Technical Design Document

Case Closed

Flinn Fraher

Table of Contents

Introduction:

Project Name
Point & Click Adventure Framework

Project Genre
Point & Click Puzzle/Adventure

Project Key Goal/s
To create a robust toolset for the creation of a point & click adventure game, with an accompanying vertical slice/demo project to assist in usage.

Project Key Challenges and Risks
- Dialogue system – creating this using UE4's behaviour tree system as an FSM.
- Interaction system

Software and Hardware Required
- Unreal Engine 4.26.X
- Blender
    - Jim Kroovy's Mr Mannequin Tools – Animation pose creation
- Adobe's Mixamo

Platforms:

The chosen target platform for this framework will be PC, although the framework could easily be adapted to work with touch controls, and possibly mobile platforms if the environment/art assets are of a suitable quality level.

Minimum Platform Specification / Recommended Specification

The minimum and recommended platform requirements for a build of the vertical slice are shown below. Note that these requirements will vary in other titles made using the framework.

Minimum Requirements

Intel Core i3 2.4GHz or AMD Ryzen 3 3200

4GB RAM

PACKAGED SIZE HERE GB of free space

Windows 10

GPU with DX11 support or higher

Recommended Requirements

Intel Core i5 2.8GHz or AMD Ryzen 5 3500

8GB RAM

PACKAGED SIZE HERE GB of free space

Nvidia GeForce GTX 670/ AMD Radeon HD 7870 or equivalent GPU with 2GB of dedicated VRAM or higher, DX 11 support or higher

Windows 10 64-bit

Engine Specifics Specifications and Limits

Storage / Disk Budget

The base size of the framework's uproject on disk is 15GB.

Texture Compression

To ensure asset materials take up minimal storage space, any texture which is not utilised by UMG components in the framework will have Unreal's texture compression applied by default

Engine Summary:

Engine Version

The engine version used will be the official release of Unreal Engine 4.26.2

Plugins / Middleware

The framework will not require the use of any third party plugins, however should a designer decide to add additional plugins during their own development, the base framework will not pose any conflict issues.

Game Systems and Diagrams

Game Mechanics

Camera Movement

The camera during gameplay is determined by which room the player is currently in – these cameras can be placed by the designer in-engine and connected to specific rooms in the level which are represented by geometry the designer can customise. The camera switches when the player leaves the current camera region and moves to a different area – this switch takes place smoothly by interpolating between the two camera's locations.

Player Movement

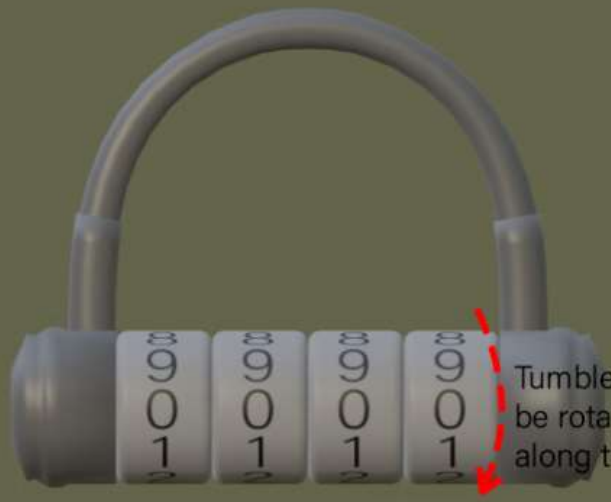
Player movement is achieved by left clicking on a walkable surface in the game world. The player character then moves to the chosen location using Unreal's built-in pathfinding solution. If an interactable object is clicked instead, depending on the object the player may still move to the clicked location, before then playing an animation of picking the object up or interacting with it, however these events are handled by the interaction system.

Puzzles

The base framework has three core puzzles – a combination lock, an electrical grid-based puzzle, and a 'slider' puzzle.

