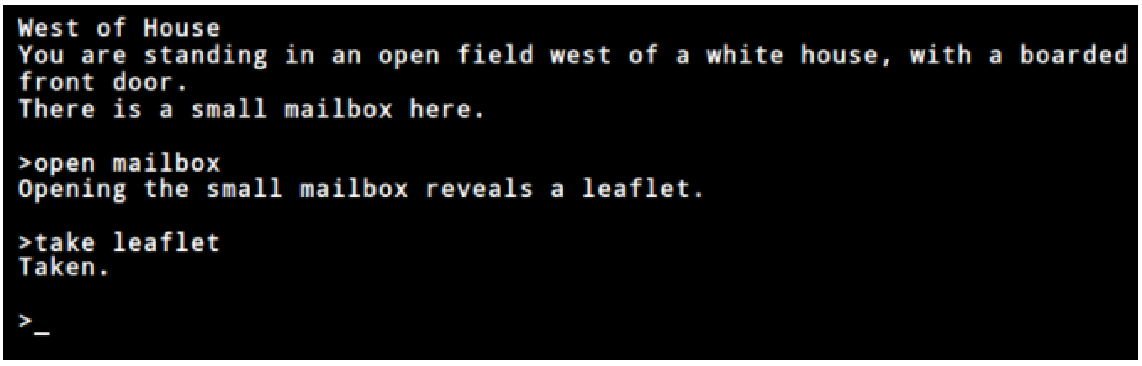
TEXT BASED RPG GAME

1.INTRODUCTION

Text-based games are complex, interactive simulations in which text describes the game state and players make progress by entering text commands.Our aim is to create a text based RPG game which can run on any platform. We would mainly use Android studio and Java to support our project. We should be able to create bug free and user friendly and interesting game where everyone can enjoy without a hitch.



```
West of House
You are standing in an open field west of a white house, with a boarded
front door.
There is a small mailbox here.

>open mailbox
Opening the small mailbox reveals a leaflet.

>take leaflet
Taken.

>_
```

Fig. 1.1:Intro to Zore

Text-based games are turn-based games usually played through a command line terminal. At each turn, several lines of text describe the state of the game, and the player may enter a text command to change this state in some desirable way (i.e., to move towards a goal). A game's built-in parser or interpreter deciphers player commands and maps them to state changes (events in the game). The genre became popular in the early 1980s especially with the release of Zork [Infocom, 1980], which featured rich storytelling and an advanced command parser.

2. LITERATURE SURVEY/REVIEW OF LITERATURE

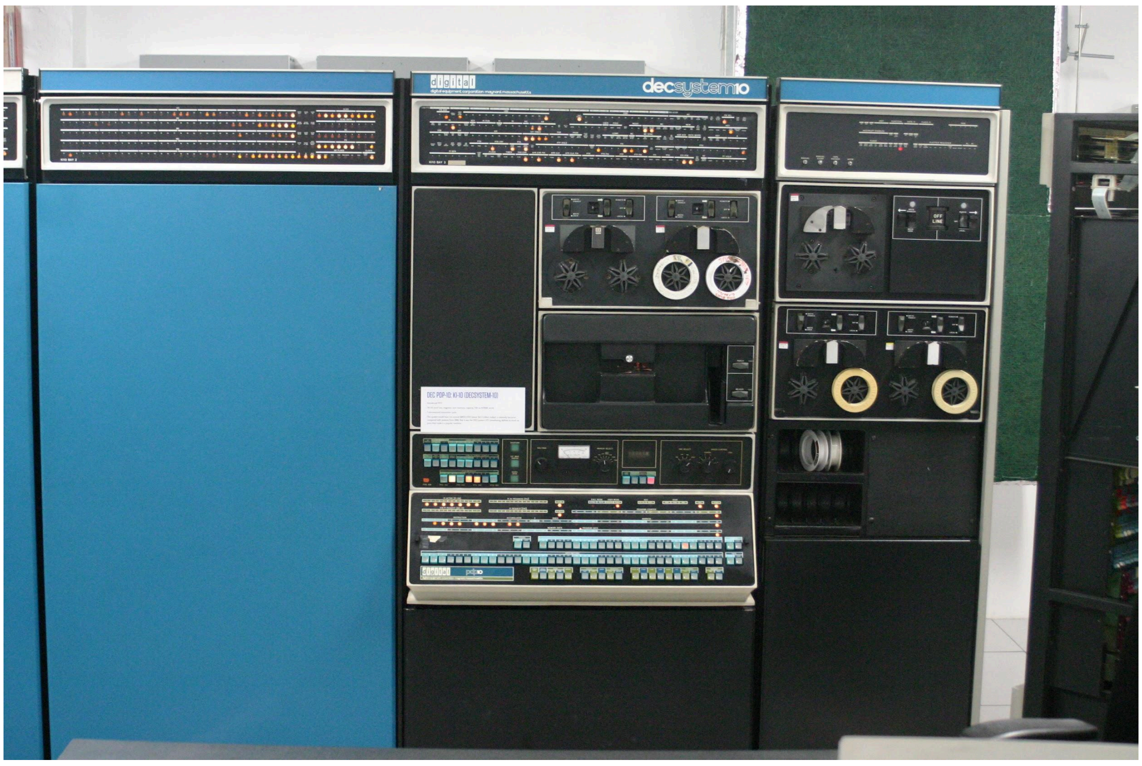
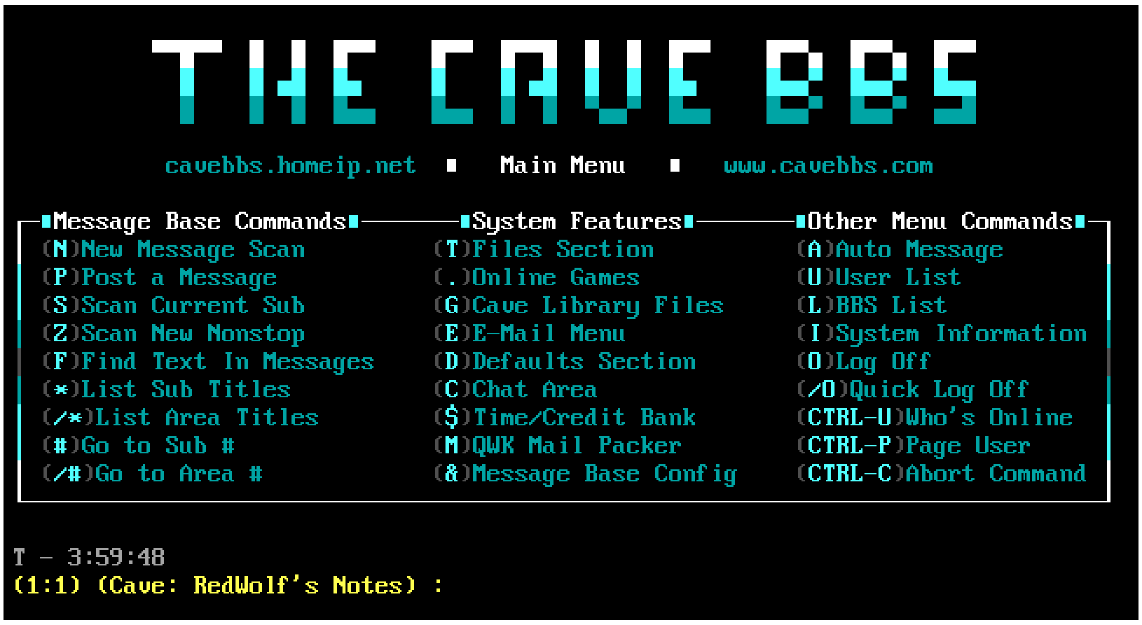


