ABSTRACT

IQBAL, OWAIS. Villanelle: Towards Authorable Autonomous Characters in Interactive Narrative. (Under the direction of Dr. Chris Martens).

Innovations in intelligent narrative technologies have continued to accelerate, but elude adoption by a broad and diverse audience of storytellers. Authoring accessibility for these technologies is an open challenge, largely due to tensions between story adaptability and author control, explainability, and predictability.

We describe the Villanelle project, an approach to autonomous character authoring that integrates scripting with generativity, using a logic-based foundation that unifies these approaches with reasoning principles that carry through the authoring process.

This thesis presents an implementation of an autonomous character authoring framework that uses behavior trees for scripting agent interaction, user interaction and narrative events. We discuss our implementation which is a UI authoring tool with debugging and rendering of the game along with three examples made using the framework. Ultimately, we hope to extend the project into planning and reasoning to create powerful yet accessible autonomous character authorship.

Villanelle: Towards Authorable Autonomous Characters in Interactive Narrative

by
Owais Iqbal

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Dr. Arnav Jhala	Dr. James Lester

Dr. Chris Martens
Chair of Advisory Committee

DEDICATION

To my friends and family.

BIOGRAPHY

Owais Iqbal was born in Mumbai, India in 1991. He obtained his Bachelor's in Computer Engineering in 2013 from Mumbai University and joined the POEM lab at NCSU during his Master's starting from 2017 in order to pursue his interests in the numerous fields related to interactive narrative.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Chris Martens for all her support and guidance throughout my time at the POEM lab. I would also like to thank my committee members, Dr. Arnav Jhala and Dr. James Lester.

A big thank you to all of the POEM lab members with whom I had the pleasure of working and interacting with - Sasha, Chinmay, Alex, Markus, Abhijeet, Tony, Rook and Claire.

Thank you Sasha, for helping me throughout these two years (and for being the life of any social event).

Thank you Mohit for being a constant this past decade - I consider myself lucky and grateful.

Thank you Salma - this thesis would not have been possible without your support and motivation.

Thank you Animesh, I wouldn't have begun this Master's journey if it wasn't for you.

And thank you Osama, my brother towards whom I can always look at for a source for inspiration.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Interactive narrative is a form of a storytelling experience where the user is able to influence the events and outcomes of the narrative. Interactive narratives can allow for extremely immersive experiences as the user is directly controlling the fate of the story unlike other storytelling media such as fiction novels or movies where the story is predetermined. The extent to which the user is able to change the storyline can vary in different types of interactive narrative.

Due to the high level of immersion that usually comes with interactive narratives, they are well suited not just to entertainment but also to education and training purposes. The rise in popularity of interactive narratives in recent years has led to the introduction of a variety of authoring tools which seek to bridge the gap between the two different skillsets required for creating an interactive narrative: creativity (for authoring the narrative, world and characters) and programming (for realizing the narrative and the different mechanisms the author has in mind).

Authoring tools allow an author to quickly write and test out the narrative ideas that they have in mind without focusing the majority of their attention on implementation details. There is a tradeoff however, with how expressive a tool is versus how approachable it can be for non-programmers. These tools also tend to focus on the player and their choices explicitly and less on how the other characters in the world carry on decision making without the player's actions.

While it makes sense to focus the narrative entirely around the player, there is a rich set of interactive narratives where the player is just another gear in the world's clockwork. Interacting with and influencing the different characters, who have their own objectives and goals in mind, can be

a very engaging and immersive experience if done right. The current existing tools lie at different ends of the approachability-capability scale when it comes to intelligent character authoring, which is to say that if a tool has the capability to encode a complex behavior for an autonomous character, then it also has a steep learning curve.

Some of the properties defined in [MM06] for the wide adoption of an authoring tool for interactive narratives are:

- Usability

  There should be

  - a gentle learning curve
  - less scope for errors
  - easy to remember concepts

- Debugging capabilities

  The author should be able to understand

  - why a particular change of narrative did not work out as they expected
  - which parts of the interactive narrative cannot ever be explored by the player

- Scope

  An interactive narrative contains many parts, and the tool should provide a singular place to author these:

  - characters
  - story
  - user actions
  - user effects
  - descriptions

This thesis presents Villanelle - a framework and tool designed to focus on authoring character behavior with behavior trees, and also using the same core principles to design all the other components of an interactive narrative, keeping in mind the properties mentioned above.

Behavior trees have proven to be quite effective in being able to generate powerful behavior for AI characters in AAA games [MF09]. Their true utility shines in how easy they are to compose, maintain and scale, allowing designers to quickly be able to create the behavior they want in autonomous characters without getting lost in minute implementation details. The designers can craft reusable

subtrees of behavior to be used in different characters or even the same character. Behavior trees allow one to focus on the overall agent behavior they want to achieve.

The idea where Villanelle shines is using behavior trees not only for composing non-player agents in the interactive narrative but also for writing the different descriptions, rules and player actions of the interactive narrative. In this way, we believe that once the author grasps the basics of implementing a behavior tree, they would be able to extend that knowledge to start authoring the narrative they want. We believe this provides a healthy balance of expressiveness and approachability while being a character-focused tool.

1.1 Thesis organization

- Chapter 2 describes related work around interactive narratives, authoring tools and behavior trees.
- Chapter 3 goes describes in detail the Villanelle framework and tool.
- Chapter 4 gives examples of narratives created using Villanelle
- Chapter 5 offers a conclusion and describes future work that can be a part of Villanelle's feature set.

CHAPTER

2

RELATED WORK

2.1 Types of Interactive Narrative

There are different kinds of interactive fiction in digital entertainment:

- Parser based

  The user types in commands to interact with the world

- Multi-choice

  The user is given a set of options to choose from in order to progress the story

- Hypertext

  The user can explore the narrative using the hyperlinks in the text

Quality-based narrative is a kind of narrative where the next piece of content shown to the player depends on certain attributes of the world. They allow for a wide array of choices and get rid of narrative complications introduced by branching narratives.

2.2 Existing tools

Inform7 is a rules-based, natural language style scripting framework that allows one to craft quality based narratives [Ree10]. Choicescript is another scripting language that creates multi-choice input

based narratives [Fab].

Our work takes inspiration from the above tools: being able to craft rules based on world state and presenting the user with relevant choices to mutate that world state. It is a declarative approach to authoring instead of a procedural one as done in tools like Twine [Kli]. When the player does take an action, the other characters developed can choose to react to the new state given their internal logic. The next piece of narrative is then displayed to the user after taking into account the sum of changes caused by the player and all the other characters in the world.

ABL [MS02] is a behavior scripting language which utilizes the mechanics of behavior trees and this was used in the classic interactive fiction work Facade [MS03]. Villanelle seeks to use similar constructs like sequencing, selection and conditioning of behaviors but provide a more accessible route to the creation of the entire narrative using these constructs.

We are also heavily inspired by works like Comme il Faut [McC10] and Versu [ES14]. Versu uses its own authoring domain specific languages to script autonomous character behavior. The user can choose to not act at all and let the autonomous behaviors decide the next events. These kind of systems that allow you to create believable agents with a sophisticated level of autonomy facilitate the creation of social simulation works [McC12].

2.3 Behavior Trees

Behavior trees are a well-researched topic in games, simulations and robotics. Behavior trees provide numerous advantages over finite state machines [Isl], the main ones being scalability and authorability. They have seen widespread adoption in the gaming industry in particular given the ease of authorability they provide. Villanelle chooses to focus on the core concepts of a behavior tree implementation: sequencing, selection, effectors to state and preconditions to all of these. Villanelle also does not tick from the root of every behavior tree, instead continuing from the currently executing node for every tick. The implementation is also done in a functional programming manner, using the formalism described in [Mar18]. Additional features and nodes for the behavior tree implementation are part of our future work and detailed in Chapter 5. A brief precursor to behavior trees is given in Chapter 3.

2.4 Narrative authoring

Behavior trees for narrative authoring and user interaction is explored in detail in [Kap15]. Behavior trees can be used to create branching narratives or rule-based narratives as they can transition from one subtree to another. Their modularity and reusability gives added benefits for encoding a narrative structure with different characters. Unlike the paper, however, Villanelle chooses to map subtrees to user inputs instead of having a singular tree which monitors user input.

CHAPTER

3

DESCRIPTION OF VILLANELLE

3.1 The Villanelle language framework

Villanelle was designed with interactive fiction authors as the primary audience. Thus, the actions and subsequent changes to the state of the world occur in a discrete manner. The game world moves ahead one "tick" when the player chooses to perform an action. If there is a need for the author to progress the world state without explicitly asking the player to make a choice, the author could provide a "Do nothing" action that is internalized as a no-op action on the world state.

We chose this design as Villanelle is intended primarily as a narrative authoring tool. The discrete nature of the framework allows for easier comprehension, development and debugging. This is ideal if the primary purpose is developing/prototyping narrative ideas or gameplay that doesn't necessarily require time-critical input.

On selection of an action to perform by the player, Villanelle takes the behavior tree mapped to that action and all the other behavior trees in the game (agents, user interaction) and executes the next tick for each of them sequentially.

3.1.1 Behavior tree core

Villanelle implements a basic behavior tree implementation as the core that drives all the other components of the framework. This implementation is done in a functional programming manner. The following briefly describes how this implementation is done and could also serve as a revision

for those familiar with behavior trees.

Villanelle stores state in a global "blackboard" similar to common behavior tree implementations. A blackboard is a store of key-value mappings. The value can be of any primitive type but the key should be a string. As variables are only primitive values, they are evaluated by value always.

The primary type that defines all nodes in Villanelle's behavior tree system is a Tick whose type is given by:

```
Tick: () => Status
```

A Tick type is a function that returns a Status of SUCCESS, FAILURE or a RUNNING upon execution. The three main types of nodes that Villanelle uses are:

3.1.1.1 Action node

An 'Action' node is the leaf node of any tree: this is the node type that is actually responsible for changing variables of the world state.

```
Action: (precondition, effects, ticks) => Tick
```

The different components needed to create an Action Tick are:

- Precondition

  ```
  precondition: () => boolean
  ```

  A precondition is a function that will inspect certain variables in the world state and return a boolean value. If it is true, the effects parameter gets executed and if it is false, the Tick returns a status of FAILURE.