Combination Lock

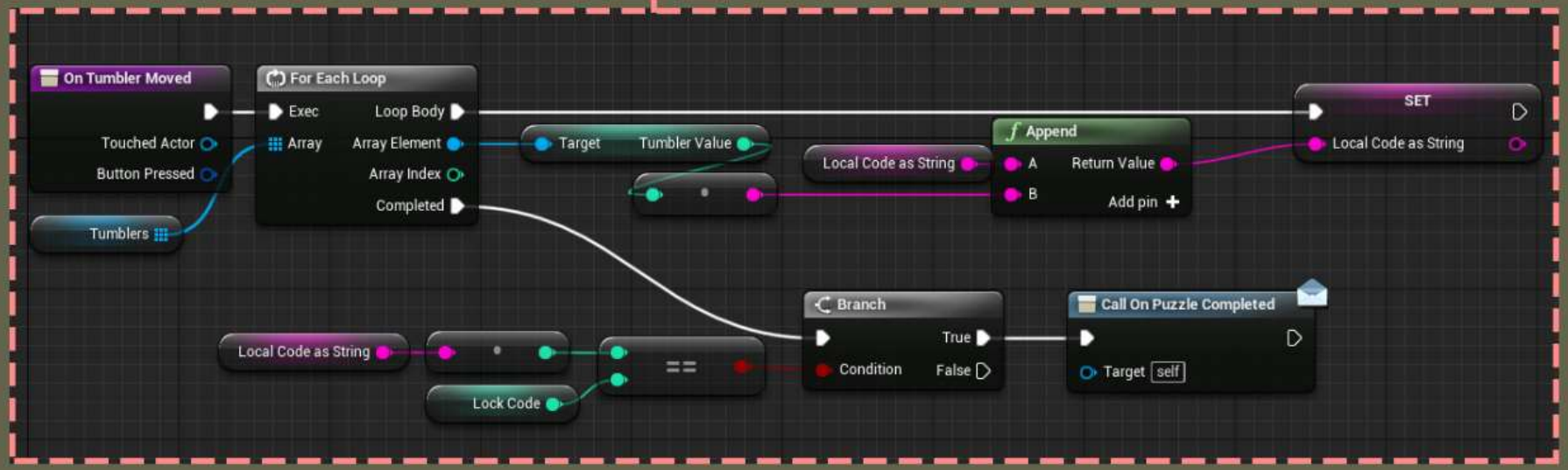
The combination lock utilises two main actors in-game – a lock base, which derives from the BasePuzzle actor and uses event dispatchers to communicate the current state of the puzzle to other gameplay objects, and the puzzle tumblers, which handle communication between the core puzzle and the player. The player can rotate each tumbler of the lock clockwise to the next digit by clicking it – when the tumblers are in an orientation which equates to the puzzles correct combination, the lock is opened.



When a tumbler is rotated, the base lock recalculates the current combination:

- (local Y rotation/36) - 1
  eg. Y rotation 72°, corresponding value is 1.

Combine whole combination into single integer, check if equal to the set code.
If it's equal, call the PuzzleSolved event dispatcher

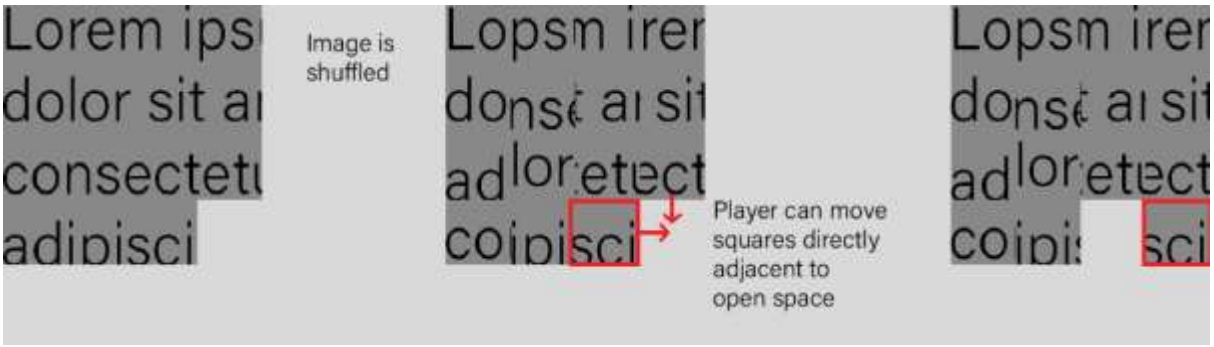


Slider Puzzle

Like the grid-based puzzles, slider puzzles task the player with manipulating shuffled cut-out squares of an image back to the original image's form. By clicking on a square with a free space next to it, the player can move a section of the image to the free space, allowing the movement of some adjacent pieces. Eventually through these movements, the orientation of each square will correspond to that section's position in the original image.

Figure X shows a representation of an abstracted version of this puzzle type. The '15 puzzle' requires a player to move each numbered square into a state such that the numbers count from the top left-hand side to 15 in order.