Fig. 2.1:PDP-10

Before the widespread availability of graphical displays, text adventures were one of the few game genres that owed their existence solely to computing. The first text adventure was Colossal Cave (also known simply as Adventure), written in 1976 by Will Crowther for the PDP-10 mainframe. With the advent of home computing in the late 1970s, Colossal Cave and other games such as Zork were enjoyed by many. The majority of early text adventures used a narration-action loop that accepted simple commands of the general form VERB or VERB NOUN (e.g. ‘look’ , ‘go west’ , ‘take box’) via console input. In response to such commands, the programs provided a description of the immediate environment, e.g. ‘You are in an open field on the west side of a white house with a boarded front door. There is a small mailbox here.’ Early adventures typically involved exploration and treasure hunting, but more sophisticated narratives emerged in the 1980s (e.g. in the Infocom range of games). An active “Interactive Fiction” community still exists, using powerful natural language authoring tools such as Inform [2] to create new and diverse titles.


```
                         THE CAVE BBS

     cavebbs.homeip.net  ■  Main Menu  ■  www.cavebbs.com

 ■Message Base Commands■      ■System Features■        ■Other Menu Commands■
 (N)New Message Scan          (T)Files Section          (A)Auto Message
 (P)Post a Message            (.)Online Games           (U)User List
 (S)Scan Current Sub          (G)Cave Library Files     (L)BBS List
 (Z)Scan New Nonstop          (E)E-Mail Menu            (I)System Information
 (F)Find Text In Messages     (D)Defaults Section       (O)Log Off
 (*)List Sub Titles           (C)Chat Area              (/O)Quick Log Off
 (/*)List Area Titles         ($)Time/Credit Bank       (CTRL-U)Who's Online
 (#)Go to Sub #               (M)QWK Mail Packer        (CTRL-P)Page User
 (/#)Go to Area #             (&)Message Base Config    (CTRL-C)Abort Command

T - 3:59:48
(1:1) (Cave: RedWolf's Notes) :
```

Fig. 2.2 The cave BBSes

Text-based games were also early forerunners to online gaming. From the late-1970s. until the worldwide dominance of the Internet in the mid-1990s, home computer users could still interact remotely with other computers by using dial-up modems, connecting them via telephone wires. These computers were often directed via text-based terminal emulators to hobbyist-run bulletin board systems (BBSes), which tended to be accessible—often freely—by area codes to cut costs from more distant communications. Without a graphical program for clients, most online computer games could only run using textual graphics, and where the user did have such a program, the often limited bandwidth of the modem made downloading graphics much slower than text. Online games designed for BBSes(Bulletin board system) initially used ASCII as the character set, but since the late-1980s, most BBSes employed colored ANSI art as the graphical standard. These online games became known as "BBS door games", as connecting to a BBS opened the "door" between the client and the games on the BBS.

3. SOFTWARE REQUIREMENT ANALYSIS

There are many hurdles while developing a game. There are many requirements to be met when developing a text based game. Generally a text based game is a lot less straining than a graphical game. But the need to develop a game is more challenging than any essential application in the market. As a game is played at leisure and free time, it should be less time consuming and fast to load. We have to choose an appropriate platform as well as language to develop which ticks most of our requirements.

3.1 UNDERSTANDING THE PROBLEM

Observation Modality Observations consist in the environment's textual feedback to the previous command. This means that the observation space is unbounded, consisting of arbitrary-length sequences of characters. To simplify things, we assume that an observation is made of space separated words that may or may not be found in an English dictionary. One drawback of looking only at words is that we may lose some information provided by the spacing (e.g., ASCII art in Infidel [Berlyn, 1983] or a sonar map in Seastalker [Galley and Lawrence, 1984]).

Understanding Parser Feedback Text-based games process player input using a parser. The parser varies from game to game in terms of the actions it recognizes. For example, nearly all games recognize actions like get, take and go, but only some games recognize verbs like tickle, swim, dig and bribe. Part of the challenge of playing a parser-based text game is understanding which verbs and objects are recognized by the parser. Making this task more difficult, failure messages vary from game to game when the parser receives an invalid or unrecognized command.

3.2 SOFTWARE AND LANGUAGE USED IN DEVELOPMENT

Choosing a particular software for a simple project is much harder than a complex project as there are more number of options present. So we had to go through a number of languages and platforms to make a decision. And we made:

3.2.1 UNITY

Fig. 3.1:Unity logo and 2020 version interface

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. As of 2018, the engine had been extended to support more than 25 platforms. The engine can be used to create three-dimensional, two-dimensional, virtual reality, and augmented reality games, as well as simulations and other experiences.[4][5] The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering and construction.

Unity gives users the ability to create games and experiences in both 2D and 3D, and the engine offers a primary scripting API in C#, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality. Prior to C# being the primary programming language used for the engine, it previously supported Boo, which was removed with the release of Unity 5, and a version of JavaScript called UnityScript, which was deprecated in August 2017, after the release of Unity 2017.1, in favor of C#.

Within 2D games, Unity allows importation of sprites and an advanced 2D world renderer. For 3D games, Unity allows specification of texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.

Working with Unity is simple and easy to learn for a beginner. As there are many options to choose from we choose unity because of its simplicity, customizability, and live demo options.

3.3 C#

Fig. 3.2: C Sharp logo

C# (pronounced see sharp, like the musical note C♯, but written with the number sign)[b] is a general-purpose, multi-paradigm programming language encompassing static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.[16]

C# was developed around 2000 by Microsoft as part of its .NET initiative and later approved as an international standard by Ecma (ECMA-334) in 2002 and ISO (ISO/IEC 23270) in 2003. It was designed by Anders Hejlsberg, and its development team is currently led by Mads Torgersen, being one of the programming languages designed for the Common Language Infrastructure (CLI). The most recent version is

9.0, which was released in 2020 in .NET 5.0 and included in Visual Studio 2019 version 16.8.