- Effects

  ```
  effects: () => void
  ```

  effects is a function that mutates the world state via the framework.

- Ticks

  ```
  ticks: number
  ```

  The author may wish to delay a particular action for a number of turns/world ticks, for example while an agent repairs an engine or when an agent needs time to recover, etc. This parameter specifies the number of times this Action node will return a status of RUNNING before returning a status of SUCCESS when invoked.

  An example of composing an action node is given as:

```
action( () => isDoorUnlocked, () => openDoor() )
```

3.1.1.2 Composite Tick

A 'Composite' is self-explanatory: it is any non-leaf node of a tree that takes an array of other Tick functions as parameters, which become its children in the tree.

```
Composite: (Tick[]) => Tick
```

The two types of composites currently implemented in Villanelle are (these are the same as common behavior tree implementations):

- Sequence

  This node executes its children sequentially until one of them returns a FAILURE. This node returns a SUCCESS on successful execution of all children. If a child node is in the RUNNING status, this node will return a RUNNING status as well.

- Selector

  An inverse of the sequence: it executes children sequentially until one of them returns a SUCCESS and hence the name (it 'selects' a successful node from its children). The node fails if it doesn't find a single successful node. The case for when a child returns RUNNING is the same as for the sequence node.

3.1.1.3 Guard Tick

Villanelle also provides a Guard node, which allows the author to couple a composite node with a precondition. The behavior is intuitive: if the precondition fails, this node would fail else it would return whichever status the Composite node returns.

```
Guard: (precondition, Composite) => Tick
```

3.1.2 Agents

An agent is a structural entity that consists of a behavior tree and variables specific to the agent. Variables are still written to the blackboard, but they are scoped to the agent. Agents provide an easy to understand way to label behavior trees, as the typical use case would be to attach a different behavior tree for each major character in the narrative. It is not limited to only characters though, as the author could also provide a behavior tree for major narrative events in the game with a "Director" agent.

3.1.3 Player Interaction with the Agents and the World

Throughout the development of Villanelle, the primary goal was to make the tool accessible to people unfamiliar with programming constructs. Behavior trees were chosen as the expressive palette for an author as they provide the best cost-to-benefit ratio in terms of learning and expressiveness. The systems for defining how the player interaction and game rules would work were designed keeping in mind the same behavior tree approach used for defining agent behavior.

This makes it easier for the author to reapply the knowledge they learned in order to create all the other pieces of the interactive narrative. This uniformity also has an additional benefit: it allows us to easily create tooling or a high level language on top of this, as the same core concepts are used everywhere.

3.1.3.1 User interaction trees

As described previously, Villanelle does not have a 'scene' or a linear structure which the author can specify. Instead, because of its declarative nature, the author specifies the player interaction given a certain state of the world. The author does this by defining special "user interaction trees" that the framework runs after all the agent trees have run. The framework provides a "user interaction object" which contains the description of the scene, the effect and the list of actions the player can take given the current state. These user interaction trees populate the user interaction object with the necessary components of player interaction to render the game.

There are two main components for player interaction:

- what the player sees:
  - The description of the current 'scene'
  - The current state of the variables (those the author wishes to show the player)
  - A list of actions the player can perform given the current state
- The effects of the action the player chooses to carry out:
  - The mutation of the world state based on certain conditions
  - The description of the effect

The user interaction object is given as:

```
{
    descriptionText: string,
    userActions: [],
    actionEffectsText: string
}
```

We will describe how each of these components are modeled in an author-able manner in the framework.

3.1.3.2 The description of the current 'scene'

The primary text to be displayed on the screen at any given time is scripted through the `displayDescriptionAction`. As the name suggests, this is an action node in the behavior tree system. The type definition is given as:

```
displayDescriptionAction: (text: string) => action
```

This is syntactic sugar for an action node which has the effect of populating the given text into the *descriptionText* field of the user interaction object. If there are multiple *displayDescriptionAction* nodes for a tick, the texts are concatenated and shown to the player. The author can place `$variableName` in the text in order to use the value of a variable in their description.

3.1.3.3 The list of actions

The choice a player can carry out is defined as a structure consisting of the text of that choice and an accompanying behavior tree which would change the world state if that choice is carried out. In this manner, the author can define effects in a declarative way for an action the player takes. Encoding the effects of a player action as a behavior tree reinforces the uniform approach to developing narrative in Villanelle. The type definition is given as:

```
addUserActionTree: (text: string, effectTree: Tick) => action
```

This in turn returns an action node to be plugged into a user interaction tree of a particular location or a scene (depending on relevant variables in the world state). Thus, the act of inserting a behavior tree, which would be evaluated later on when the player actually selects the choice, is itself an action node.

3.1.3.4 User action effect text

An author may wish to display feedback when a player interacts with the world and Villanelle provides a `displayActionEffectText` function as a means of doing so. Action effect texts take precedence over description texts and will override the latter if both are present for a tick. Multiple action effect texts are concatenated together. The author can use `$variableName` in these texts as well.

3.1.4 A game cycle in Villanelle

The `initialize()` method needs to be called to get the initial user interaction object. This method will run all the user interaction trees and populate the user interaction object accordingly.

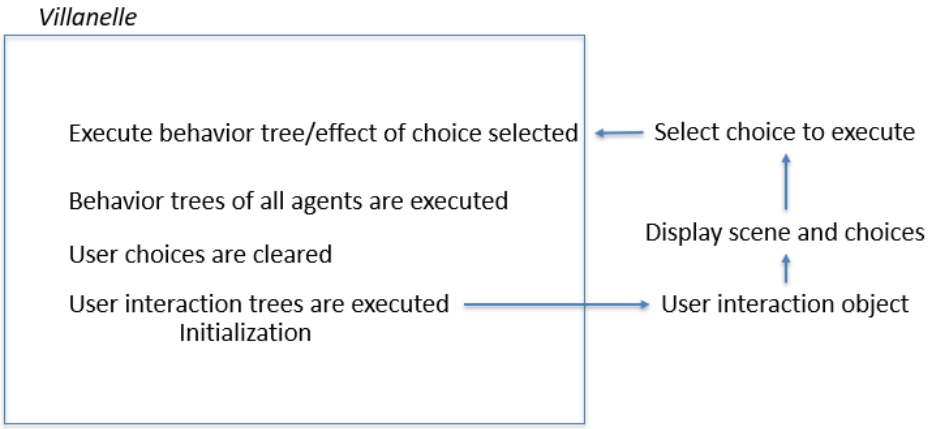

Figure 3.1 The core loop when using Villanelle

Once the player selects an action to execute, the following occurs in sequence:

- `actionEffectsText` is cleared from the user interaction object
- The behavior tree mapped to that user action is run (ticked)
- Each agent's behavior tree is run based on the order the agents were created
- The `descriptionText` and `userActions` fields are cleared from the user interaction object
- User interaction trees are run based on the order they were created

3.1.5 Miscellaneous features

The core API also provides methods for adding a location graph and a basic pathfinding method to navigate that graph. There are also methods to add items and variables related to those items specifically. These are explored in detail in Chapter 4.

3.2 Attempt to make Villanelle visual - Blockly

One of the goals of Villanelle is to be an authoring tool which is intuitive to work in. As the different parts of an interactive narrative in Villanelle are built using components, we reasoned that Google's

Blockly framework would be a great fit given its component based visual editor that has been developed specifically for people with little to no programming experience.

As a brief introduction, Blockly is a library which allows you to integrate a visual editor into your javascript app which can generate code/text given the arrangement of draggable blocks. We developed the blockly integration in the following manner: The user is shown the different categories of blocks on a side panel to choose from:

3.2.1 Components

3.2.1.1 Variables

This has 'Create Variable' as an option in addition to blocks related to already created variables.

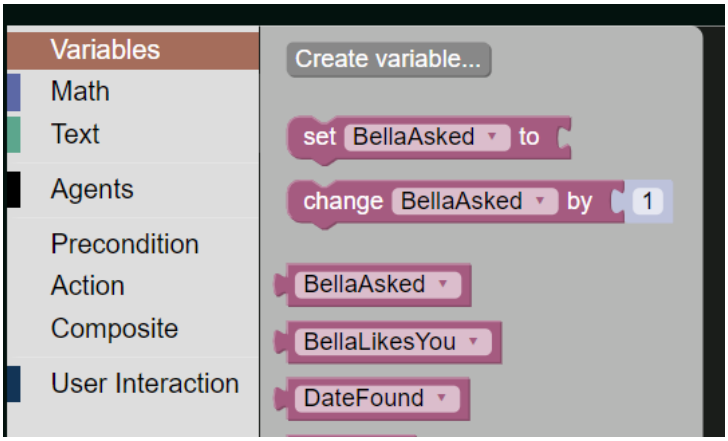

Figure 3.2 Variables menu in the blockly implementation

3.2.1.2 Math

Basic arithmetic blocks are taken from the standard Blockly framework.

Figure 3.3 Math menu in the blockly implementation

3.2.1.3 Text

Text blocks to be used for player interaction, namely, the description, user action text or effect text.

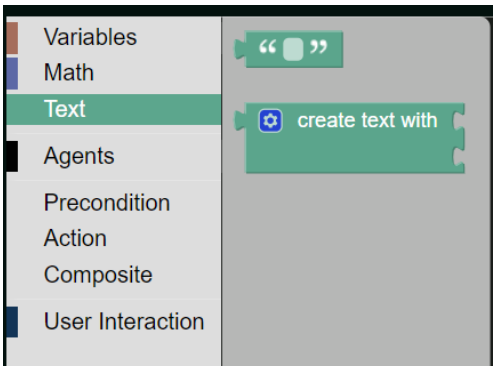

Figure 3.4 Text menu in the blockly implementation

3.2.1.4 Agents

This category has an option to create an agent and a block to attach trees onto for a created agent.