This core idea is identical to the slider puzzle I intend to implement, only instead of ordering numbers, squares need to be orientated to view the final image they collectively make (see Fig. Y)



Inventory System and Items

The framework's inventory system utilises a series of data tables, with each entry having a unique ID to differentiate each item in the game. The framework supports a selection of different object types:

- Puzzle items – items which are required to complete a specific puzzle in the game – these items can be linked to actors in the game and be used to gate completion of a puzzle before obtaining this item and combining it with its associated puzzle actor.
- Conversational items – items which can be selected and used on NPCs to trigger new conversation options when speaking to them. These items can also be used to sequentially progress gameplay objectives depending on the dialogue options they allow access to. For instance, an NPC might ask the player to find an item they want to progress – the player can then find that item and give it to them which subsequently completes a gameplay objective
- Combinational items – items which can be combined with others to create a unique item. This is arguably more of a sub-category, as the item created from combining two of these could be any of the other categories listed here.
- Misc. items – items which don't serve a particular purpose, but might provide hints to the player on how to progress when interacted with/inspected

Dialogue

The dialogue system makes use of Unreal's finite state machine system inside of AI behaviour trees. Each segment of dialogue is represented as a single task inside the behaviour tree and can have either one or many connected nodes depending on the dialogue prompt's type.

The types are as follows:

- **Standard dialogue** – statement said by any character in the game. Connects to either a **standard dialogue** node, or a **dialogue choice**
- **Dialogue choice** –
    - **Standard choice** – a statement which is followed by a selection of responses the player can choose between. Each choice can connect to any other dialogue node
    - **Conditional choice** – a dialogue response which can only be seen/selected if the player has met a certain condition. The condition is driven by a behaviour tree decorator which can check if specific gameplay parameters have been met. A designer can make children of this base to check if different world events have been met and attach them to relevant dialogue tasks.
- **End conversation** – a node which ends the current conversation and returns the player to normal gameplay.
- **World Task Nodes** – nodes which don't display any dialogue but cause an in-game event to occur. A designer can make a child task of this and implement custom behaviour if they desire. This task will have a reference to the player character, the NPC currently in dialogue, and the game's objective manager.

Conditional nodes, and gameplay-driven logic

performing checks inside the conversation will be done using the standard methods inside behaviour trees – selectors and decorators that allow the dialogue to be driven by player, actor references and objective state.

Ambient Dialogue

Ambient dialogue can also take place during gameplay – this is dialogue that the player character or other NPC's can say without triggering the dialogue interface. This will be used to convey information when interacting with environment objects, or to display hints when interacting with different items in a manner like other games in this genre (Grim Fandango, Twelve Minutes, Harvester, etc.)