Mono is a free and open-source project to develop a cross-platform compiler and runtime environment (i.e. virtual machine) for the language.

We choose C# because

- The language is intended to be a simple, modern, general-purpose, object-oriented programming language.
- The language, and implementations thereof, should provide support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection. Software robustness, durability, and programmer productivity are important.
- The language is intended for use in developing software components suitable for deployment in distributed environments.
- Portability is very important for source code and programmers, especially those already familiar with C and C++.
- Support for internationalization is very important.
- C# is intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.
- Although C# applications are intended to be economical with regard to memory and processing power requirements, the language was not intended to compete directly on performance and size with C or assembly language.

4. SOFTWARE DESIGN

Designing the game with as few steps as possible was our goal, but there happens to be a lot of steps for a simple game like this.

- Write how to create a game which accepts text input and responds to it
- Write how to work with strings and some useful data collections to use with them, including Dictionary and list.
- Write how to create a flexible architecture and workflow for authoring game content using the Delegate Pattern
- Create a log of all our actions.
- Display descriptions of rooms and interactable objects.
- Accept text input from the player
- Move between Rooms which contain Exits.

4.1 THINGS TO LEARN BEFORE STARTING THE PROJECT

4.1.1 LIST

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

Does not need to have a size declared when declared. As new elements are added the list gets longer. Useful when we don't know how many objects we will need to store the type of the list must be specified using <> (angle brackets) we can create a new List of Transforms like this:

```
List <Transform> some Transforms = new List<Transform>();
```

4.1.2 SYSTEM.SERIALIZABLE

The Serializable attribute lets you embed a class with sub properties in the inspector. You can use this to display variables in the inspector similar to how a Vector3 shows up in the inspector. The name and a triangle to expand its properties. To do this you need create a class that derives from System.Object and give it the Serializable attribute.

4.1.3 DICTIONARY

- The data structure kind. Not a big heavy book.
- In computer science, an associative array, map. symbol table, or dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection
- A generic dictionary allows us to specify a type for both the key and the value, for example

Dictionary string, GameObject myDictionary

This allows us to pass in a string (key) and get back a GameObject (value)

GameObject player 1 = myDictionary Player 11.

4.1.4 INHERITANCE AND ABSTRACT CLASSES

Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes, The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class.

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property indexer, or event.

While learning and implementing these in the program we had to create object in game such as

- Room: A ScriptableObject that holds a description of the room and a set of exits.
- Exit: A serialized class which holds a description of the exit and a reference to the next room.
- RoomNavigation: Holds a reference to the current room and loads and unloads rooms as we move between them
- GameController Manages all the subsystems of the game and the actionLog, a list of strings which contains everything that has happened so far.
- TextInput: Takes input from the player displays it and calls InputActions
- InputAction. A ScriptableObject which holds code to execute when an action is needed

At the end we made a system of items which the player can interact with

- A simple inventory system
- Allow players to Take, Examine and Use objects
- Change game state in response to Use actions.