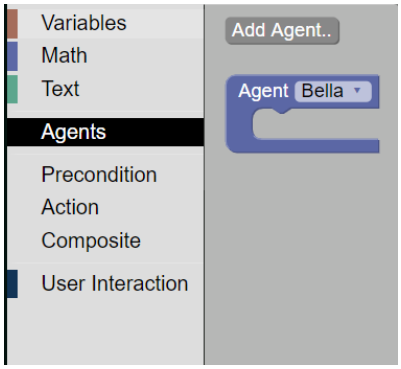

Figure 3.5 Agents menu in the blockly implementation

3.2.1.5 Precondition

This category uses the standard Blockly boolean blocks.

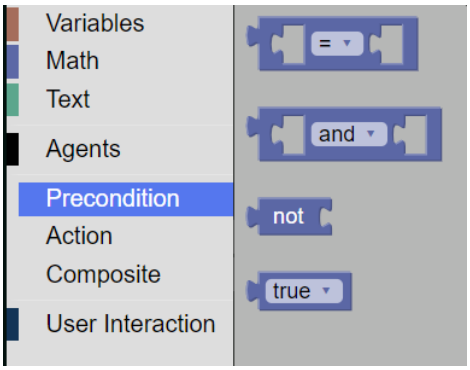

Figure 3.6 Precondition menu in the blockly implementation

3.2.1.6 Action

The action block has attachments for Precondition, Effects and Number of ticks (labeled as Ticks). Effects is a statement block, allowing multiple blocks to be chained together inside the Effects notch.

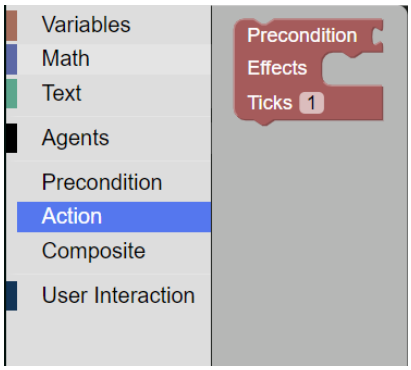

Figure 3.7 Action menu in the blockly implementation

3.2.1.7 Composite

This category has a Sequence, Selector and a Guard block. The sequence and selector were statement blocks, similar to the Effects block. The Guard block has two notches: one to attach a condition and another to run the subsequent node.

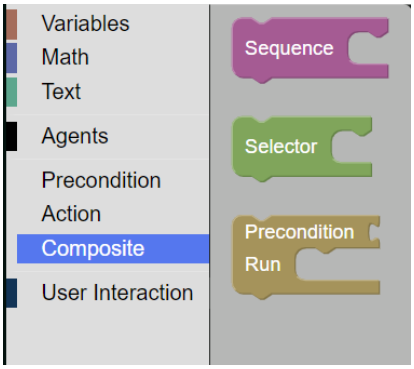

Figure 3.8 Composite menu in the blockly implementation

3.2.1.8 User interaction

This category has the three main components for creating player interaction: a Description block, a User Action block (consisting of 'User action text' and 'Effect behavior tree') and an Effect text block. It also has a main User interaction tree block to which the trees that are defined for user interaction are attached to.

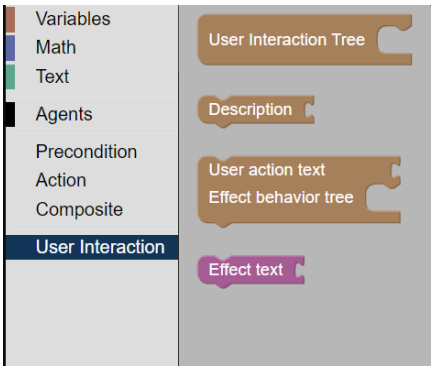

Figure 3.9 User interaction menu in the blockly implementation

3.2.2 Conclusion for blockly

Blockly gave us a nice, snappy visual programming editor that felt pleasing to use and great to look at (an example demonstrating the blocks in use is detailed in Chapter 4). It had some great features out of the box like zooming in and out, quick deletion of blocks and some basic type checking. However, Blockly had limitations when it came to debugging step by step Villanelle behavior expressions. It is still a great tool that helps in creating a narrative without writing a single line of code, but we took the decision to keep it in limbo for the time being and develop our own standalone tool with debugging capabilities. Integrating the blockly editor back into our standalone tool or developing a different kind of visual editor (more tree based as we will see later) is part of our future work.

3.3 Creation of a DSL

We decided to create a DSL (Domain Specific Language) on top of the existing typescript framework to aid with scripting. The idea was to avoid the general boilerplate and overhead that would come with using something like javascript or typescript, and to make it easy to just focus on Villanelle's parts in addition to providing syntax assistance.

3.3.1 Choice of YAML

Using a data format like JSON or XML was our initial thought, as a commonly used data format would have a double benefit:

- it would be more likely that the author is familiar with the format and if not, there would be extensive resources to help out
- it reduces the burden of composing a grammar for lexical breakdown and parsing of the DSL

We chose to go with YAML for the following reasons:

- Less verbosity when compared to some other data formats like XML
- Freedom from the clutter of braces, as YAML uses an indent based structure instead
- YAML has some powerful features such as the ability to reuse previously defined structures as references. This is particularly useful if for example you have a component which is common in multiple behavior trees
- It is a superset of JSON, so it provides flexibility for those more familiar with JSON
- You are able to write comments describing in the YAML structure, unlike JSON

However, using just YAML for the creation of the DSL is not possible, as the author still has to write expressions for conditions and variable assignment and arithmetic calculation. We have to write a grammar for this requirement. To achieve this, we are using ANTLR4 - a general purpose parser generator for custom grammars. The description of how we used these tools is given in the following sections:

3.3.2 ANTLR4

We will first describe how ANTLR4 was used for evaluating expressions. The different tokens that Villanelle identifies are:

- BOOL: booleans (true/false, case-insensitive)
- INT: integers (currently only supporting integer arithmetic)
- STRING: strings (begin and end with single quotation marks)
- ID: variable names (strings that don't begin and end with quotation marks)

3.3.2.1 Assignment expression

The parser rule for an assignment statement is given as:

```
assign: ID := expression
```

Here the := operator is used for assignment of a value obtained by evaluating an expression to a variable.

The expression parser rule is given as:

```
expr: '(' expr ')'
| expr ( '*' | '/' ) expr
| expr ( '+' | '-' ) expr
| expr ( '<' | '>' | '==' | '>=' | '<=' | '!=') expr
| expr 'and' expr
| expr 'or' expr
| 'not' expr
| INT
| BOOL
| STRING
| ID
```

As seen above, Villanelle borrows python syntax for composing boolean expressions. The arithmetic expression syntax uses only the four basic operators as of now. Parenthesis are supported as well to evaluate sub-expressions first in either boolean or arithmetic expressions.

A statement can either be an assignment or an expression:

```
stat: expr NEWLINE | assign
```

A set of statements would constitute a program:

```
prog: stat+
```

The areas where this grammar is used in the DSL are:

- conditional statements

  These occur when defining preconditions for action nodes or using them as a condition to couple with a composite node(a sequence or a selector).

- effects statements

  The list of effect statements (or a program according to the grammar rule specified above) is expected when defining the postconditions of an action node.

- initialization

  As we will see further, the YAML schema requires an 'Initialization' block which uses a list of statements (these would be assignment expressions defining initial values for variables).

3.3.3 The YAML schema

The schema for the YAML DSL is described as follows: The YAML input should be an object, which should have two required properties:

- Initialization
- User Interaction

Any properties apart from the above two will be considered as agent names.

3.3.3.1 The Initialization property

Initialization is required to be an array of YAML string inputs. The items in this array must all be unique. These strings have to adhere to the grammar defined above. This section of the DSL is to be used to initialize values to the variables to be used in the interactive narrative. There isn't a need to declare the variables names separately. The assignment operation writes the value to the blackboard of the framework.

3.3.3.2 The User Interaction property

This property requires an array of behavior tree nodes.

3.3.3.3 The Agent name properties

Property names which are neither 'Initialization' nor 'User Interaction' at the parent level of the YAML script are considered to be agent names. The value of these properties are then required to be behavior tree nodes. Each agent name can only be followed by a single behavior tree node (the root of the tree for the agent).

3.3.3.4 A Behavior tree node

A behavior tree object is an object which must have one of the following schemas:

- An object with the 'effects' property.

  This would form an action node in the framework. This property takes an array of strings (which are expressions of the ANTLR grammar). There can be a condition property present here as well in which case the precondition of the action node would be the given condition expression. If a condition property is not present, the action node would have an always true precondition (`() => true`). This object can have a ticks property as well, which needs to be a non-negative integer specifying the number of world ticks this action node would require.

```
condition: atGym
effects:
    - strength := strength + 2
ticks: 1
```

- An object with a 'sequence' keyword.

  The value of this sequence property is an array of Behavior tree nodes. This object can have a condition property as well. The condition property must have a value of string, which would be a condition in the ANTLR grammar. If a condition property exists, this would be represented as a guard node with a condition and a sequence node in the framework. The author however, does not have to change the node type: all he has to do is specify if a condition is needed to govern this node.

```
condition: isDoorUnlocked
sequence:
      - effects:
            - doorOpened := true
      - condition: isKeyPresent
        effects:
            - keyPickedUp := true
```

- An object with the 'selector' keyword.

  This is similar to the sequence object. This too can have a condition property in addition to the selector property.

```
condition: isDoorLocked
selector:
    - condition: haveKey
      effects:
        - doorOpened := true
    - condition: strength > 4
      effects:
        - doorOpened := true
```

- An object with the 'description' property.

  This needs a value of string and will call the *displayDescriptionAction* function of the framework to set up the description text of a scene.

- An object with the 'user action' property.

  This will have a value of the following object:

  – An object with the 'effect text' and 'action text' properties.

The 'effect tree' keyword takes as input a behavior tree node. The 'action text' property is the text of the choice to be displayed.

```
user action:
    action text: "Eat noodles"
    effect tree:
        effects:
            - hunger := hunger - 2
```

3.4 Standalone Villanelle tool

We have developed a standalone cross-platform desktop tool for writing and debugging the DSL, along with live visualization of all the created behavior trees and live rendering of the game. This will allow authors to quickly prototype their ideas with the built-in editor and play the game immediately after making their changes without requiring any additional step. With live visualization of the trees, the authors can graphically understand the structures that they are building and use the live error reporting to help fix syntax and semantic issues instantly. If the compilation succeeds, the user can play the game in the tool itself and see the statuses of the different nodes of the behavior trees as the game progresses.

