Introduction

Work in progress, coming soon.

Filter by title

Getting Started

Clone the repo

Clone the repo to the local machine.

```
git clone https://github.com/benpollarduk/adventure-framework.git
```

Hello World

```
// create the player. this is the character the user plays as
var player = new PlayableCharacter("Dave", "A young boy on a quest to find the meaning of life.");

/// create region maker. the region maker simplifies creating in game regions. a region contains a series of rooms
var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of town.")
{
    // add a room to the region at position x 0, y 0, z 0
    [0, 0, 0] = new Room("Cavern", "A dark cavern set in to the base of the mountain.")
};

// create overworld maker. the overworld maker simplifies creating in game overworlds. an overworld contains a series or regions
var overworldMaker = new OverworldMaker("Daves World", "An ancient kingdom.", regionMaker);

// create the callback for generating new instances of the game
// - the title of the game
// - an introduction to the game, displayed at the start
// - about the game, displayed on the about screen
// - a callback that provides a new instance of the games overworld
// - a callback that provides a new instance of the player
// - a callback that determines if the game is complete, checked every cycle of the game
// - a callback that determines if it's game over, checked every cycle of the game
var gameCreator = Game.Create(
    "The Life Of Dave",
    "Dave awakes to find himself in a cavern...",
    "A very low budget adventure.",
    x => overworldMaker.Make(),
    () => player,
    x => EndCheckResul.NotEnded,
    x => EndCheckResult.NotEnded);

// begin the execution of the game
Game.Execute(gameCreator);
```

Example game

The quickest way to start getting to grips with the structure of BP.AdventureFramework is by taking a look at the examples. An example game is provided in the BP.AdventureFramework.Examples (https://github.com/benpollarduk/adventure-framework/tree/main/BP.AdventureFramework.Examples) directory and have been designed with the aim of showcasing the various features.

Running the examples

The example applications can be used to execute the example BP.AdventureFramework game and demonstrate the core principals of the framework. Set the **BP.AdventureFramweork.Examples** project as the start up project and build and run to start the application.

Introduction

Work in progress, coming soon.

Filter by title

Overworld

Overview

An Overworld is the top level location in a game. A game can only contain a single Overworld. An Overworld can contain multiple Regions.

```
Overworld
├── Region
│   ├── Room
│   ├── Room
│   ├── Room
├── Region
│   ├── Room
│   ├── Room
```

Use

An Overworld can be simply instantiated with a name and description.

```
var overworld = new Overworld("Name", "Description.");
```

Regions can be added to the Overworld with the **AddRegion** method.

```
overworld.AddRegion(region);
```

Regions can be removed from an Overworld with the **RemoveRegion** method.

```
overworld.RemoveRegion(region);
```

The Overworld can be traversed with the **Move** method.

```
overworld.Move(region);
```

OverworldMaker

The OverworldMaker simplifies the creation of the Overworld, when used in conjunction with RegionMakers.

```
var overworldMaker = new OverworldMaker("Name", "Description.", regionMakers);
```

However, the main benefit of using an OverworldMaker is that it allows multiple instances of an Overworld to be created from a single definition of an Overworld.

```
var overworld = overworldMaker.Make();;
```

Region

Overview

A Region is the intermediate level location in a game. An Overworld can contain multiple Regions. A Region can contain multiple Rooms.

```
Overworld
├── Region
│   ├── Room
│   ├── Room
│   ├── Room
├── Region
│   ├── Room
│   ├── Room
```

A Region represents a 3D space.

- The **x** location always refers to the horizontal axis, with lower values being west and higher values being east.
- The **y** location always refers to the vertical axis, with lower values being north and higher values being south.
- The **z** location always refers to the depth axis, with lower values being down and higher values being up.