4.5 LET'S LOOK INTO SOME ACTION CLASS UML DIAGRAMS

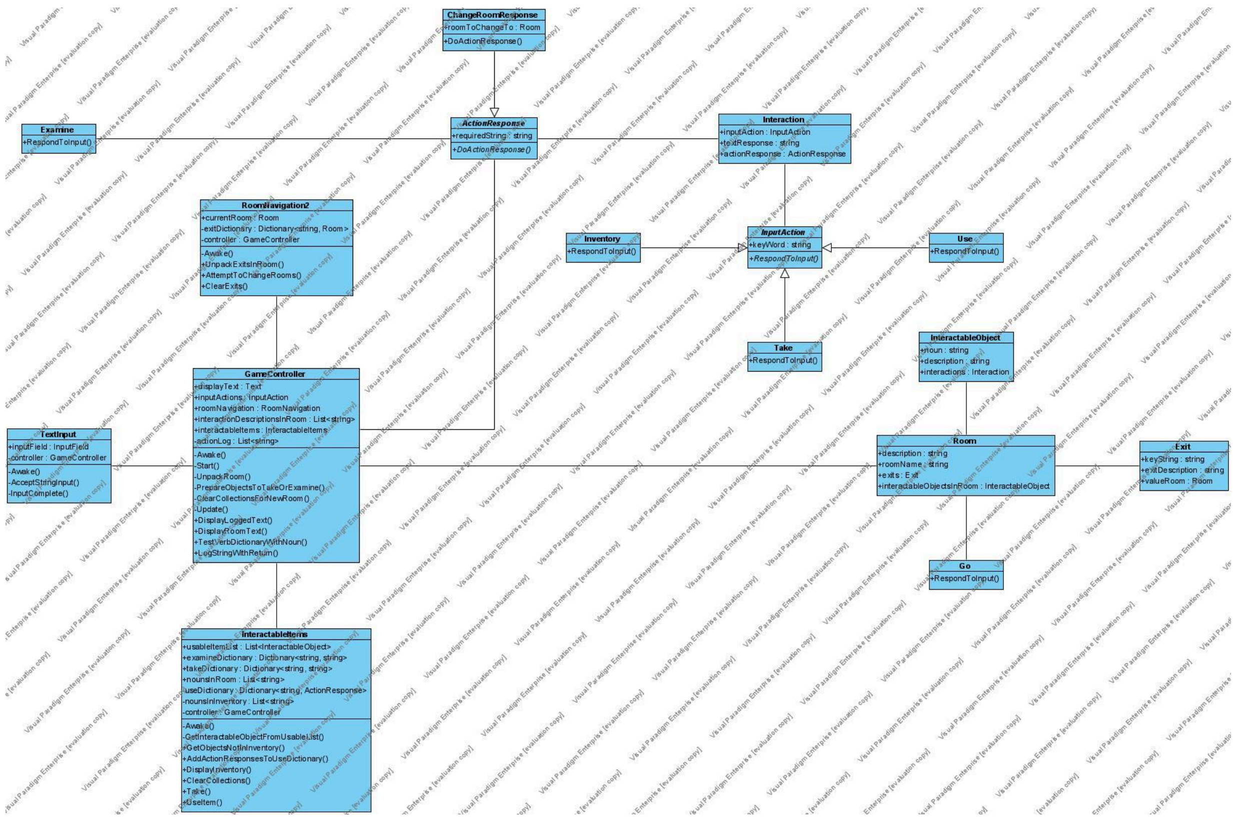

ChangeRoomResponse
+roomToChangeTo : Room
+DoActionResponse()
Examine
+RespondToInput()
ActionResponse
+requiredString : string
+DoActionResponse()
Interaction
+inputAction : InputAction
+textResponse : string
+actionResponse : ActionResponse
RoomNavigation2
+currentRoom : Room
-exitDictionary : Dictionary<string, Room>
-controller : GameController
-Awake()
+UnpackExitsInRoom()
+AttemptToChangeRooms()
+ClearExits()
Inventory
+RespondToInput()
InputAction
+keyWord : string
+RespondToInput()
Use
+RespondToInput()
Take
+RespondToInput()
InteractableObject
+noun : string
+description : string
+interactions : Interaction
GameController
+displayText : Text
+inputActions : InputAction
+roomNavigation : RoomNavigation
+interactionDescriptionsInRoom : List<string>
+interactableItems : InteractableItems
-actionLog : List<string>
-Awake()
-Start()
-UnpackRoom()
-PrepareObjectsToTakeOrExamine()
-ClearCollectionsForNewRoom()
-Update()
+DisplayLoggedText()
+DisplayRoomText()
+TestVerbDictionaryWithNoun()
+LogStringWithReturn()
TextInput
+inputField : InputField
-controller : GameController
-Awake()
-AcceptStringInput()
-InputComplete()
Room
+description : string
+roomName : string
+exits : Exit
+interactableObjectsInRoom : InteractableObject
Exit
+keyString : string
+exitDescription : string
+valueRoom : Room
Go
+RespondToInput()
InteractableItems
+usableItemList : List<InteractableObject>
+examineDictionary : Dictionary<string, string>
+takeDictionary : Dictionary<string, string>
+nounsInRoom : List<string>
-useDictionary : Dictionary<string, ActionResponse>
-nounsInInventory : List<string>
-controller : GameController
-Awake()
-GetInteractableObjectFromUsableList()
+GetObjectsNotInInventory()
+AddActionResponsesToUseDictionary()
+DisplayInventory()
+ClearCollections()
+Take()
+UseItem()

