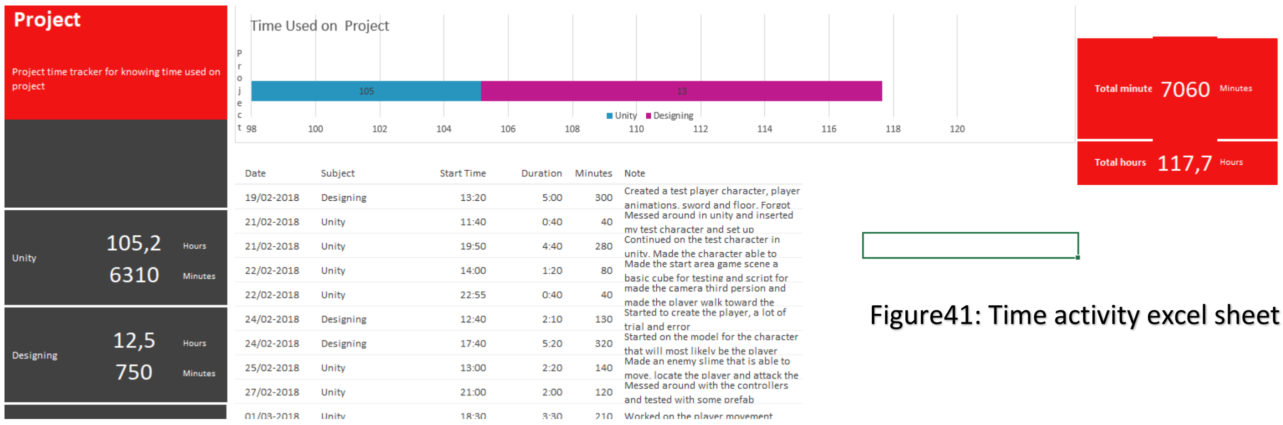


Figure40: Dialogue UI

Process

When the project first started I sat down and set up some ground rules for myself about how I was going to work and act in this project, I will work every day no matter how small (at least 15 min). I will log every time I work directly at the project and to research about games. The reason I setup the rule about working every day for at least 15 min, was so that I would set the habit of working and when I started working it usually ended with more than 15 min.



Project

Project time tracker for knowing time used on project

Unity 105,2 Hours 6310 Minutes

Designing 12,5 Hours 750 Minutes

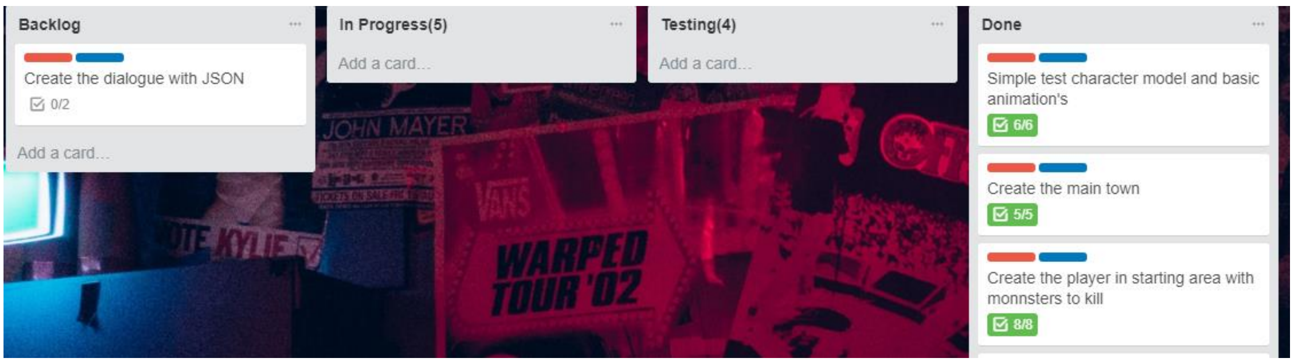
Total minute 7060 Minutes

Total hours 117,7 Hours

Date	Subject	Start Time	Duration	Minutes	Note
19/02-2018	Designing	13:20	5:00	300	Created a test player character, player animations, sword and floor. Forgot
21/02-2018	Unity	11:40	0:40	40	Messed around in unity and inserted my test character and set up
21/02-2018	Unity	19:50	4:40	280	Continued on the test character in unity. Made the character able to
22/02-2018	Unity	14:00	1:20	80	Made the start area game scene a basic cube for testing and script for
22/02-2018	Unity	22:55	0:40	40	made the camera third persion and made the player walk toward the
24/02-2018	Designing	12:40	2:10	130	Started to create the player, a lot of trial and error
24/02-2018	Designing	17:40	5:20	320	Started on the model for the character that will most likely be the player
25/02-2018	Unity	13:00	2:20	140	Made an enemy slime that is able to move, locate the player and attack the
27/02-2018	Unity	21:00	2:00	120	Messed around with the controllers and tested with some prefab
01/03-2018	Unity	18:30	3:30	210	Worked on the player movement