As the core API was developed in Typescript, it was a straightforward decision to use Electron as the framework for creating a cross-platform desktop application. We used React as the library for handling the view state of the application, and used Palantir's Blueprintjs to give it a sleek UI theme.

There are two main tabs in the application: Script and Play.

3.4.1 Script Tab - Editor Pane

The script tab primarily consists of the editor. The editor was realized using an open source embeddable code editor called Ace Editor. We used the built-in language mode for YAML which provided features like:

- Tabs considered as spaces: since YAML does not allow tabs, the editor does the mapping for us when writing the DSL
- Expandable/Collapsible structures: every individual indented YAML structure can be expanded/collapsed to allow one to get a bigger picture/have a more minimal view.

In addition to this, the Ace Editor provides general features like a powerful search/replace functionality (which has regex support), highlighting other same tokens when one is highlighted and line numbers.

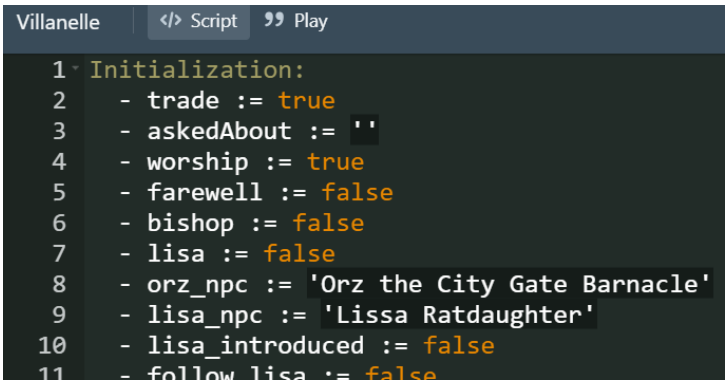

Figure 3.10 Screenshot of the Editor Pane

3.4.1.1 Error checking

The different checks for types and schema validation have been described in the section 3.3.3. We use *json-schema* and *ajv* libraries to perform these checks. We also run the condition and assignment expressions against the ANTLR4 grammar. Any errors in these checks are reported in a bottom bar as shown:

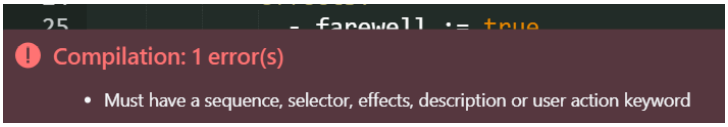

Figure 3.11 The error display

If the error checks succeed, the bar becomes green:

Figure 3.12 The success display

Every change to the YAML input in the editor causes the error checks to be run again, and if they succeed, it causes the game to be rendered again.

3.4.1.2 Auto-complete

The editor has a set of keywords it suggests for auto-complete based on matching prefixes of what the user is currently typing. It also suggests tokens that have already appeared in the YAML input.

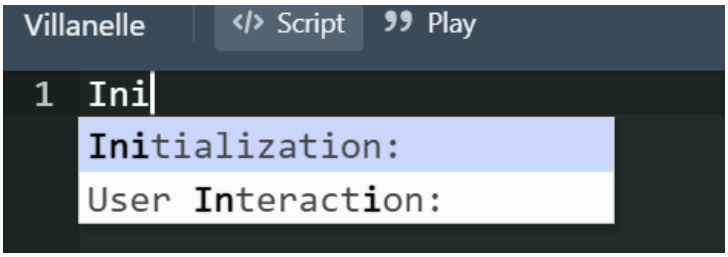

Figure 3.13 The autocomplete prompt

The keywords currently are:

- sequence
- selector
- condition
- effects
- ticks
- effect text
- description
- action text
- effect tree
- user action
- Initialization
- User Interaction

3.4.2 Script Tab - Tree Visualization

In a side panel next to the editor, a tree is rendered live with every change the user makes to the YAML in the editor. This tree is also responsible for highlighting the errors in the code structure if there are any. Every individual node which has children is expandable and collapsible. The following is how the different components are shown graphically:

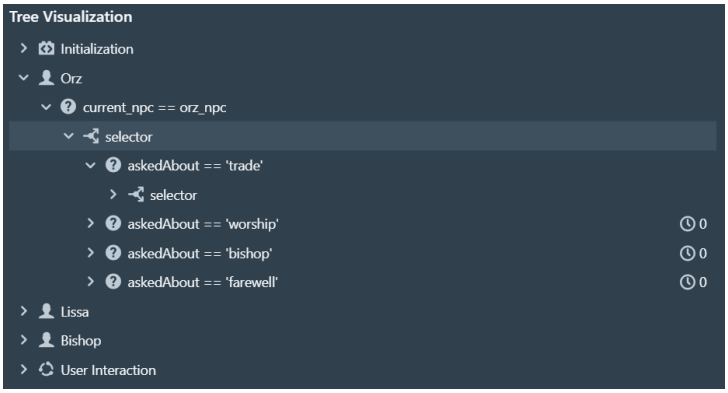

Figure 3.14 The tree visualization panel

3.4.2.1 Incorrect YAML input

If the input is an invalid YAML schema, i.e. it violates any of the general YAML rules, the tree isn't rendered and a message is shown as such:

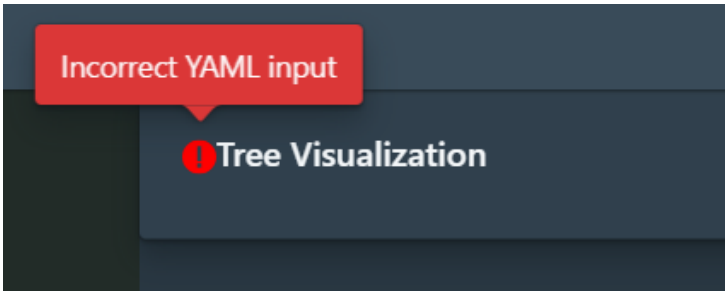

Figure 3.15 Error displayed when it is not a well-formed YAML input

3.4.2.2 Initialization

The different statements for initializing variables which will be used in the world state are shown under the Initialization node.

Figure 3.16 Initialization subtree

3.4.2.3 Agent nodes

Behavior trees for agent nodes are rendered under each agent:

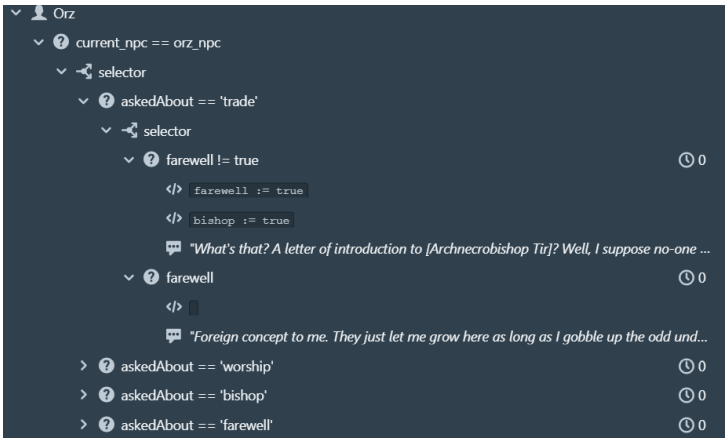

Figure 3.17 The behavior tree of an agent NPC

The visual structure of the behavior trees matches node for node the structure in the YAML input. However, the conditions in actions, sequence or selectors are represented as individual nodes themselves, with the associated behavior tree node rendered as a child. This was done to visually create a sense of 'gate-keeping' the conditions provide in terms of their coupling with nodes of a behavior tree.

For example, for an action, Fig. 3.18 would be shown:

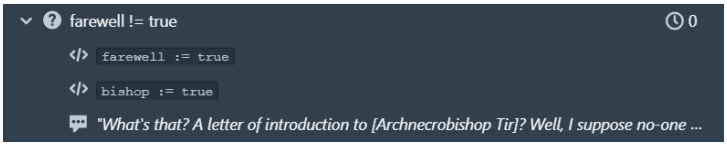

Figure 3.18 Nodes rendered under a condition node

Here, the condition node can be collapsed and the underlying children will be hidden, thus allowing for better clarity when it comes to viewing at a glance as you could hide the nodes that won't activate for a failing condition under that condition node.

The number of ticks an action node takes is displayed as a clock symbol on the right hand side of the corresponding condition node (if the action has no explicit condition node, a 'true' condition node is rendered) as shown in Fig. 3.18.

The graphics for the remainder of the nodes are listed in the appendices.

3.4.2.4 Errors on the nodes

Nodes which have errors with types or with the Villanelle YAML schema are reported as red nodes and their children are not rendered. The error message is displayed on hovering over the erroneous node.

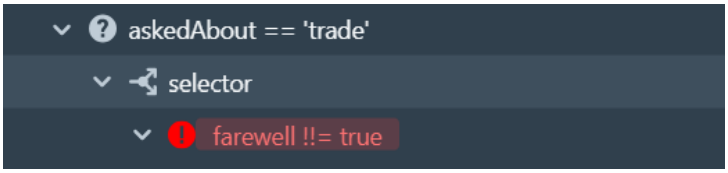

Figure 3.19 The condition expression is syntactically incorrect

All ancestors of the erroneous node are automatically expanded so the author does not have to search on their own.

3.4.3 Play Tab

On every change to the YAML input, the game is rendered immediately in the 'Play' tab. If there are any errors in the input, the compilation fails and a message is displayed instead:

Figure 3.20 The game is not rendered on compilation errors

3.4.3.1 Rendered Game

The rendered game has two main components: the text display and the multi-choice input.

- Text display

This consists of the title of the game and the scene description or effects description, if any. Scene description is retrieved from *text* and the effects text from *actionEffectsText*

- Multi-choice input

  Each choice gets rendered as a clickable button which will execute the behavior tree associated with that action on click. The text for the actions is retrieved from the 'userActionsText' of the user interaction object.