Fig. 4.1: Complete class diagram of the project

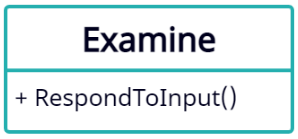


Fig. 4.2: examine class diagram

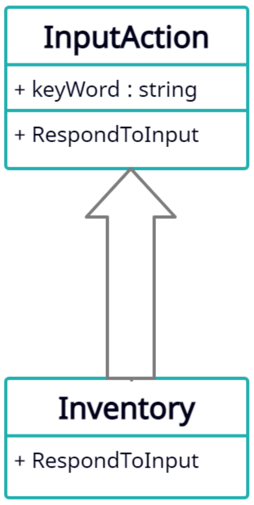


Fig. 4.3: inventory and input action relation

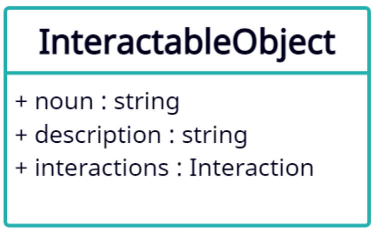


Fig. 4.4:class diagram interactable object

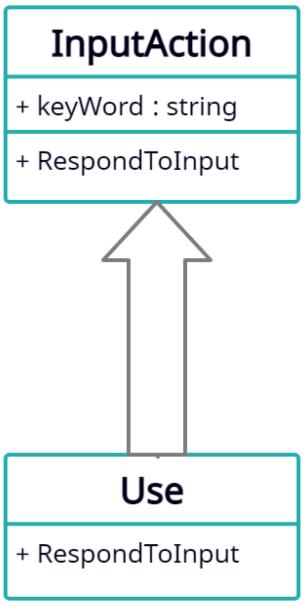


Fig. 4.5: relation between input action and use input

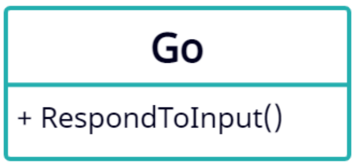


Fig. 4.6: class diagram for Go

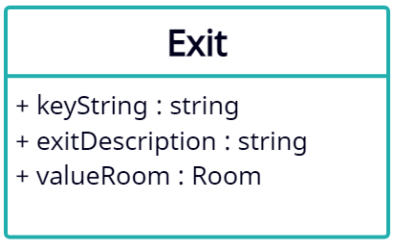


Fig. 4.7: class diagram for exit

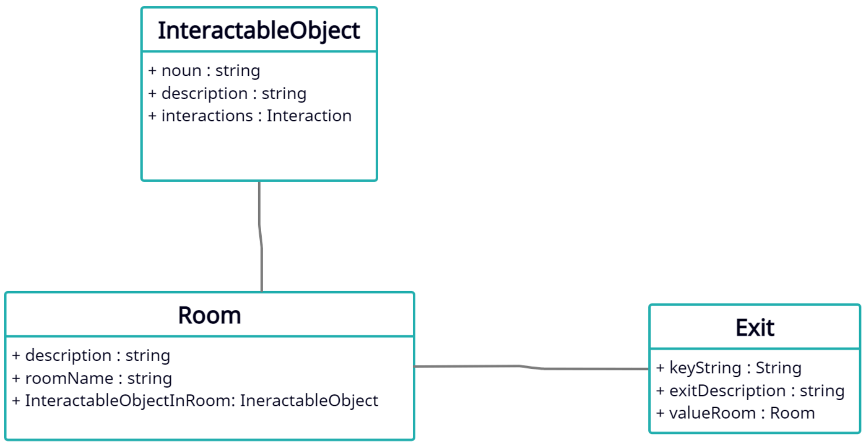


Fig. 4.8: Relation between ActionResponse, Interaction and InputAction

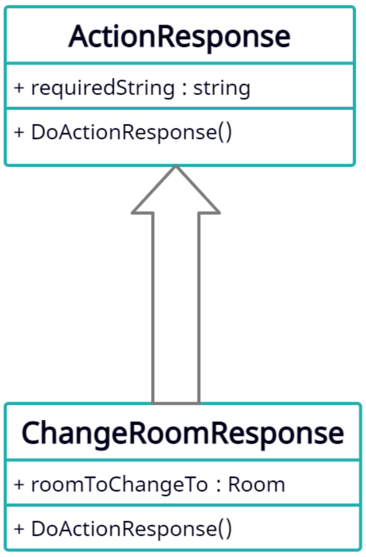


Fig. 4.9:Relation between ActionResponse and ChangeRoomResponse

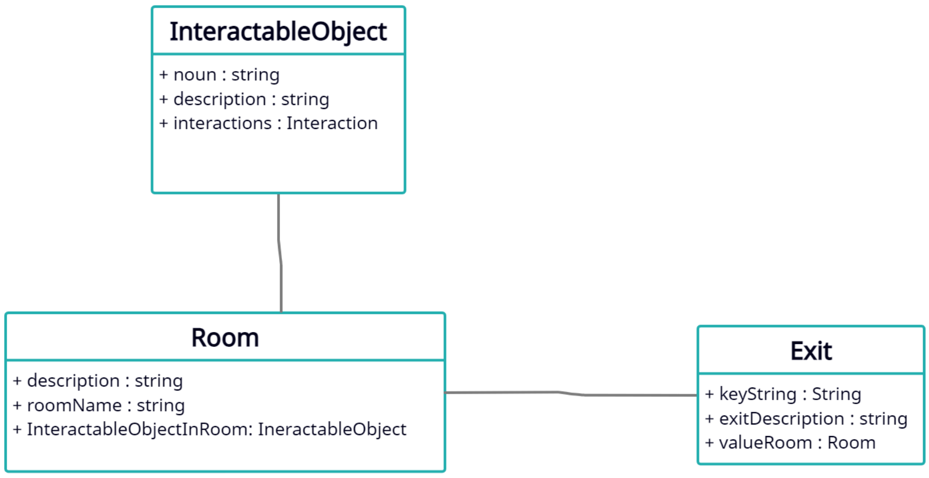


Fig. 4.10: Relation between Interactableobject, room and exit

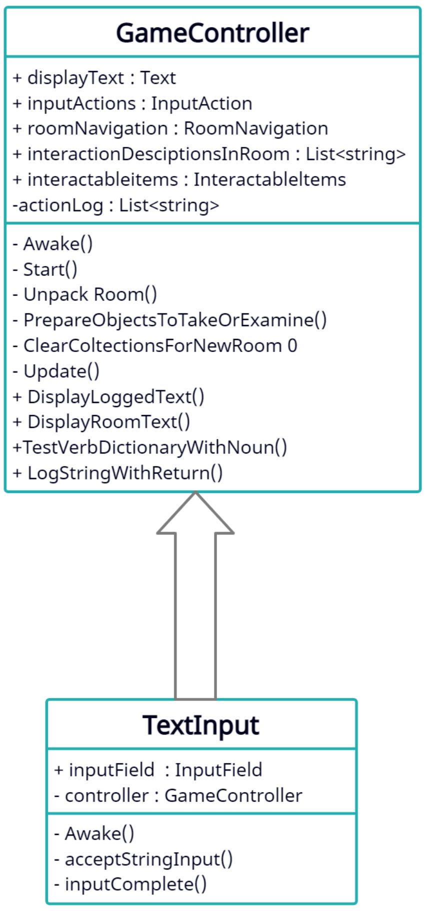

GameController
+ displayText : Text
+ inputActions : InputAction
+ roomNavigation : RoomNavigation
+ interactionDesciptionsInRoom : List<string>
+ interactableitems : InteractableItems
-actionLog : List<string>
- Awake()
- Start()
- Unpack Room()
- PrepareObjectsToTakeOrExamine()
- ClearColtectionsForNewRoom 0
- Update()
+ DisplayLoggedText()
+ DisplayRoomText()
+TestVerbDictionaryWithNoun()
+ LogStringWithReturn()
TextInput
+ inputField : InputField
- controller : GameController
- Awake()
- acceptStringInput()
- inputComplete()

Fig. 4.11: Relation between GameController and TextInput

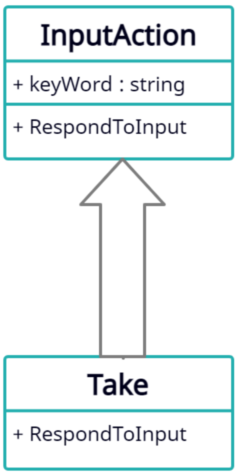


Fig. 4.12: Relation between InputAction and Take

5. SOFTWARE AND HARDWARE REQUIREMENTS

System		**Minimum requirements**
Desktop		
	Operating system	Windows 7 SP1+ macOS 10.12+ Ubuntu 16.04+
	CPU	SSE2 instruction set support.
	GPU	Graphics card with DX10 (shader model 4.0) capabilities.
iOS		iOS 9.0 or higher.
Android		OS 4.1 or later ARMv7 CPU with NEON support or Atom CPU
WebGL		Any recent desktop version of Firefox, Chrome, Edge or Safari.
Universal Windows Platform		Windows 10 and a graphics card with DX10 (shader model 4.0) capabilities. OpenGL ES 2.0 or later.

Table 5.1

6. CODING /CODE TEMPLATES

Lets see the code block by block

Room

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;


[CreateAssetMenu(menuName = "TextAdventure/Room")]

public class Room : ScriptableObject

{

   [TextArea]

   public string description;

   public string roomName;

   public Exit[] exits;

   public InteractableObject[] interactableObjectsInRoom;

}
```

GameController

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;
```

```
public class GameController : MonoBehaviour {

    public Text displayText;
    public InputAction[] inputActions;
    [HideInInspector] public RoomNavigation roomNavigation;
    [HideInInspector] public List<string> interactionDescriptionsInRoom = new
List<string> ();
    [HideInInspector] public InteractableItems interactableItems;
    List<string> actionLog = new List<string>();

    // Use this for initialization
    void Awake ()
    {
        interactableItems = GetComponent<InteractableItems> ();
        roomNavigation = GetComponent<RoomNavigation> ();
    }

    void Start()
    {
        DisplayRoomText ();
        DisplayLoggedText ();
    }

    public void DisplayLoggedText()
    {
        string logAsText = string.Join ("\n", actionLog.ToArray ());
        displayText.text = logAsText;
    }

    public void DisplayRoomText()
    {
        ClearCollectionsForNewRoom ();
        UnpackRoom ();
        string joinedInteractionDescriptions = string.Join ("\n",
interactionDescriptionsInRoom.ToArray ());
        string combinedText = roomNavigation.currentRoom.description + "\n" +
joinedInteractionDescriptions;
        LogStringWithReturn (combinedText);
    }

    void UnpackRoom()
    {
        roomNavigation.UnpackExitsInRoom ();
        PrepareObjectsToTakeOrExamine (roomNavigation.currentRoom);
    }

    void PrepareObjectsToTakeOrExamine(Room currentRoom)
    {
```

```
    for (int i = 0; i < currentRoom.interactableObjectsInRoom.Length; i++)
    {
       string descriptionNotInInventory = interactableItems.GetObjectsNotInInventory
(currentRoom, i);
       if (descriptionNotInInventory != null)
       {
          interactionDescriptionsInRoom.Add (descriptionNotInInventory);
       }
       InteractableObject interactableInRoom =
currentRoom.interactableObjectsInRoom [i];

       for (int j = 0; j < interactableInRoom.interactions.Length; j++)
       {
          Interaction    interaction = interactableInRoom.interactions [j];
          if (interaction.inputAction.keyWord == "examine")
          {
             interactableItems.examineDictionary.Add (interactableInRoom.noun,
interaction.textResponse);
          }
          if (interaction.inputAction.keyWord == "take")
          {
             interactableItems.takeDictionary.Add (interactableInRoom.noun,
interaction.textResponse);
```

```
            }
         }
      }
   }

   public string TestVerbDictionaryWithNoun(Dictionary<string, string>
verbDictionary, string verb, string noun)
   {
      if (verbDictionary.ContainsKey (noun))
      {
         return verbDictionary [noun];
      }
      return "You can't " + verb + " " + noun;
   }

   void ClearCollectionsForNewRoom()
   {
      interactableItems.ClearCollections ();
      interactionDescriptionsInRoom.Clear ();
      roomNavigation.ClearExits ();
   }
   public void LogStringWithReturn(string stringToAdd)
   {
      actionLog.Add (stringToAdd + "\n");
```

```
    }
}
```