Example Dialogue Tree

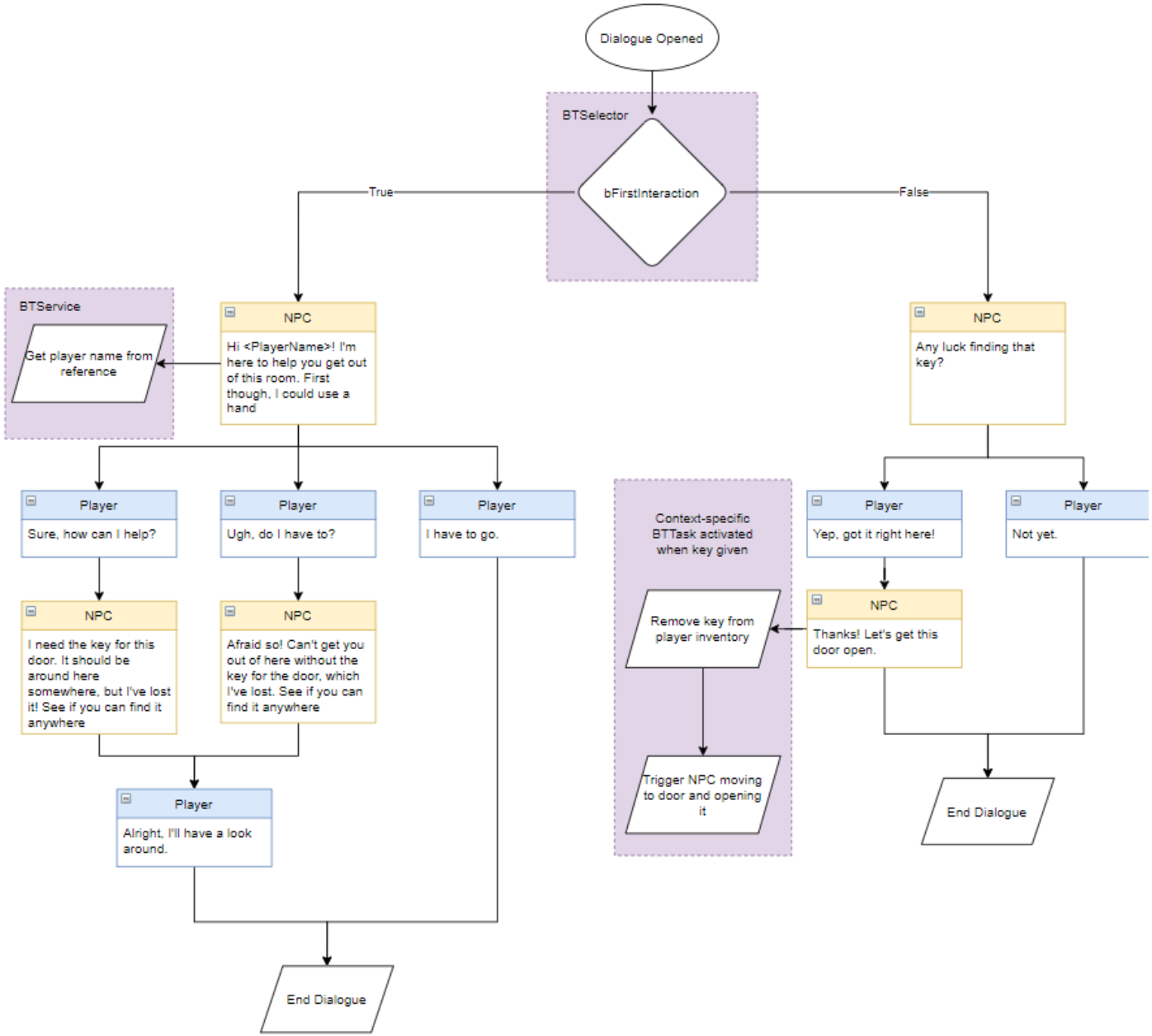
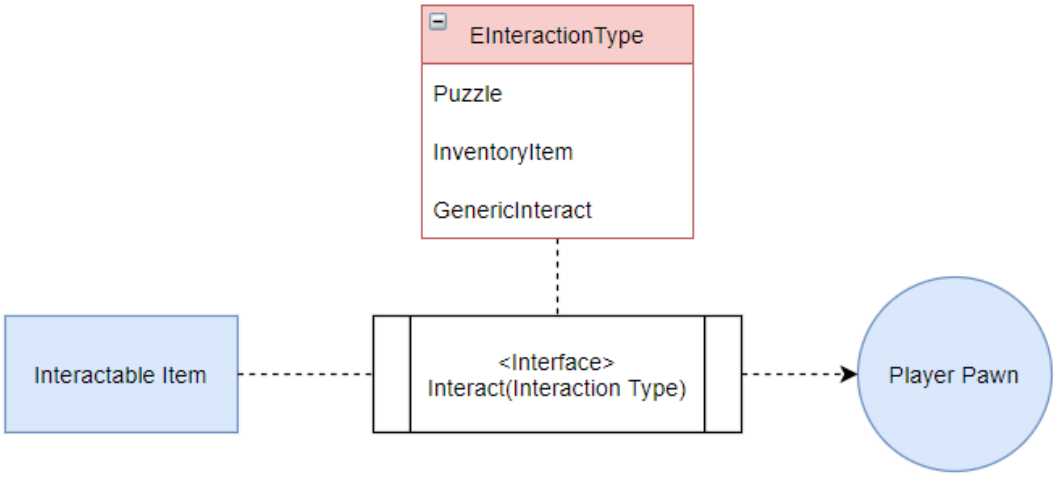


Figure 1 - An example of a basic dialogue tree. Yellow nodes are NPC dialogue, blue is player dialogue, and purple are custom BT nodes which are context-specific