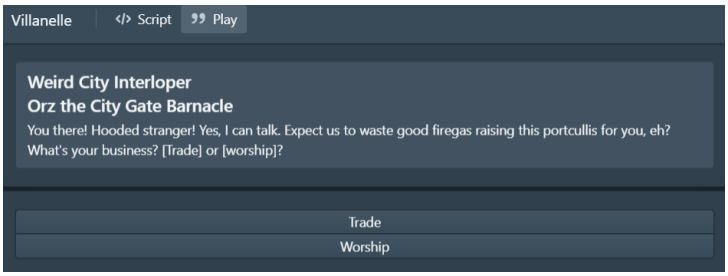

Figure 3.21 The game is rendered in the play tab

3.4.3.2 Tree Visualization

As the author plays through the game, the visualized nodes are highlighted based on how they were processed:

- green

  The node was executed successfully i.e. it has a Status of SUCCESS/the condition was evaluated as being true.

- red

  The node has a Status of FAILURE/the condition was evaluated as being false.

- orange

  The node has a Status of RUNNING/the ticks remaining for this node or for one of its descendants is non-zero.

- not colored

  The node was not evaluated last cycle.

On every action the user takes (as described in section 3.1.4), the statuses of the nodes change and the tree is refreshed showing the new changes. In this manner, the author gets live feedback on taking any action in the game and they can adjust the YAML input accordingly.

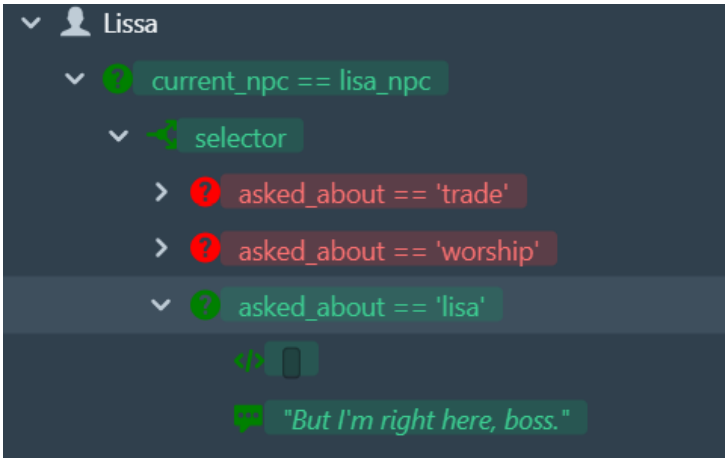

Figure 3.22 The different statuses for the nodes show up as you play the game

CHAPTER

4

EXAMPLES

4.1 Example using the Typescript framework - Alien Isolation

4.1.1 Overview

Alien Isolation is an example game that was created when the first core APIs of Villanelle were developed. This is inspired by the Creative Assembly game of the same name. Similar to the AAA game, here the player has to make his way out of the spaceship after collecting two crew cards that would activate an elevator allowing for escape, all the while trying to avoid being caught by the human hunting alien.

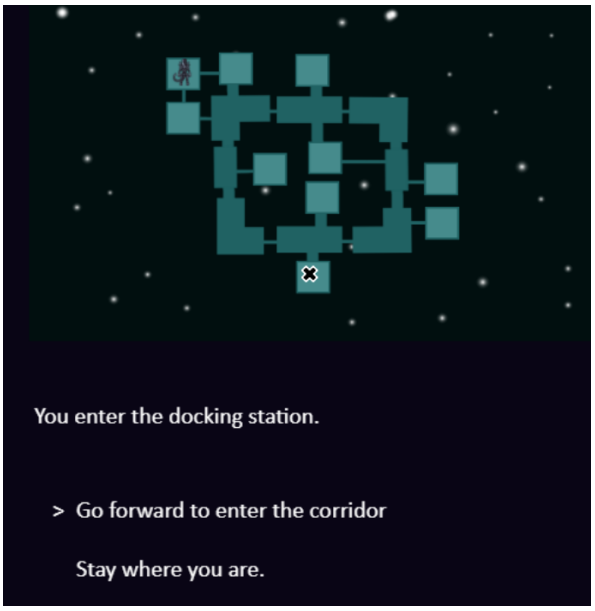

Figure 4.1 The alien starts in the top-left room, while the player (marked with the X icon) starts from the bottom

The alien icon is shown for demo purposes. If the alien and the player end up in the same location, the game ends in a loss. The player is given different navigation choices based on which room they are present in. They are also given choices for picking up crew cards if they are present. If the alien is in an adjacent room, the player is given a prompt telling them that they hear sounds nearby.

4.1.2 Location navigation

For this game, we make use of the basic location navigation APIs in the core framework.

We add the location graph as follows:

```
//addLocation(locationName, adjacentLocations)

addLocation(START, [BC_CORRIDOR]);
addLocation(BC_CORRIDOR, [BL_CORRIDOR, BR_CORRIDOR, LAB]);
addLocation(BL_CORRIDOR, [ML_CORRIDOR]);
```

There is a Villanelle pathfinding method to select the next adjacent room en route to a given destination:

```
getNextLocation(current, destination)
```

4.1.3 Alien Behavior

The alien selects a random destination and makes its way toward it. If it reaches its destination, it will select a new random destination.

If the player enters the same location as the alien or if it's the other way round, the alien will "eat" the player and the game ends. The behavior tree for the alien can be given as:

```
selector([
    eatPlayer,
    sequence([
        search, eatPlayer
    ])
])
```

`eatPlayer` is just an action node that ends the game if the precondition that the alien and player are in the same location is true. This forms the first child of the parent selector node. If this precondition fails, the selector moves on to the next node, where it performs a sequence: move to the next location and perform the `eatPlayer` action again.

In this manner, we cover both scenarios: checking if the player is present before and after movement.

The `search` is a subtree that moves towards a destination and sets a new one if we have already reached the current one.

```
sequence([
    gotoNextLocation,
    guard(
            () => destination == current, setNextDestination
    ),
])
```

4.1.4 Location based narrative

For user interaction, we define multiple subtrees for each of the locations specifying the navigation actions the player can take and their effects (changing the player's location).

For example, the starting location's tree is:

```
guard(
    () => getVariable(playerLocation) == START, //precondition
    sequence([
            displayDescriptionAction("You enter the docking station."),
```

```
            addUserAction("Go forward to enter the corridor", () =>
                ↪ setVariable(playerLocation, BC_CORRIDOR)),
            addUserAction("Stay where you are.", () => {})
        ]
))
```

When the user is in a location with a crew card, the following subtree activates granting the user the choice to pick up the crew card:

```
guard(
() => getVariable(playerLocation) == getItemVariable(crewCard, "
    ↪ currentLocation"),
sequence([
        displayDescriptionAction("You notice a crew card lying around.")
            ↪ ,
        addUserAction("Pick up the crew card",
            action(()=>true, () => {
                displayActionEffectText("You pick up the crew card.");
                setItemVariable(crewCard1, "currentLocation", "player");
                setVariable(crewCardsCollected, getVariable(
                    ↪ crewCardsCollected) + 1);
            })
        )
    ]
))
```

If the user is in the elevator area and has both crew cards, he should be able to end the game:

```
guard(() => getVariable(playerLocation) == EXIT_ELEVATOR,
sequence([
        displayDescriptionAction("You reach the exit elevator."),
        selector([
            guard(() => getVariable(crewCardsCollected) >= 2,
                sequence([
                    displayDescriptionAction("You can now activate the
                        ↪ exit and flee!"),
                    addUserAction("Activate and get out!", () => {
                        setVariable("endGame", "win");
                        setVariable(playerLocation, "NA")
```

```
                    })
                ])),
            displayDescriptionAction("You need 2 crew cards to activate
                ↪ the exit elevator system.")
        ]),
        addUserAction("Move back in the corridor", () => setVariable(
            ↪ playerLocation, TC_CORRIDOR)),
        addUserAction("Stay where you are.", () => {})
    ]
))
```

Finally, the thumping effect was achieved using the following subtree:

```
guard(
    () => areAdjacent(getVariable(playerLocation), getAgentVariable(
        ↪ alien, "currentLocation")),
    displayDescriptionAction("You hear a thumping sound. The alien is
        ↪ nearby.")
)
```

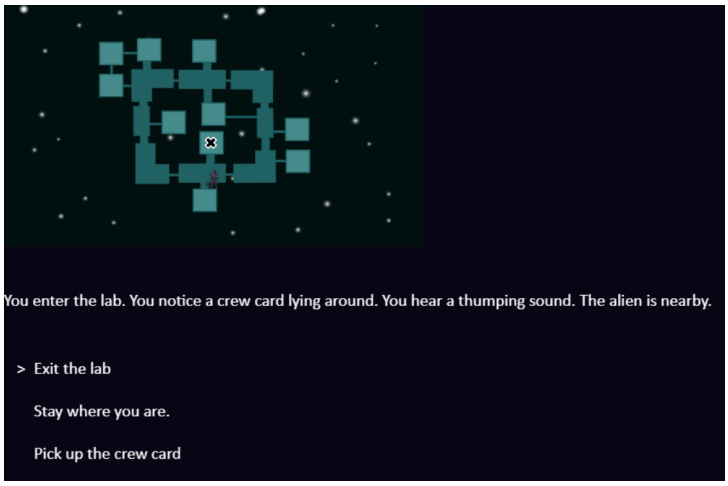

Figure 4.2 Here the two description texts are concatenated and shown together: noticing of the crew card and hearing of the alien

4.1.5 Conclusion

The rendering of the game and choice input was done using HTML5 canvas. The new state and choices were displayed based on the user interaction object given by the framework at every tick.

The Alien Isolation game gave us a nice example to work with when developing the core APIs of Villanelle. It showcases a location based narrative and uses all the features of Villanelle.

4.2 Example using Blockly - Monster Prom

4.2.1 Overview

To showcase all the Villanelle features using Blockly, a small narrative work inspired by Monster Prom was developed. Unlike the original game, this example has only two characters: Bella (a vampire) and Jacobine (a werewolf). Your task is to find a date: you can ask either of them but you will be rejected based on something you are missing. After acquiring sufficient intelligence/strength and fangs/fur, the next attempt would result in a date depending on who you ask.