Figure41: Time activity excel sheet

Then I started my documentation I created an excel sheet for my project time activity where I logged all the time I worked directly on the project (not including research, learning etc.), started on my problem statement, learning goals, Kanban board and on the skeleton of the report.



Backlog	In Progress(5)	Testing(4)	Done
Create the dialogue with JSON ☑ 0/2	Add a card...	Add a card...	Simple test character model and basic animation's ☑ 6/6
Add a card...			Create the main town ☑ 5/5
			Create the player in starting area with monnsters to kill ☑ 8/8

Figure42: Kanban board

I moved then on to creating the first test player model and character controller. My reasoning behind creating a test character with basic animations was so I could figure out how I wanted the character controller to behave and to test if I could learn to use the animations with the controller.

I then proceeded to implement dialogue, combat, inventory, pause, options, inputs, UI's, and questing. Every time I started on any work, I noted the date and time, at the end I noted down how long I worked and wrote a comment that described what I did.

At the end I started to stitch the game together by creating a start area, a town and an area that was a depiction of my own test area from when I first started. I added some characters and enemies, so I could quest and combat. At this time I also added portals and polished up some of my scripts e.g. added an event for when I deleted items from my inventory for collecting quests.

When it was getting closer to the delivery of the project, I started to focus purely on my report and do some small tweaks on the game that I did not log. The process has gone well and I followed the process pretty much as I wanted to.

Conclusion

I managed to achieve all my goals in the main area, I created a decent dialogue, a functioning questing system, a decent character controller player inputs and a decent start to the story. Even though I did not make my main areas as good as I wanted to or use the technology I wanted to I managed to get them to an acceptable level.

I learned quite a lot in this project. I achieved almost all of my learning goals except character creation in a 3D modelling program. I was able to understand a lot of how to design game (still a long way to go). I got the basic understanding of how to use Unity, its features and learned a lot of how games work and how to make them work. I improved my programming skills by quite a bit by doing all the work by myself, having no one to rely on makes me go from the attitude of "if I don't do it, someone else will do it" to "if I don't do it, no one will" and this made me more happy to do the work when I don't have a safety net to rely on to do the work for me. I truly enjoyed working with environment creation. It was an exciting and new experience to think about how the player will perceive it and how to make it work in a good and easy way for the player.

In conclusion, I loved working on this project and especially doing it alone as I would do it all. I valued getting this experience a lot and in the future, I am planning to learn C++ so I can create this game in C++ from scratch and get the full experience where I create a game engine that is custom made for my game.

References

[1] ” Scrum & Waterfall: An At-a-Glance Comparison of Their Underlying Principles” [Online]. Available: http://www.innolution.com/blog/plan-driven-vs-agile-principles [Accessed may 2018]

[2] “Learning loop definition” [Online]. Available: http://www.innolution.com/resources/glossary/learning-loop [Accessed may 2018]

[3] “Batch size” [Online]. Available: http://www.innolution.com/resources/glossary/batch-size [Accessed may 2018]

[4] “Extreme programming: A gentle introduction” [Online]. Available: http://www.extremeprogramming.org/ [Accessed may 2018]

[5] “What is lean” [Online]. Available: https://www.lean.org/WhatsLean/ [Accessed may 2018]

[6] “Unity documentation” [Online]. Available https://docs.unity3d.com/Manual/index.html [Accessed may 2018]

[7] “Unity Answers” [Online]. Available https://answers.unity.com/index.html [Accessed may 2018]

[8] “Unity Homepage” [Online]. Available https://unity3d.com/unity [Accessed may 2018]

[9] “Unreal engine Homepage” [Online]. Available https://www.unrealengine.com/en-US/what-is-unreal-engine-4 [Accessed may 2018]

[10] “Unity: character controller vs rigidbody” [Online]. Available https://medium.com/ironequal/unity-character-controller-vs-rigidbody-a1e243591483 [Accessed may 2018]

[11] “Character controller” [Online]. Available https://docs.unity3d.com/ScriptReference/CharacterController.html [Accessed may 2018]