Environmental Interaction

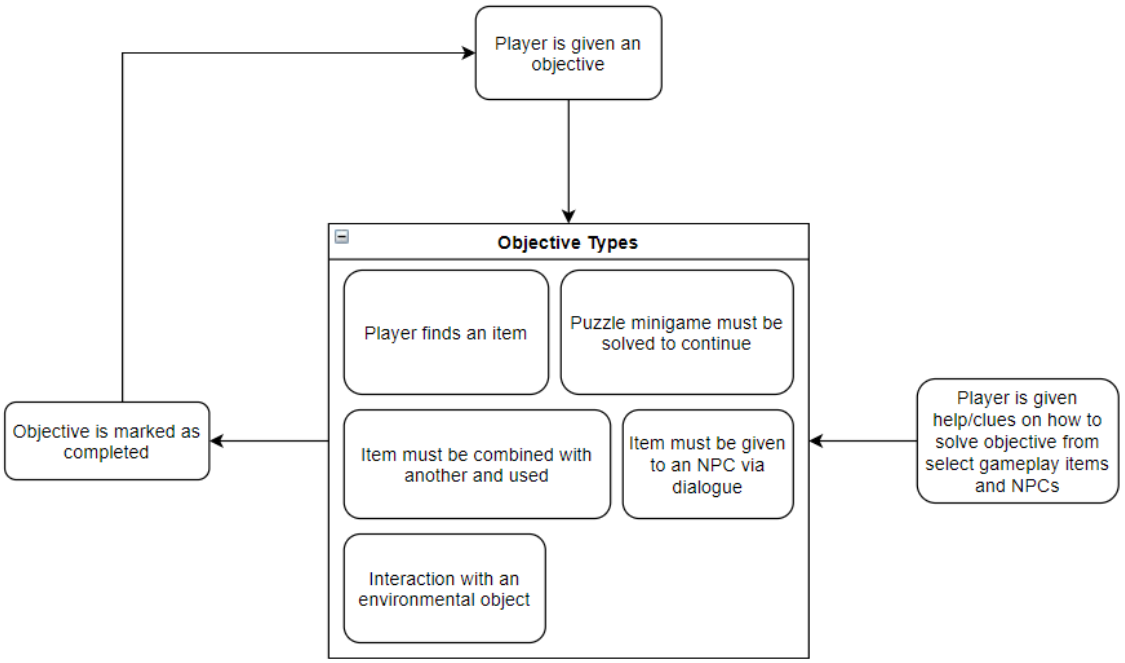
Interaction with in-game objects will be handled through one of the core hierarchies of the framework – using an interface which is implemented on the player controller, an interactable object can call an OnInteract event when the player is within a minimum specified distance, and the player clicks on the object. When this is called, the interaction sequence for the specified object takes place – this varies per object, and can be implemented on each actor which derives from the BaseInteractable C++ class.

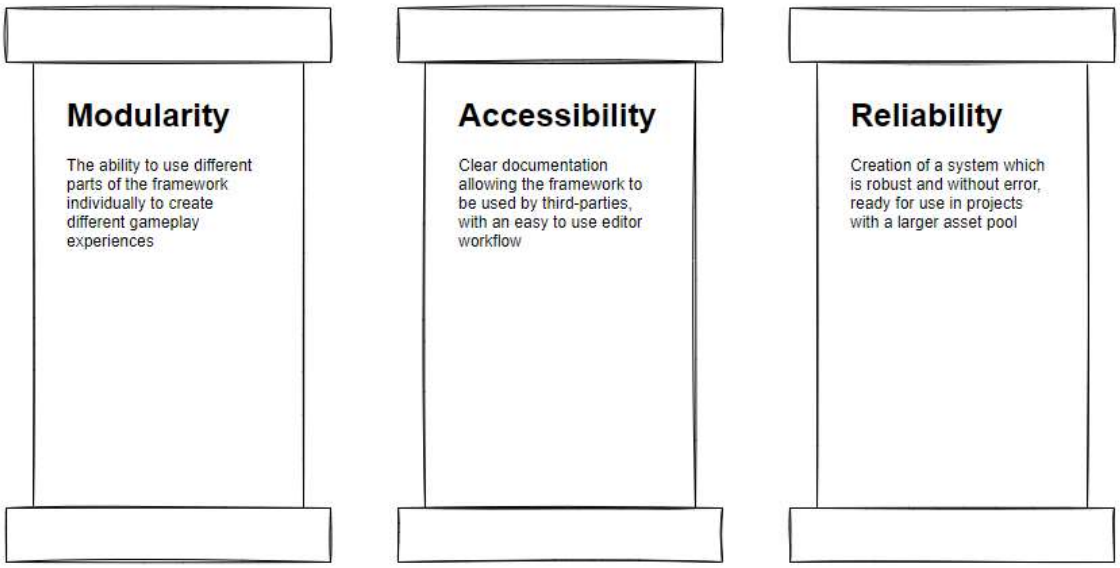
You should reference key gameplay features and mechanics.