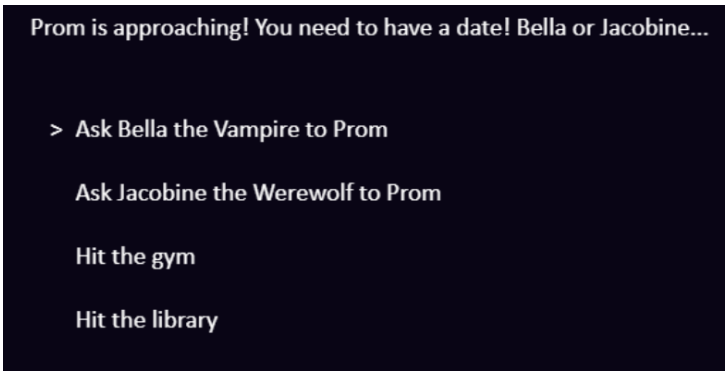

Figure 4.3 The Monster Prom inspired game

4.2.2 Initialization

The initialization statement blocks come before any other blocks.

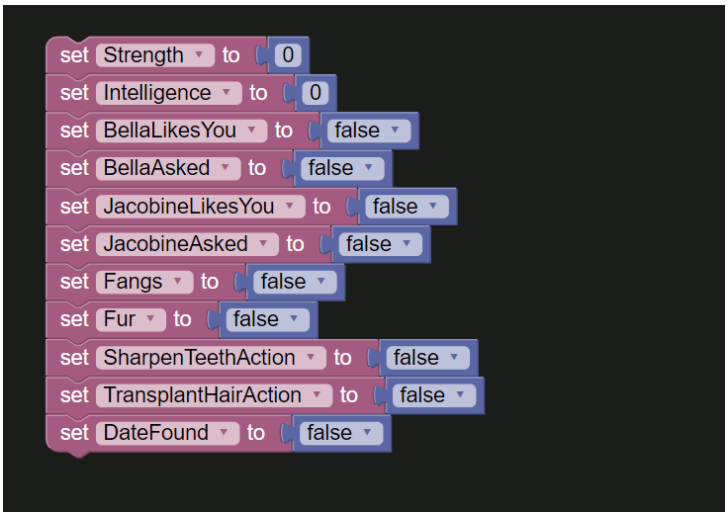

Figure 4.4 Initializing all the variables in the game

4.2.3 Blocks for Agent trees

Behavior tree blocks are assigned to their respective agent blocks.

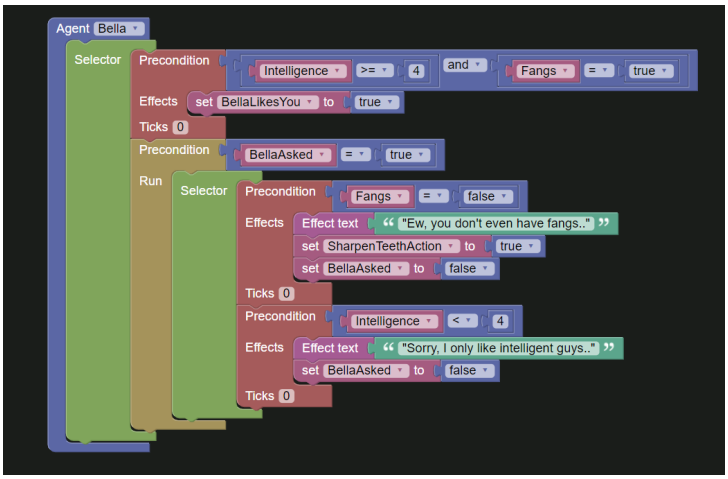

Figure 4.5 Agent behavior tree of Bella the Vampire

4.2.4 Blocks for User Interaction

Different subtrees for user interaction rules are chained together in the user interaction block.

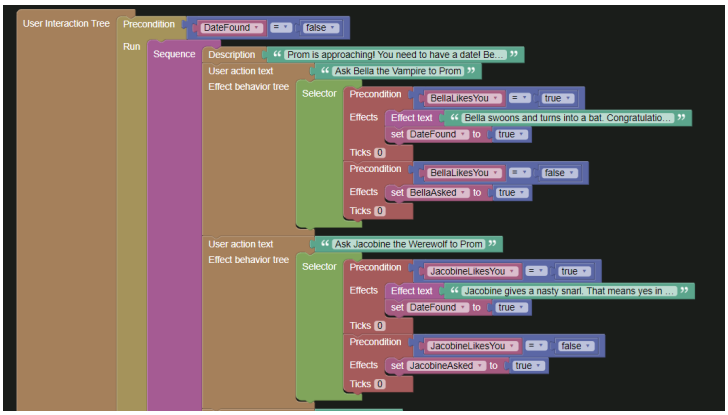

Figure 4.6 A part of the user interaction block

4.2.5 Conclusion

Blockly would generate a typescript file which would render the game when used with the Villanelle framework.

The end result was a mini-example of 2 non-player characters, done entirely in a visual manner.

4.3 Example using the Standalone Tool - Weird City Interloper

4.3.1 Overview

Weird City Interloper is an Inform7 game featuring a potential heir to the throne trying to save the city's king from an evil priest. It has about a dozen different characters and 27 topics to ask about to each of these characters. Not all topics or characters are immediately available but you can uncover them by asking about a discovered topic to an introduced character.

Orz the City Gate Barnacle
You there! Hooded stranger! Yes, I can talk. Expect us to waste good firegas raising this portcullis for you, eh? What's your business? [Trade] or [worship]?

>trade
What's that? A letter of introduction to [Archnecrobishop Tir]? Well, I suppose no-one cares what the leader of a dead religion spends his coin on. I'll let you through. [Farewell].

>
(Some topics of interest: [trade] and [worship].)

(Say [farewell] to enter the city.)

>

Figure 4.7 The original Inform7 game

The game is unique in the sense that the scenes of the narrative are the different characters themselves. The player moves between different characters by suggesting their name to their guide. The game doesn't really need a parsing interface as the available topics are shown to the player if he wishes to see them.

We translated a portion of the game in the standalone tool: 3 characters and 10 topics. The discovered topics show up as multiple choices.

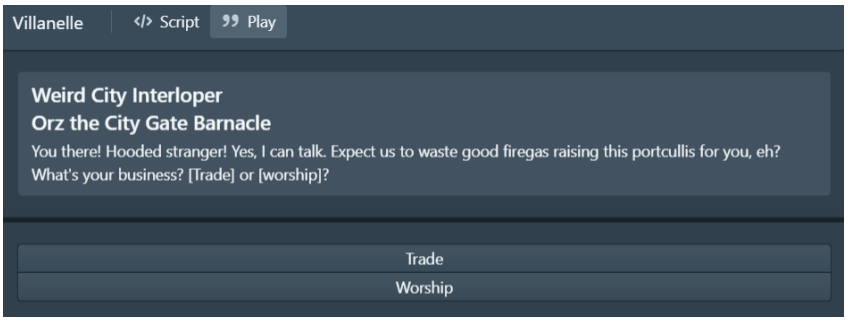

Figure 4.8 The recreated version in Villanelle

4.3.2 Variables

We keep a track of discovered topics (eg: trade, worship, etc.) and characters (Orz, Lissa, Tir) in their individual variables:

Figure 4.9 Initializing all the variables

In addition, we also keep a track of the current npc we are talking to in the variable `current_npc`. The current topic being asked about is stored in the variable `asked_about`.

4.3.3 Agent Behaviors

Each character tree starts with a condition making sure that the `current_npc` is themself. The condition is wrapped around a selector node which selects a subtree based on the value of the `asked_about` variable.

If asking about the topic does not lead to a new discovery, the only effect the node would have would be to display `effect text` describing the character's thoughts on the topic.

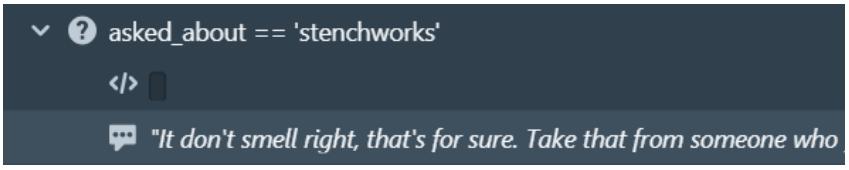

Figure 4.10 The response doesn't involve any state change

If it does lead to a new topic, the corresponding boolean variable for that topic is set to true along with displaying the `effect text`.

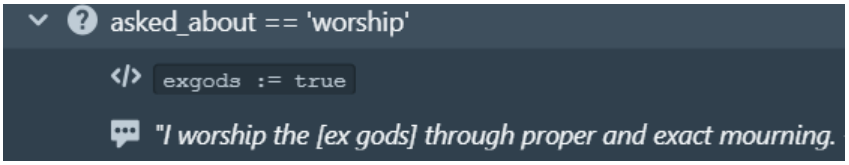

Figure 4.11 Asking about worship to the bishop reveals the 'ex gods' topic

4.3.4 User Interaction

There is a user interaction tree that is a sequence node that goes through all the topics in the game and displays a choice to the player if the corresponding topic variable is true. Some of the nodes for topics are combined with an always true node to a parent selector node, thus making sure the selector node would always succeed and the topics following the current one are not ignored in the parent sequence operation. The effect of executing the choice is to set the `asked_about` variable to the topic chosen.

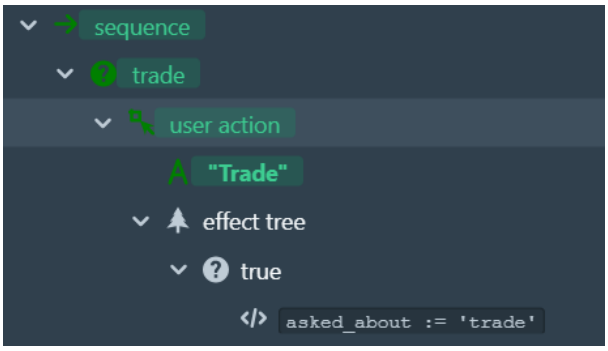

Figure 4.12 The 'trade' topic subtree

There is another tree for displaying description text for each of the characters, as the characters in the game are scenes themselves.