InputAction

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class InputAction : ScriptableObject
{
    public string keyWord;
    public abstract void RespondToInput (GameController controller, string[]
separatedInputWords);
}
```

RoomNavigation

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RoomNavigation : MonoBehaviour {
    public Room currentRoom;

    Dictionary<string, Room> exitDictionary = new Dictionary<string, Room> ();

    GameController controller;

    void Awake()

    {

        controller = GetComponent<GameController> ();

    }

    public void UnpackExitsInRoom()

    {

        for (int i = 0; i < currentRoom.exits.Length; i++)

        {

            exitDictionary.Add (currentRoom.exits [i].keyString, currentRoom.exits
[i].valueRoom);

            controller.interactionDescriptionsInRoom.Add (currentRoom.exits
[i].exitDescription);

        }

    }

    public void AttemptToChangeRooms(string directionNoun)

    {

        if (exitDictionary.ContainsKey (directionNoun)) {

            currentRoom = exitDictionary [directionNoun];

            controller.LogStringWithReturn ("You head off to the " + directionNoun);

            controller.DisplayRoomText ();

        }
        else
        {
            controller.LogStringWithReturn ("There is no path to the " + directionNoun);
        }
    }
    public void ClearExits()
    {
        exitDictionary.Clear ();
    }
}
```

Exit

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]
public class Exit
{
    public string keyString;
    public string exitDescription;
    public Room valueRoom;
}
```

TextInput

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class TextInput : MonoBehaviour
{
   public InputField inputField;
   GameController controller;
   void Awake()
   {
      controller = GetComponent<GameController> ();
      inputField.onEndEdit.AddListener (AcceptStringInput);
   }
   void AcceptStringInput(string userInput)
   {
      userInput = userInput.ToLower ();
      controller.LogStringWithReturn (userInput);
      char[] delimiterCharacters = { ' ' };
      string[] separatedInputWords = userInput.Split (delimiterCharacters);
      for (int i = 0; i < controller.inputActions.Length; i++)
      {
         InputAction inputAction = controller.inputActions [i];
```

```
            if (inputAction.keyWord == separatedInputWords [0])
            {
                inputAction.RespondToInput (controller, separatedInputWords);
            }
        }
        InputComplete ();
    }

    void InputComplete()
    {
        controller.DisplayLoggedText ();
        inputField.ActivateInputField ();
        inputField.text = null;
    }
}
```

InputAction

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class InputAction : ScriptableObject
{
    public string keyWord;
    public abstract void RespondToInput (GameController controller, string[]
separatedInputWords);
}
```

Go

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName= "TextAdventure/InputActions/Go")]
public class Go : InputAction
{
    public override void RespondToInput (GameController controller, string[]
separatedInputWords)
    {
        controller.roomNavigation.AttemptToChangeRooms (separatedInputWords [1]);
    }
}
```

InteractableObject

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
[CreateAssetMenu (menuName = "TextAdventure/Interactable Object")]
public class InteractableObject : ScriptableObject
{
    public string noun = "name";
    [TextArea]
    public string description = "Description in room";
    public Interaction[] interactions;


}
```

Interaction

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]
public class Interaction
{
    public InputAction inputAction;
    [TextArea]
    public string textResponse;
    public ActionResponse actionResponse;
}
```

InteractableItems

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;


public class InteractableItems : MonoBehaviour

{

   public List<InteractableObject> usableItemList;

   public Dictionary<string, string> examineDictionary = new Dictionary<string,
string> ();

   public Dictionary<string, string> takeDictionary = new Dictionary<string, string> ();

   [HideInInspector] public List<string> nounsInRoom = new List<string>();

   Dictionary <string, ActionResponse> useDictionary = new Dictionary<string,
ActionResponse> ();

   List<string> nounsInInventory = new List<string>();

   GameController controller;


   void Awake()

   {

      controller = GetComponent<GameController> ();

   }
```

```
    public string GetObjectsNotInInventory(Room currentRoom, int i)
    {
        InteractableObject interactableInRoom = currentRoom.interactableObjectsInRoom
[i];

        if (!nounsInInventory.Contains (interactableInRoom.noun))
        {
            nounsInRoom.Add (interactableInRoom.noun);
            return interactableInRoom.description;
        }
        return null;
    }

    public void AddActionResponsesToUseDictionary()
    {
        for (int i = 0; i < nounsInInventory.Count; i++)
        {
            string noun = nounsInInventory [i];
            InteractableObject interactableObjectInInventory =
GetInteractableObjectFromUsableList (noun);
            if (interactableObjectInInventory == null)
                continue;
            for (int j = 0; j < interactableObjectInInventory.interactions.Length; j++)
            {
```

```
            Interaction interaction = interactableObjectInInventory.interactions [j];

            if (interaction.actionResponse == null)
                continue;

            if (!useDictionary.ContainsKey (noun))
            {
                useDictionary.Add (noun, interaction.actionResponse);
            }
        }
    }
}

InteractableObject GetInteractableObjectFromUsableList(string noun)
{
    for (int i = 0; i < usableItemList.Count; i++)
    {
        if (usableItemList [i].noun == noun)
        {
            return usableItemList [i];
        }
    }
    return null;
}
```