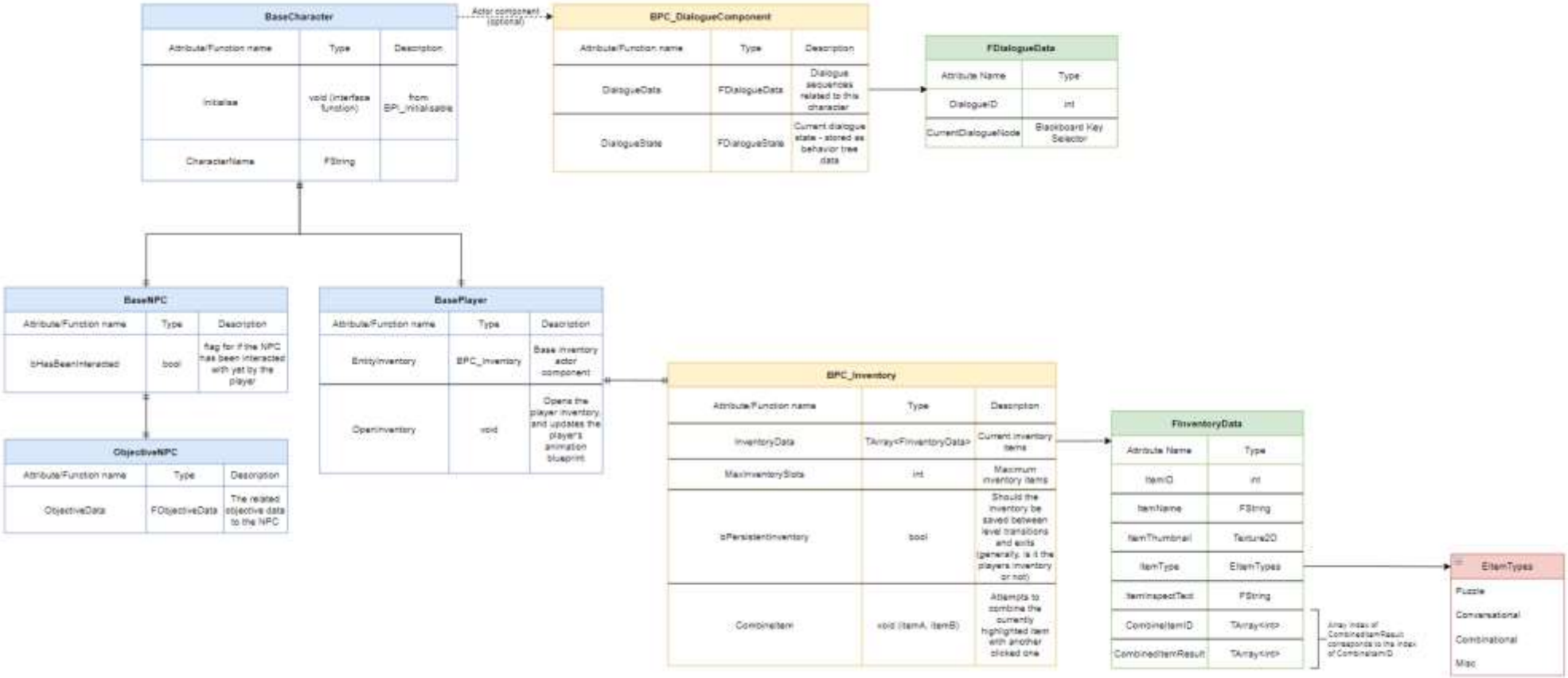


Each mechanic should have a brief description of what it is and how it works.
Each mechanic shoulder reference game engine variables so that the design team can reuse / balance.