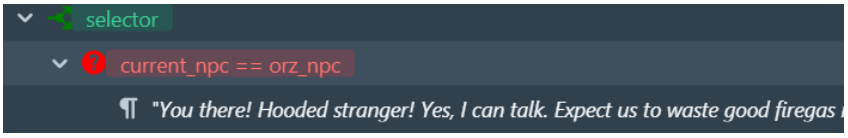

Figure 4.13 Description on meeting with Orz

There are also selectors for displaying description text depending on whether you are meeting with them for the first time.

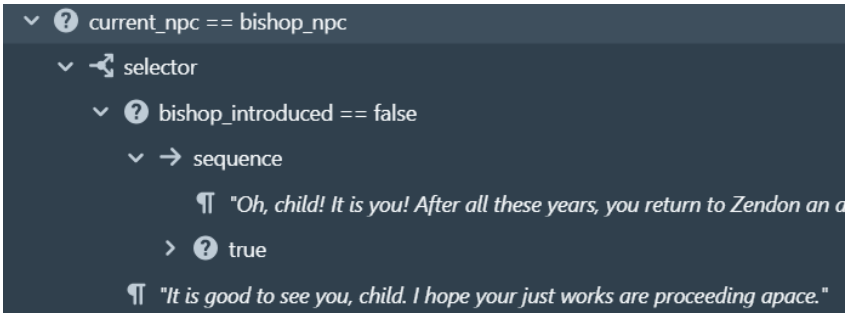

Figure 4.14 Different texts when meeting the bishop depending on whether we have met him before

4.3.5 Execution

The following is an example of the corresponding nodes being highlighted when you ask a topic to a character:

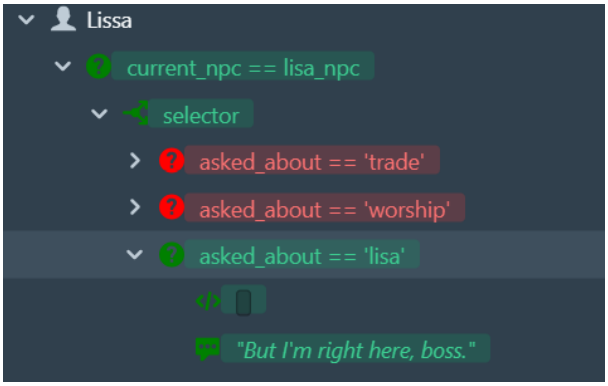

Figure 4.15 Asking Lissa about herself

4.3.6 Conclusion

This significant subset of a quite large Inform7 game was done entirely in the Villanelle DSL. Extending the example to cover the entire game is trivial as the structures are in place, but time-consuming. We believe implementing this subset in the tool demonstrates the tool's capability to handle what we had aimed for with Villanelle.

CHAPTER

5

CONCLUSION AND FUTURE WORK

In this thesis we presented Villanelle: an API framework and a standalone tool to use behavior trees not only to author character behaviors but every other part of an interactive narrative experience as well.

We accomplished this using the javascript ecosystem, namely, Typescript, Electron, React and Canvas.

We were able to demonstrate entire narrative experiences authored using the core featureset of Villanelle. However, there are still a few shortcomings with our ideas currently. Firstly, it is difficult to co-ordinate the different actions of non-player characters in order to present a particular narrative piece to the player. We have also not implemented a full social simulation [McC12] using Villanelle.

5.0.1 Future work

Villanelle provides a good base that can be built upon to provide powerful features which will enhance the authoring experience and allow for more expressiveness in authored characters.

5.0.1.1 State jumping and rewinding

The next planned feature is to allow the author to change the state of any variable they want and see the resulting transformations that occur in the narrative. A companion feature is to allow the author to undo the action they just did in the world instead of restarting the entire narrative.

5.0.1.2 Additions to the behavior tree implementation

The current behavior tree implementation only has the two composite nodes - sequence and selector. A great addition to these would be decorator nodes: nodes that wrap around other nodes and change the result returned. For example, an always succeeding decorator would return a status of success regardless of the result of execution of its descendants. This would eliminate the no-op node implementation needed in some cases.

The current implementation also does not execute the entire tree from the root node on every tick. This can be problematic in some cases: if the author wishes to stop the agent from doing what it currently is and to respond immediately to an event. Interruptible nodes/some kind of event subscribing architecture could be useful in these cases [CS09].

5.0.1.3 Story/Behavior tree analysis

As the number of characters and narrative scenes scale, the difficulty of checking completeness increases for the author. Since Villanelle employs a quality-based narrative style, it can be cumbersome to verify if all the content is reachable by the player.

A simulation based approach of trying out all combinations for a large number of iterations and reporting the nodes that were never reached is a possible approach to tackle this.

5.0.1.4 Planning

Behavior trees lend themselves very well to Hierarchical Task Network planning. We could use this technique to allow the trees of non-player agents to compose themselves using the available trees the author has expressed. [Neu18]

5.0.1.5 Advanced data structures

Adding advanced data structures like different types of collections would make it easier for advanced authors to bring about more functionality.

5.0.1.6 Different types of interactive narratives

Extending Villanelle to be used in different kinds of interactive narrative other than multi-choice input ones is planned.

BIBLIOGRAPHY

[CS09] Cutumisu, M. & Szafron, D. “An Architecture for Game Behavior AI: Behavior Multi-Queues.” *AIIDE*. 2009.

[ES14] Evans, R. & Short, E. “VersuâĂŤa simulationist storytelling system”. *IEEE Transactions on Computational Intelligence and AI in Games* **6**.2 (2014), pp. 113–130.

[Fab] Fabulich, D. *Choice of Games*. URL: `www.choiceofgames.com/make-your-own-games/choicescript-intro`.

[Isl] Isla, D. *Handling Complexity in Halo 2 AI*. URL: `http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php`.

[Kap15] Kapadia, M. et al. “Evaluating the authoring complexity of interactive narratives with interactive behaviour trees”. *Foundations of Digital Games* (2015).

[Kli] Klimas, C. *Choice of Games*. URL: `twinery.org`.

[Mar18] Martens, C. et al. “A Resourceful Reframing of Behavior Trees”. *arXiv preprint arXiv:1803.09099* (2018).

[MS02] Mateas, M. & Stern, A. “A behavior language for story-based believable agents”. *IEEE Intelligent Systems* **17**.4 (2002), pp. 39–47.

[MS03] Mateas, M. & Stern, A. “Façade: An experiment in building a fully-realized interactive drama”. *Game developers conference*. Vol. 2. 2003, pp. 4–8.

[McC10] McCoy, J. et al. “Authoring game-based interactive narrative using social games and comme il faut”. *Proceedings of the 4th International Conference & Festival of the Electronic Literature Organization: Archive & Innovate*. Citeseer. 2010, pp. 1–8.

[McC12] McCoy, J. et al. “Prom week”. *Proceedings of the International Conference on the Foundations of Digital Games*. ACM. 2012, pp. 235–237.

[MM06] Medler, B. & Magerko, B. “Scribe: A tool for authoring event driven interactive drama”. *International Conference on Technologies for Interactive Digital Storytelling and Entertainment*. Springer. 2006, pp. 139–150.

[MF09] Millington, I. & Funge, J. *Artificial intelligence for games*. CRC Press, 2009.

[Neu18] Neufeld, X. et al. “A Hybrid Approach to Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees”. *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2018.

[Ree10] Reed, A. *Creating Interactive Fiction with Inform 7*. 1st. Boston, MA, United States: Course Technology Press, 2010.

APPENDIX

APPENDIX

A

DSL SCRIPTS FOR EXAMPLE GAMES

A.1 Monster Prom

```
Initialization:
  - Strength := 0
  - Intelligence := 0
  - BellaLikesYou := false
  - BellaAsked := false
  - JacobineLikesYou := false
  - JacobineAsked := false
  - Fangs := false
  - Fur := false
  - SharpenTeethAction := false
  - TransplantHairAction := false
  - DateFound := false

Bella:
  selector:
```

```
    - condition: Intelligence >= 4 and Fangs == true #precondition for the
      ↪ action below
      effects:
        - BellaLikesYou := true
      ticks: 0 #optional
    - condition: BellaAsked == true #this is a guard for the selector below
      selector:
        - condition: Fangs == false
          effect text: Ew, you don't even have fangs..
          effects:
            - BellaAsked := false
            - SharpenTeethAction := true
          ticks: 0 #optional
        - condition: Intelligence < 4
          effect text: "Sorry, I only like intelligent guys.."
          effects:
            - BellaAsked := false
          ticks: 0

Jacobine:
  selector:
    - condition: Strength >= 4 and Fur == true #precondition for the action
      ↪  below
      effects:
        - JacobineLikesYou := true
      ticks: 0 #optional
    - condition: JacobineAsked == true #this is a guard for the selector
      ↪ below
      selector:
        - condition: Fur == false
          effect text: "I'm more of a furry type of person.."
          effects:
            - JacobineAsked := false
            - TransplantHairAction := true
          ticks: 0 #optional
```

```
      - condition: Strength < 4
        effect text: "You are kinda weak.."
        effects:
          - JacobineAsked := false
        ticks: 0

User Interaction:
  - condition: DateFound == false
    sequence:
      - description: "Prom is approaching! You need to have a date! Bella
          ↪ or Jacobine..."
      - user action:
          action text: "Ask Bella the Vampire to Prom"
          effect tree:
            selector:
              - condition: BellaLikesYou == true
                effect text: 'Bella swoons and turns into a bat.
                    ↪ Congratulations, you have a date!'
                effects:
                  - DateFound := true
                ticks: 0
              - condition: BellaLikesYou == false
                effects:
                  - BellaAsked := true
                ticks: 0
      - user action:
          action text: "Ask Jacobine the Werewolf to Prom"
          effect tree:
            selector:
              - condition: JacobineLikesYou == true
                effect text: 'Jacobine gives a nasty snarl. That means yes
                    ↪ in werewolf-speak. Congratulations!'
                effects:
                  - DateFound := true
                ticks: 0
```

```
        - condition: JacobineLikesYou == false
          effects:
            - JacobineAsked := true
          ticks: 0
- user action:
    action text: "Hit the gym"
    effect tree:
      effects:
        - Strength := Strength + 2
      effect text:
        "Your strength improves to $Strength (+2)"
      ticks: 0
- user action:
    action text: "Hit the library"
    effect tree:
      effects:
        - Intelligence := Intelligence + 2
      effect text:
          "Your intelligence improves to $Intelligence (+2)"
      ticks: 0
- selector:
    - condition: SharpenTeethAction == true
      user action:
        action text: "Sharpen your teeth"
        effect tree:
          effect text: "You sharpen your teeth into makeshift fangs."
          effects:
            - Fangs := true
            - SharpenTeethAction := false
          ticks: 0
    - effects:
        _ ''
      ticks: 0
- selector:
    - condition: TransplantHairAction == true
```

```
      user action:
        action text: "Transplant hair onto your body"
        effect tree:
          effect text: "You become more hairy."
          effects:
            - Fur := true
            - TransplantHairAction := false
          ticks: 0
    - effects:
        _ ''
      ticks: 0
```