```
public void DisplayInventory()
{
   controller.LogStringWithReturn ("You look in your backpack, inside you have: ");
   for (int i = 0; i < nounsInInventory.Count; i++)
   {
      controller.LogStringWithReturn (nounsInInventory [i]);
   }
}

public void ClearCollections()
{
   examineDictionary.Clear();
   takeDictionary.Clear ();
   nounsInRoom.Clear();
}

public Dictionary<string, string> Take (string[] separatedInputWords)
{
   string noun = separatedInputWords [1];

   if (nounsInRoom.Contains (noun)) {
      nounsInInventory.Add (noun);
      AddActionResponsesToUseDictionary ();
```

```
            nounsInRoom.Remove (noun);
            return takeDictionary;
        }
        else
        {
            controller.LogStringWithReturn ("There is no " + noun + " here to take.");
            return null;
        }
    }

    public void UseItem(string[] separatedInputWords)
    {
        string nounToUse = separatedInputWords [1];

        if (nounsInInventory.Contains (nounToUse)) {
            if (useDictionary.ContainsKey (nounToUse)) {
                bool actionResult = useDictionary [nounToUse].DoActionResponse
(controller);
                if (!actionResult) {
                    controller.LogStringWithReturn ("Hmm. Nothing happens.");
                }
            } else {
                controller.LogStringWithReturn ("You can't use the " + nounToUse);
            }
        } else
        {
            controller.LogStringWithReturn ("There is no " + nounToUse + " in your
inventory to use");
        }
    }
}
```

Examine

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "TextAdventure/InputActions/Examine")]
public class Examine : InputAction
{
    public override void RespondToInput (GameController controller, string[]
separatedInputWords)
    {
        controller.LogStringWithReturn (controller.TestVerbDictionaryWithNoun
(controller.interactableItems.examineDictionary, separatedInputWords [0],
separatedInputWords [1]));
    }
}
```

Take

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "TextAdventure/InputActions/Take")]
public class Take : InputAction
{
  public override void RespondToInput (GameController controller, string[]
separatedInputWords)
  {
    Dictionary<string, string> takeDictionary = controller.interactableItems.Take
(separatedInputWords);

    if (takeDictionary != null)
    {
      controller.LogStringWithReturn (controller.TestVerbDictionaryWithNoun
(takeDictionary, separatedInputWords [0], separatedInputWords [1]));
    }
  }
}
```

Inventory

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "TextAdventure/InputActions/Inventory")]
public class Inventory : InputAction
{
    public override void RespondToInput (GameController controller, string[]
separatedInputWords)
    {
        controller.interactableItems.DisplayInventory ();
    }
}
```

ActionResponse

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class ActionResponse : ScriptableObject
{
    public string requiredString;
    public abstract bool DoActionResponse(GameController controller);

}
```

ChangeRoomResponse

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "TextAdventure/ActionResponses/ChangeRoom")]
public class ChangeRoomResponse : ActionResponse
{
    public Room roomToChangeTo;

    public override bool DoActionResponse (GameController controller)
    {
        if (controller.roomNavigation.currentRoom.roomName == requiredString)
        {
            controller.roomNavigation.currentRoom = roomToChangeTo;
            controller.DisplayRoomText ();
            return true;
        }

        return false;
    }
}
```