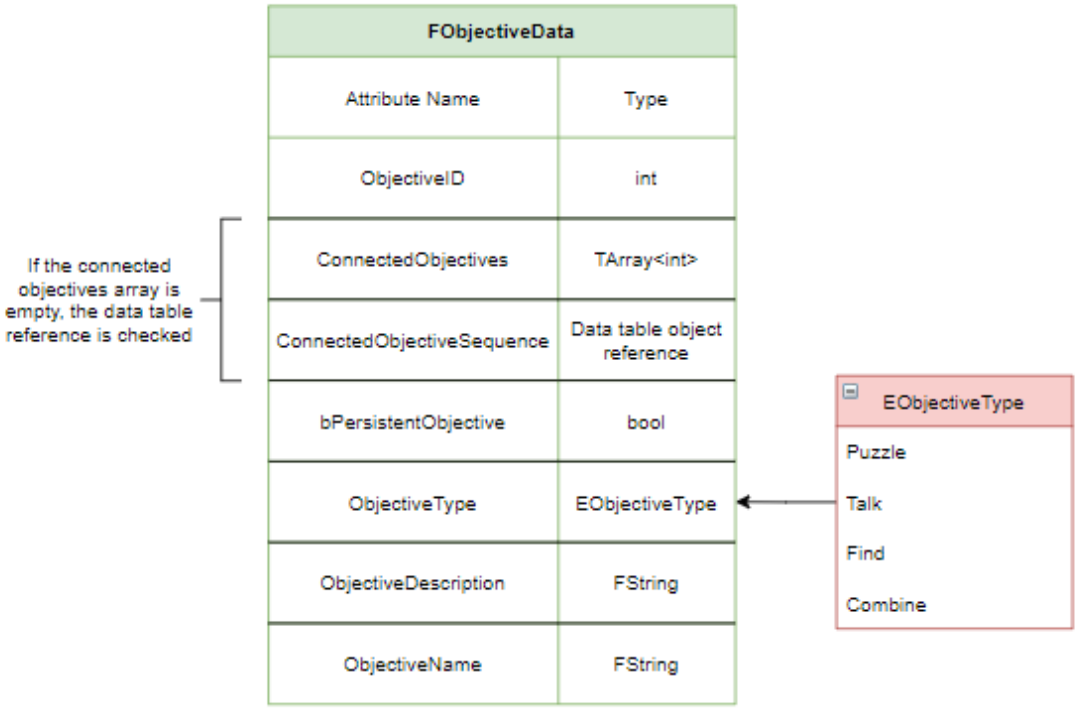
Game Loops


Player is given an objective
Objective Types
Player finds an item
Puzzle minigame must be solved to continue
Item must be combined with another and used
Item must be given to an NPC via dialogue
Interaction with an environmental object
Objective is marked as completed
Player is given help/clues on how to solve objective from select gameplay items and NPCs


Modularity
The ability to use different parts of the framework individually to create different gameplay experiences
Accessibility
Clear documentation allowing the framework to be used by third-parties, with an easy to use editor workflow
Reliability
Creation of a system which is robust and without error, ready for use in projects with a larger asset pool


BaseCharacter
Attribute/Function name
Type
Description
InitaBase
void (interface function)
from BPI_InitaiSable
CharacterName
FString
Actor component (optional)
BPC_DialogueComponent
Attribute/Function name
Type
Description
DialogueData
FDialogueData
Dialogue sequences related to this character
DialogueState
FDialogueState
Current dialogue state - stored as behavior tree data
FDialogueData
Attribute Name
Type
DialogueID
int
CurrentDialogueNode
Blackboard Key Selector
BaseNPC
Attribute/Function name
Type
Description
bHasBeenInteracted
bool
flag for if the NPC has been interacted with yet by the player
ObjectiveNPC
Attribute/Function name
Type
Description
ObjectiveData
FObjectiveData
The related objective data to the NPC
BasePlayer
Attribute/Function name
Type
Description
EntityInventory
BPC_Inventory
Base inventory actor component
OpenInventory
void
Opens the player inventory and updates the player's animation blueprint
BPC_Inventory
Attribute/Function name
Type
Description
InventoryData
TArray<FInventoryData>
Current inventory items
MaxInventorySlots
int
Maximum inventory items
bPersistentInventory
bool
Should the inventory be saved between level transitions and exits (generally, is it the players inventory or not)
CombineItem
void (ItemA, ItemB)
Attempts to combine the currently highlighted item with another clicked one
FInventoryData
Attribute Name
Type
ItemID
int
ItemName
FString
ItemThumbnail
Texture2D
ItemType
EItemTypes
ItemInspectText
FString
CombineItemID
TArray<int>
CombinedItemResult
TArray<int>
Array index of CombinedItemResult corresponds to the index of CombineItemID
EItemTypes
Puzzle
Conversational
Combinational
Misc

When are Child Actors used?



FObjectiveData

Attribute Name	Type
ObjectiveID	int
ConnectedObjectives	TArray<int>
ConnectedObjectiveSequence	Data table object reference
bPersistentObjective	bool
ObjectiveType	EObjectiveType
ObjectiveDescription	FString
ObjectiveName	FString

EObjectiveType

- Puzzle
- Talk
- Find
- Combine

BP_BaseObjectiveManager

Attribute/Function name	Type	Description
CurrentObjectives	TArray<FObjectiveData>	Holds data on the current objectives the player has
ObjectivesData	Data table reference	Reference to the current objective sequence
GetCurrentObjective	returns TArray<FObjectiveData>	Gets the current player objectives
UpdateCurrentObjective	void	Updates the current objective to the next one in the sequence

FAreaLevelData	
Attribute Name	Type
LevelName	FString
LevelPath	FString
LevelObjectiveData	FObjectiveData

GI_GameplayManager		
Attribute/Function name	Type	Description
GetSavedObjectiveData	returns TArray<FObjectiveData>	Gets the current objective data from the persistent storage of the game instance
SaveObjectiveData	void (TArray<FObjectiveData>)	Saves the specified objective data in the game instance. Called before loading a new level
LoadNewArea	void (FAreaLevelData)	Loads a new level from the specified level data, and saves the current objective data if required
ObjectiveData	TArray<FObjectiveData>	The currently saved objective data
PlayerInventoryData	TArray<FInventoryData>	The saved player inventory data
SavePlayerInventory	void	Saves the player's current inventory
GetSavedPlayerInventory	returns TArray<FInventoryData>	Gets the currently saved inventory data

What are the key components?
What are Macros Used

Data Flow

Show a diagram of main initiation chains / spawning of major classes
Link to data tables for stats driven features
When are Instance Editable Variables used?
When are Interfaces and Casts used?