A.2 Weird City Interloper

```
Initialization:
  - trade := true
  - asked_about := ''
  - worship := true
  - farewell := false
  - bishop := false
  - lisa := false
  - exgods := false
  - heretics := false
  - nodroth := false
  - stenchworks := false
  - zook := false
  - orz_npc := 'Orz the City Gate Barnacle'
  - lisa_npc := 'Lissa Ratdaughter'
  - lisa_introduced := false
  - follow_lisa := false
  - bishop_npc := 'Archnecrobishop Tir'
  - bishop_introduced := false
  - zook_npc := 'Zook Spiralhouse'
  - zook_introduced := false
```

```
  - current_npc := orz_npc
  - going_to_npc := 'No one'

Orz:
  condition: current_npc == orz_npc
  selector:
    - condition: asked_about == 'trade'
      selector:
        - condition: "farewell != true"
          effect text: "What's that? A letter of introduction to [
              ↪ Archnecrobishop Tir]? Well, I suppose no-one cares what the
              ↪  leader of a dead religion spends his coin on. I'll let you
              ↪  through. [Farewell]."
          effects:
            - farewell := true
            - bishop := true
        - condition: "farewell"
          effect text: "Foreign concept to me. They just let me grow here
              ↪ as long as I gobble up the odd undesirable."
          effects:
            _ ""
    - condition: asked_about == 'worship'
      effect text: "Always changing. One lot in, then out, new ones coming
          ↪ up all the time, then driven out or changed beyond recognition.
          ↪  What a palaver."
      effects:
        _ ""
    - condition: asked_about == 'bishop'
      effect text: "He's never passed through here, but he used to get a
          ↪ lot of visitors. You're the first in a long time, mind."
      effects:
        _ ""
    - condition: asked_about == 'farewell'
      effects:
        - current_npc := lisa_npc
```

```
      - lisa := true
      - asked_about := 'nothing'

Lissa:
  condition: current_npc == lisa_npc
  selector:
    - condition: asked_about == 'trade'
      selector:
        - effect text: "Humans throw enough good stuff away in this city
            ↪ that I don't never need to work a day."
          effects:
            _ ""
    - condition: asked_about == 'worship'
      effect text: "Rats don't worship any higher beings. Just feed off
          ↪ their leftovers."
      effects:
        _ ""
    - condition: asked_about == 'lisa'
      effect text: "But I'm right here, boss."
      effects:
        _ ""
    - condition: asked_about == 'bishop'
      selector:
        - condition: bishop_introduced == false
          effect text: "That crusty old fogey? You'll find him moping in
              ↪ his crumbling spire, poring over the cobwebbed leftovers of
              ↪  dead and exiled deities. Follow me."
          effects:
            - follow_lisa := true
            - going_to_npc := bishop_npc
        - condition: bishop_introduced == true
          effects:
            - current_npc := bishop_npc
            - asked_about := 'nothing'
    - condition: asked_about == 'heretics'
```

```
    effect text: "I don't associate well enough with humans to know too
      ↪ many, but if you're looking to meet people from any walk of
      ↪ life, [Zook Spiralhouse] has probably met them."
    effects:
      - zook := true
  - condition: asked_about == 'stenchworks'
    effect text: "It don't smell right, that's for sure. Take that from
      ↪ someone who grew up in the sewers and the garbage."
    effects:
      _ ''
  - condition: asked_about == 'exgods'
    effect text: "Nothing, not the most solid monument, not the most
      ↪ beloved god, nothing doesn't wind up rotting away. Only rats
      ↪ remain."
    effects:
      _ ''
  - condition: asked_about == 'farewell'
    effect text: "You don't want to be wandering the rough 'n' tumble
      ↪ streets of a strange city all by your lonesome, boss."
    effects:
      _ ""
  - condition: asked_about != 'nothing'
    effect text: "I don't really know much about that, boss.."
    effects:
      _ ""

Bishop:
  condition: current_npc == bishop_npc
  selector:
    - condition: asked_about == 'trade'
    selector:
      - effect text: "My trade is [worship]."
        effects:
          _ ""
    - condition: asked_about == 'worship'
```

```
  effect text: "I worship the [ex gods] through proper and exact
     ↪ mourning. Our prayers can no longer reach them; their favour
     ↪ can no longer fall on our actions; their laws are left unjudged
     ↪ . We honour them by correctly lamenting their absence."
  effects:
     - exgods := true
- condition: asked_about == 'lisa'
  effect text: "Forgive me, but such matters are beyond one who dwells
     ↪ in this lonely tower."
  effects:
    _ ""
- condition: asked_about == 'bishop'
  effect text: "Just an old fool with outdated beliefs who they leave
     ↪ around to appease the [heretics]."
  effects:
    - heretics := true
- condition: asked_about == 'nodroth'
  effect text: "Summoned from some dark superplane of existence to
     ↪ topple the city's [ex gods], subjugate its peoples and create a
     ↪  new and unjust order led by an evil priest"
  effects:
    - exgods := true
- condition: asked_about == 'heretics'
  effect text: "I've always been quite the opposite of an iconoclast,
     ↪ but perhaps that's what you need right now. If you can find [
     ↪ Zook Spiralhouse], she's always known people from all walks of
     ↪ life."
  effects:
    - zook := true
- condition: asked_about == 'zook'
  effect text: "They say few live their whole lives here without
     ↪ meeting Zook Spiralhouse once."
  effects:
    _ ''
- condition: asked_about == 'stenchworks'
```

```
      effect text: "When the wind is unfavourable, the odour reaches me
         ↪ even here. What toil produces such foul vapours, I do not know
         ↪ ."
      effects:
        _ ''
    - condition: asked_about == 'exgods'
      effect text: "They may have been bickering, contradictory and
         ↪ impossible, but better that than this single unswerving and
         ↪ terrible unity."
      effects:
        _ ''
    - condition: asked_about == 'farewell'
      effects:
        - asked_about := 'nothing'
        - current_npc := lisa_npc

User Interaction:
  - selector:
      - condition: current_npc == orz_npc
        description: "You there! Hooded stranger! Yes, I can talk. Expect
           ↪ us to waste good firegas raising this portcullis for you, eh?
           ↪  What's your business? [Trade] or [worship]?"
      - condition: current_npc == lisa_npc
        selector:
          - condition: lisa_introduced == false
            sequence:
              - description: "Pssst! The rat queen sent me. She knows who
                 ↪ you are. She knows what you want. And she's on your
                 ↪ side.\nMe? I've been scraping a life on these streets
                 ↪ all my life. Born, plopped straight out onto the
                 ↪ cobbles and left to be raised by the rats. No pocket
                 ↪ can't be picked by these fingers; no rumours what can
                 ↪ escape these ears; no citizen this nose can't sniff out
                 ↪ .\nYou say a name, I'll lead you right to 'em."
              - effects:
```

```
            - lisa_introduced := true
        - description: "Who are we going to see next, boss?"
    - condition: current_npc == bishop_npc
      selector:
        - condition: bishop_introduced == false
          sequence:
            - description: "Oh, child! It is you! After all these years,
                ↪ you return to Zendon an adult.\nAnd not a moment too
                ↪ soon. This city suffers under the crushing rule of [
                ↪ Nodroth]. The favoured classes have grown depraved and
                ↪ cruel. And the ordinary men and women must choose to
                ↪ either poison themselves slaving in the [stenchworks]
                ↪ or languish in the slums."
            - effects:
                - bishop_introduced := true
                - nodroth := true
                - stenchworks := true
        - description: "It is good to see you, child. I hope your just
            ↪ works are proceeding apace."
- selector:
    - condition: follow_lisa
      user action:
        action text: "Follow Lissa"
        effect tree:
          effects:
            - follow_lisa := false
            - current_npc := going_to_npc
            - asked_about := 'nothing'
    - sequence:
        - condition: "trade"
          user action:
            action text: "Trade"
            effect tree:
              effects:
                - asked_about := 'trade'
```

```
- condition: "worship"
  user action:
    action text: "Worship"
    effect tree:
      effects:
        - asked_about := 'worship'
- condition: "bishop"
  user action:
    action text: "Archnecrobishop Tir"
    effect tree:
      effects:
        - asked_about := 'bishop'
- condition: "farewell"
  user action:
    action text: "Farewell"
    effect tree:
      effects:
        - asked_about := 'farewell'
- condition: "lisa"
  user action:
    action text: "Lissa Ratdaughter"
    effect tree:
      effects:
        - asked_about := 'lisa'
- condition: "nodroth"
  user action:
    action text: "Nodroth"
    effect tree:
      effects:
        - asked_about := 'nodroth'
- selector:
    - condition: heretics
      user action:
        action text: "Heretics"
        effect tree:
```

```
            effects:
              - asked_about := 'heretics'
        - effects:
            _ ''
    - selector:
        - condition: stenchworks
          user action:
            action text: "Stenchworks"
            effect tree:
              effects:
                - asked_about := 'stenchworks'
        - effects:
            _ ''
    - selector:
        - condition: exgods
          user action:
            action text: "Ex-gods"
            effect tree:
              effects:
                - asked_about := 'exgods'
        - effects:
            _ ''
    - selector:
        - condition: zook
          user action:
            action text: "Zook Spiralhouse"
            effect tree:
              effects:
                - asked_about := 'zook'
        - effects:
            _ ''
```