Use

A Region can be simply instantiated with a name and description.

```
var region = new Region("Name", "Description.");
```

Rooms can be added to the Region with the **AddRoom** method. The x, y and z location within the Region must be specified.

```
region.AddRoom(room, 0, 0, 0);
```

Rooms can be removed from a Region with the **RemoveRoom** method.

```
region.RemoveRoom(room);
```

The Region can be traversed with the **Move** method.

```
region.Move(Direction.North);
```

The Region can be traversed with the **Move** method.

```
region.Move(Direction.North);
```

The start position, that is the position that the Player will start in when entering a Region, can be specified with **SetStartPosition**.

```
region.SetStartPosition(0, 0, 0);
```

The **UnlockDoorPair** method can be used to unlock an **Exit** in the current Room, which will also unlock the corresponding Exit in the adjoining **Room**.

```
region.UnlockDoorPair(Direction.East);
```

Like all Examinable objects, Regions can be assigned custom commands.

```
region.Commands =
[
    new CustomCommand(new CommandHelp("Warp", "Warp to the start."), true, (game, args) =>
    {
        region.JumpToRoom(0, 0, 0);
        return new Reaction(ReactionResult.OK, "You warped to the start.");
    })
];
```

RegionMaker

The RegionMaker simplifies the creation of a Region. Rooms are added to the Region with a specified **x**, **y** and **z** position within the Region.

```
var regionMaker = new RegionMaker("Region", "Description.")
{
    [0, 0, 0] = new Room("Room 1", "Description of room 1."),
    [1, 0, 0] = new Room("Room 2", "Description of room 2."),
};
```

The main benefit of using a RegionMaker is that it allows multiple instances of a Region to be created from a single definition of a Region.

```
var region = regionMaker.Make();
```

Room

Overview

A Room is the lowest level location in a game. A Region can contain multiple Rooms.

```
Overworld
├── Region
│   ├── Room
│   ├── Room
│   ├── Room
├── Region
│   ├── Room
│   ├── Room
```

A Room can contain up to six Exits, one for each of the directions **north**, **east**, **south**, **west**, **up** and **down**.

Use

A Region can be simply instantiated with a name and description.

```
var room = new Room("Name", "Description.");
```

Exits can be added to the Room with the **AddExit** method.

```
room.AddExit(new Exit(Direction.East));
```

Exits can be removed from a Room with the **RemoveExit** method.

```
region.RemoveExit(exit);
```

Items can be added to the Room with the **AddItem** method.

```
room.AddItem(new Item("Name", "Description."));
```

Items can be removed from a Room with the **RemoveItem** method.

```
region.RemoveItem(item);
```

Characters can be added to the Room with the **AddCharacter** method.

```
room.AddCharacter(new Character("Name", "Description."));
```

Characters can be removed from a Room with the **RemoveCharacter** method.

```
region.RemoveCharacter(character);
```

Rooms can contains custom commands that allow the user to directly interact with the Room.

```
room.Commands =
[
    new CustomCommand(new CommandHelp("Pull lever", "Pull the lever."), true, (game, args) =>
    {
        room.FindExit(Direction.East, true, out var exit);
        exit.Unlock();
        return new Reaction(ReactionResult.OK, "The exit was unlocked.");
    })
];
```

Exit

Overview

An Exit is essentially a connector bewtween to adjoining rooms.

Use

An Exit can be simply instantiated with a direction.

```
var exit = new Exit(Direction.North);
```

An Exit can be hidden from the player by setting its **IsPlayerVisible** property to false, this can be set in the constructor.

```
var exit = new Exit(Direction.North, false);
```

Or set explicitly:

```
exit.IsPlayerVisible = false;
```

Optionally, a description of the Exit can be specified.

```
var exit = new Exit(Direction.North, true, new Description("A door covered in iv
y."));
```

This will be returned if the player examines the Exit.

Like all Examinable objects, an Exit can be assigned custom commands.

```
exit.Commands =
[
    new CustomCommand(new CommandHelp("Shove", "Shove the door."), true, (game, arg
s) =>
    {
        exit.Unlock();
        return new Reaction(ReactionResult.OK, "The door swung open.");
    })
];
```

Item

Overview

Items can be used to add interactivity with a game. Items can be something that a player can take with them, or they may be static in a Room.