Use

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu (menuName = "TextAdventure/InputActions/Use")]
public class Use : InputAction
{
    public override void RespondToInput (GameController controller, string[]
separatedInputWords)
    {
        controller.interactableItems.UseItem (separatedInputWords);
    }
}
```

7. TESTING

7.1 TYPES OF GAME TESTING

Game testing, a subset of game development, is a software testing process for quality control of video games. The primary function of game testing is the discovery and documentation of software defects (aka bugs). Interactive entertainment software testing is a highly technical field requiring computing expertise, analytic competence, critical evaluation skills, and endurance. In recent years the field of game testing has come under fire for being extremely strenuous and unrewarding, both financially and emotionally.

7.1.1 FUNCTIONAL TESTING

Functionality QA testers look for the generic problems within the game or its user interface & graphics, such as game mechanic issues, stability issues, and game asset integrity. User interface testing ensures user-friendliness of the game

Example: Checking colors and backgrounds, menu structure, screen orientation and screen resolution, font size, alignment errors, usability, system navigation such as loading time, timeout and display, sorting, confirmation messages, sequences, animations and audio elements aspects of the game, instructions, and dialogue messages. User Interactions, User Interfaces, Transactions testing, Calibration and accuracy testing of mobile phone cameras, Screen resolutions, Mobile responsive design testing, Audio quality Testing

7.1.2 COMPATIBILITY TESTING

Checking if the game is compatible across different devices, and on different configurations of hardware and software.

Example: Install and Uninstall the game on all supported consoles/desktops/mobiles.

7.1.3 PERFORMANCE TESTING

The overall performance of the Game is checked. Performance tuning is performed to optimize game speed.

Importance parameters checked during performance testing

Response time on client and servers, Transaction completion time(s), Peak load performance, Longevity, network coverage, Memory leakage, low memory, low battery, Time taken to download applications, simultaneous(Multiple users) access to application's server, speed, throughput, reliability, scalability, etc.

Battery Consumption and graphics performance: Measure the battery consumption of the mobile game. Battery Consumption must be optimum over long hours, and game responses should be satisfactory under varying heavy loads across different devices

Processor and memory constraints: Performance counters are used to measure the CPU and memory consumption of the application.

Network connectivity: Measures the response time of the mobile games on different network types (Wi-Fi, 2G, 3G, 4G), It gives an overall insight into how well the game will perform on unreliable networks. It also checks connectivity between

mobile devices, data centers or the cloud. The whole Peak Times, Jittery Connections, Duplication of Data, Packet loss, Fragmentation of Data are monitored.
Testing mobile games performance especially MMO

7.1.4 CONFORMANCE /COMPLIANCE TESTING

Marketplace guidelines compliance (e.g., Apple App Store policies), Enterprise policy compliance (e.g., prohibited content. Compliance may also refer to regulatory bodies such as PEGI and ESRB. The game targets a particular content rating. If there is an objectionable content that is inappropriate for the desired rating, then they are identified and reported. Even a single violation in submission for license approval may have the game rejected, incurring additional costs in further testing and resubmission.

Example: If the game is to be published in European countries, test for PAL conversion if the game is produced for Northern America, test for NTSC conversions.

7.1.5 LOCALIZATION TESTING

Localization testing becomes essential when a game is targeted for the global markets. Game titles, content, and texts need to be translated and tested with devices in multiple languages. These types of tests can be performed quickly (with the help of cloud-based device access and test automation).

Example: Localization needs specific to MENA region (Middle East/North Africa), Arabic localization(Right-to-Left text support, Bi-Directional displays), Pseudo-localization testing, double-byte characters (for East Asian languages), local time/date, currency, address formats, and other local requirements.

7.1.6 SOAK TESTING

This game automation testing involves leaving the game running for a prolonged period in various modes of operation. For example, idling paused, or at the title screen. Soaking can identify memory leaks or rounding errors.

Example: Game has begun, and the character is made to stand idle for 24 hours. This technique is used to detect crashes brought on by memory leaks and other faults in the game engine.

7.1.7 RECOVERY TESTING

In software, recovery testing checks how well the application can be recovered from crashes, hardware failures, and other similar failures. The application is forced to fail, and later it will be observed how it recovers from the failure conditions and the environment.

Example: While a gaming application is running, suddenly restart the gaming console, & check the validate the data integrity

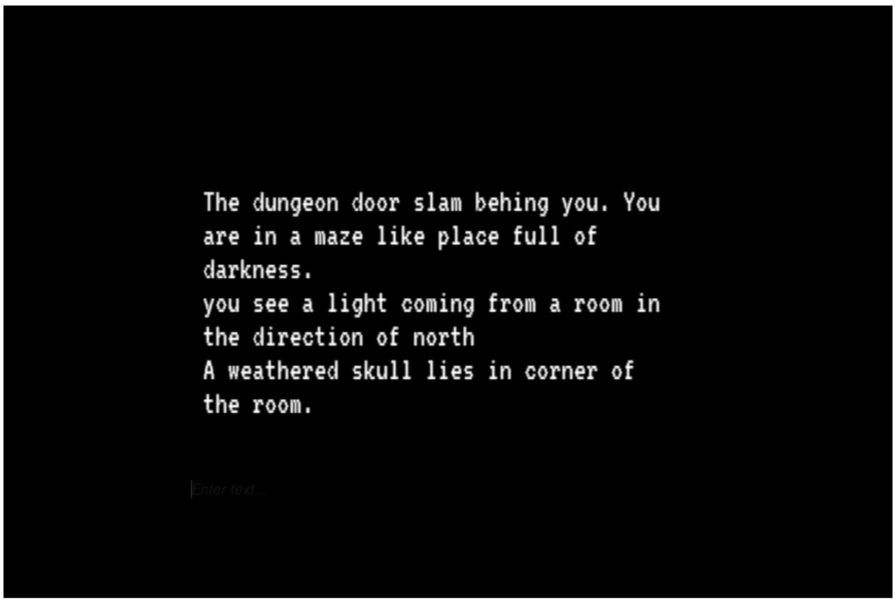
7.1.8 SECURITY TESTING

It is done to check how safe the software works from external threats. Data protection from external threats, uncontrolled system access restrictions, data breach, operating system flaws, communication system flaws and weak encryption algorithms. Example: Changing a URL from /login to /play on a gaming site should not allow direct access to the games.

Testing in unity is very simple. We can do black box testing and white box testing at the same time. There is a feature in Unity platform where we can see live progress of the game being developed and tested at the same time.

The real challenge is when exporting it to the required file format. Unity gives many options to export to, from PS5 to a website, we can export to any format required. Necessary changes should be made for correct orientation of the black board present to us.

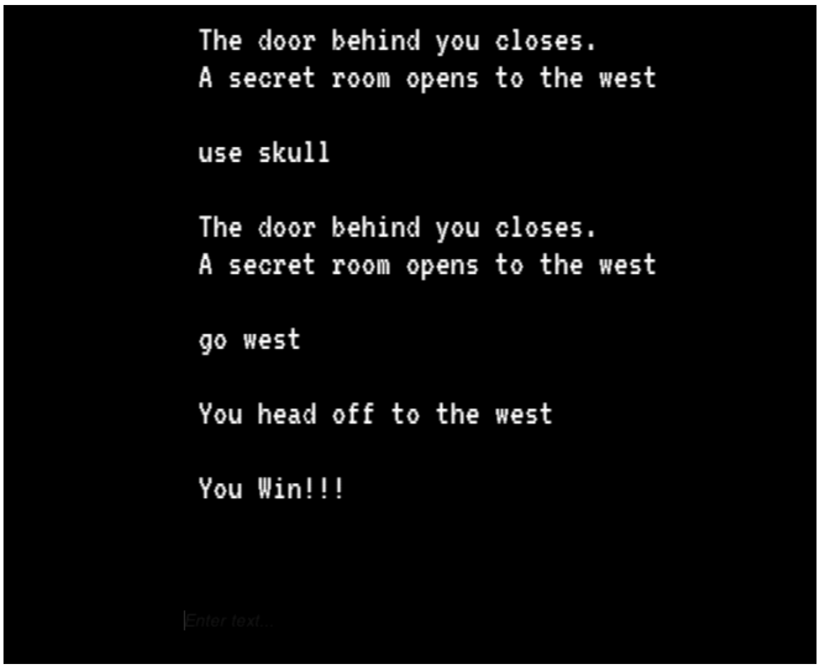
8. Output Screens



```
The dungeon door slam behing you. You
are in a maze like place full of
darkness.
you see a light coming from a room in
the direction of north
A weathered skull lies in corner of
the room.

Enter text...
```

Fig. 8.1: starting page



```
The door behind you closes.
A secret room opens to the west

use skull

The door behind you closes.
A secret room opens to the west

go west

You head off to the west

You Win!!!


Enter text...
```

Fig. 8.2:Ending page

9. CONCLUSION

We wanted it to be more interactable and give the story more thought. As it is a game there is always scope for improvement, there is no game with bugs or glitches present in development so far and hence we would like this to be a good experience. Recently text based games have become very popular as they are creative in making. We get novel based feelings while playing such but we are present as a character in the game. There is more creativity than logic present in the game, how well we can use screen to create your own story. There are many areas that need to be improved in the game such as gameplay and event handling but we feel that this game has enough potential to become a full fledged game with simple algorithms and non-complex programming style. With this as the base we can improve the game according to the programmers needs without hassle. As this was the first programming project, there were a lot of difficulties we faced but with enough patience and persistence we have achieved a base model that we feel is right for our mini-project.

REFERENCES

1. https://learn.unity.com/tutorial/recorded-video-session-text-adventure-game-part-1#5c7f8528edbc2a002053b601
2. https://learn.unity.com/tutorial/recorded-video-session-text-adventure-game-part-2#5c7f8528edbc2a002053b63b
3. https://arxiv.org/pdf/1806.11532.pdf
4. https://www.researchgate.net/publication/2935572_Knowledge-Gathering_Agents_in_Adventure_Games
5. https://arxiv.org/pdf/1812.01628v2.pdf
6. https://en.wikipedia.org/wiki/Text-based_game
7. https://pdf.sciencedirectassets.com/271600
8. https://arxiv.org/pdf/1808.01262.pdf