Art Workflows

Geometry

Geometry used in the framework is being created using Blender. To ensure each model is visually consistent when imported into the engine, several standards are being adhered to when creating meshes:

- The tri-count of created meshes should be at a reasonable level to the models scale and where it will be viewed in game. The hard limit for tris on meshes in the framework is 40,000, although I don't expect any of the meshes being created to reach close to it. This is both to keep the framework performant and enable a developer using the framework to easily export and edit the packaged meshes without running into issues that larger tri-counts can potentially cause in some modelling software.
- The normals and smoothing groups of exported meshes should be correctly configured – normals for each face on the mesh should be facing outwards to avoid holes when importing into Unreal, and the smoothing angle should be appropriately set on the asset being created.

A few asset packs are being used which include some pre-made static meshes (see the asset list for a breakdown of these). The assets created to use alongside these meshes should attempt to follow the same stylistic conventions as the asset packs to keep the environment's appearance consistent.

Materials & Textures

What packages are being used
What file format should images be i.e. TGA / PNG
Are there Master Materials
What rendering pipeline is being used i.e. PBR
Are there any texture size / Texel density considerations?

Animation

Other than the default third person mannequin animations included inside Unreal's top down framework, animations for the framework are made using Blender, with Mr Mannequin's (MQ) plugin. MQ's plugin contains a pre-rigged version of Unreal's third person mannequin in both male and female forms, and also includes a variety of handy additions to the rig such as IK for the hands and limbs, head movement to a 'look at' point, and rotation distribution of finger movement by just moving the base bone of the finger. MQ's plugin also includes an exporter specifically made for use with Unreal Engine and the plugin's rig – this will be used to export all of the animations and poses created inside Blender with the tool.

Camera Management

Camera Manager

The camera manager in the framework is driven through by the player pawn's position in the world in relation to the active room actors in the level. Room actors each have a customisable collider which can be configured by a designer and is configured to call an interface on the player controller which allows the current camera to be changed when a player enters the room. The transition between the player's current camera and the new rooms camera can take place as either a fade, or an interpolation between the two cameras depending on if geometry is detected that blocks a trace between the current camera and the new one. Each room's camera is fully modifiable inside the editor, giving a designer the ability to orient the camera in world-space as they see fit. Each camera is attached to a spring arm with a consistent length of 1200 Unreal units to keep the camera and viewport perspective as consistent as possible.

Collision

The collision for all meshes is handled inside engine on a by-mesh basis – using the asset editor for meshes inside Unreal, the simple collision for each mesh can be modified after they are imported, and in the case of more complex meshes the property 'use complex collision as simple' can be enabled to ensure collision is accurate, although this should only be used when absolutely required.

GUI

The textures, materials, and other elements of the UI are all produced inside Adobe Photoshop, exported as .PNG files, and imported into the engine with custom settings. To ensure the UI has an appropriate resolution for any modern display, the import settings for each UI texture are configured to use no compression, filtering, and are set to the UserInterface2D preset inside Unreal's asset property editor.

- The inventory icons are all 64x64 pixels in size
- When imported into Unreal, the textures for UI icons are set to have the UI texture domain, and to have no compression.

Photoshop is being used to create the textures for use inside the framework where required. When making a texture for a static or skeletal mesh, the textures will be exported in .TGA format and imported into the engine with default compression. When making textures
What Texture Compression should be used in engine
What image resolution limits are there

Menu Wireframe

Flow chart of Menu

Fonts

https://www.dafont.com/breamcatcher.font <Typodermic Fonts, Breamcatcher>

The Breamcatcher font is used in all UMG elements of the framework to keep consistency through the UI.

Audio / Video

What Media formats are being used
What compression is being used
Where packages are being used to author files

Tools

For creating additional animations, the default Unreal mannequin rig should be used. To use the same workflow as the framework, Blender should be used with Mr Mannequins tools installed. For a breakdown on how to install this, refer to https://jimkroovy.gumroad.com/l/MrMannequinsTools

Optimisation & Profiling

Profiling Script

Describe the process for testing the game

Profiling Graphics & Rendering

Describe the process for testing the game

Coding Standards and Summary

Programming Standards

To keep any C++ code inside the project accessible for editing, I'll be following Unreal's coding standards outlined in the engine documentation (https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/)

Style Guide

What are best practices to make your code / scripty readable?

Commenting Rules

Make sure everything is commented, leave little to chance.

Code Review Procedures

How often will code be reviewed for correct performance and approach?