Use

An Item can be simply instantiated with a name and description.

```
var sword = new Item("Sword", "A heroes sword.");
```

By default an Item is not takeable and is tied to a Room. If it is takeable this can be specified in the constructor.

```
var sword = new Item("Sword", "A heroes sword.", true);
```

An Item can morph into another Item. This is useful in situations where the Item changes state. Morphing is invoked with the **Morph** method. The Item that Morph is invoked on takes on the properties of the Item being morphed into.

```
var brokenSword = new Item("Broken Sword", "A broken sword");
sword.Morph(brokenSword);
```

Like all Examinable objects, an Item can be assigned custom commands.

```
bomb.Commands =
[
    new CustomCommand(new CommandHelp("Cut wire", "Cut the red wire."), true, (game, args) =>
    {
        game.Player.Kill();
        return new Reaction(ReactionResult.Fatal, "Boom!");
    })
];
```

Interaction

Interactions can be set up between different assets in the game. The **InteractionResult** contains the result of the interaction, and allows the game to react to the interaction.

```
var dartsBoard = new Item("Darts board", "A darts board.");

var dart = new Item("Dart", "A dart")
{
    Interaction = item =>
    {
        if (item == dartsBoard)
            return new InteractionResult(InteractionEffect.SelfContained, item, "The
dart stuck in the darts board.");

        return new InteractionResult(InteractionEffect.NoEffect, item);
    }
};
```

PlayableCharacter

Overview

A PlayableCharacter represents the character that the player plays as throughout the game. Each game has only a single PlayableCharacter.

Use

A PlayableCharacter can be simply instantiated with a name and description.

```
var player = new PlayableCharacter("Ben", "A 39 year old man.");
```

A PlayableCharacter can be also be instantiated with a list of Items.

```
var player = new PlayableCharacter("Ben", "A 39 year old man.",
[
    new Item("Guitar", "A PRS Custom 22, in whale blue, of course."),
    new Item("Wallet", "An empty wallet, of course.")
]);
```

A PlayableCharacter can be given items with the **AcquireItem** method.

```
player.AcquireItem(new Item("Mallet", "A large mallet."));
```

A PlayableCharactr can loose an item with the **DequireItem** method.

```
player.DequireItem(mallet);
```

A PlayableCharacter can use an item on another asset:

```
var trapDoor = new Exit(Direction.Down);
var mallet = new Item("Mallet", "A large mallet.");
player.UseItem(mallet, trapDoor);
```

A PlayableCharacter cn give an item to a non-playable character.

```
var goblin = new NonPlayableCharacter("Goblin", "A vile goblin.");
var daisy = new Item("Daisy", "A beautiful daisy that is sure to cheer up even the m
ost miserable creature.");
player.Give(daisy, goblin);
```

PlayableCharacters can contains custom commands that allow the user to directly interact with the character or other assets.

```
player.Commands =
[
    new CustomCommand(new CommandHelp("Punch wall", "Punch the wall."), true, (game,
args) =>
    {
        return new Reaction(ReactionResult.OK, "You punched the wall.");
    })
];
```

NonPlayableCharacter

Overview

A NonPlayableCharacter represents any character that the player may meet throughout the game.

Use

A NonPlayableCharacter can be simply instantiated with a name and description.

```
var goblin = new NonPlayableCharacter("Goblin", "A vile goblin.");
```

A NonPlayableCharacter can give an item to another NonPlayableCharacter.

```
var daisy = new Item("Daisy", "A beautiful daisy that is sure to cheer up even the m
ost miserable creature.");
npc.Give(daisy, goblin);
```

NonPlayableCharacters can contains custom commands that allow the user to directly interact with the character or other assets.

```
goblin.Commands =
[
    new CustomCommand(new CommandHelp("Smile", "Crack a smile."), true, (game, args)
=>
    {
        return new Reaction(ReactionResult.OK, "Well that felt weird.");
    })
];
```

Conversations

A NonPlayableCharacter can hold a conversation with the player.

- A Conversation contains **Paragraphs**.
- A Paragraph can contain one or more **Responses**.
- A **Response** can contain a delta to shift the conversation by, which will cause the conversation to jump parargraphs by the specified value.
- A **Response** can also contain a callback to perform some action when the player selects that option.

```
goblin.Conversation = new Conversation(
    new Paragraph("This is a the first line."),
    new Paragraph("This is a question.")
    {
        Responses =
        [
            new Response("This is the first response." 1),
            new Response("This is the second response.", 2),
            new Response("This is the third response.", 2)
        ]
    },
    new Paragraph("You picked first response, return to start of conversation.", -
2),
    new Paragraph("You picked second response, return to start of conversation., -
2),
    new Paragraph("You picked third response, you are dead., game => game.Player.Kil
l())
);
```

Global Commands

Overview

There are three main types of Command.

- **Game Commands** are used to interact with the game.
- **Global Commands** are used to interact with the program running the game.
- **Custom Commands** allow developers to add custom commands to the game without having to worry about extended the games interpreters.

Game Commands

Drop

Allows players to drop an item. **R** can be used as a shortcut.

```
drop sword
```

The player can also drop **all** items.

```
drop all
```

Examine

Allows players to examine any asset. **X** can be used as a shortcut.

Examine will examine the current room.

```
examine
```

The player themselves can be examined with **me** or the players name.

```
examine me
```

or

```
examine ben
```

The same is true for Regions, Overworlds, Items and Exits.

Take

Allows the player to take an Item. **T** can be used as a shortcut.

```
take sword
```

Take **all** allows the player to take all takeables Items in the current Room.

```
take all
```

Talk

Talk allows the player to start a conversation with a NonPlayableCharacter. **L** can be used as a shortcut.

If only a single NonPlayableCharacter is in the current Room no argurment needs to be specified.

```
talk
```

However if the current Room contains two or more NonPlayableCharacters then **to** and the NonPlayableCharacters name must be specified.

```
talk to dave
```

Use

Use allows the player to use the Items that the player has or that are in the current Room.

```
use sword
```

Items can be used on the Player, the Room, an Exit, a NonPlayableCharacter or another Item. The target must be specified with the **on** keyword.

```
use sword on me
```

Or

```
use sword on bush
```

Move

Regions are traversed with direction commands.

- **North** or **N** moves north.
- **East** or **E** moves east.
- **South** or **S** moves south.
- **West** or **W** moves west.
- **Down** or **D** moves down.
- **Up** or **U** moves up.

End

Only valid during a conversation with a NonPlayableCharacter, the End command will end the conversation.

```
end
```

Global Commands

About

Displays the a screen containing information about the game.

```
about
```

CommandsOn / CommandsOff

Toggles the display of the contextual commands on the screen on and off.

```
commandson
```

Or

```
commandsoff
```

Exit

Exit the current game.

```
exit
```

Help

Displays a Help screen listing all available commands.

```
help
```

KeyOn / KeyOff

Toggles the display of the map key on and off.

```
keyon
```

Or

```
keyoff
```

Map

Displays the Region map screen.

```
map
```

New

Starts a new game.

```
new.
```

Custom Commands

Custom commands can be added to many of the assets, including Room, PlayableCharacter, NonPlayableCharacter, Item and Exit. For more informations see their pages.

End Conditions

Overview

Work in progress, coming soon.

Conditional Descriptions

Overview

Normally assets are assigned a **Description** during the constructor. This is what is returned when the asset is examined.

Descriptions are usually specified as a string.

```
var item = new Item("The items name", "The items description.");
```

They can also be specified as a **Desciption**.

```
var item = new Item(new Identifier("The items name"), new Description("The items des
cription."));
```

However, sometimes it may be desirable to have a conditional description that can change based on the state of the asset.

Conditional descriptions can be specified with **ConditionalDescription** and contain a lambda which determines which one of two strings are returned when the asset is examined.

```
// the player, just for demo purposes
var player = new PlayableCharacter("Ben", "A man.");

// the description to use when the condition is true
var trueString = "A gleaming sword, owned by Ben.";

// the string to use when the condition is false
var falseString = "A gleaming sword, without an owner.";

// a lambda that determines which string is returned
Condition condition = () => player.FindItem("Sword", out _);

// the conditional description itself
var conditionalDescrption = new ConditionalDescription(trueString, falseString, cond
ition);

// create the item with the conditional description
var sword = new Item(new Identifier("Sword"), conditionalDescrption);
```

Introduction

Work in progress, coming soon.

Filter by title