Tall Tales: A Game Engine with Natural Language Processing
Final Year Project Report

DT228
BSc in Computer Science

Andrew Tully
Dr. Yupeng Liu

School of Computing
Dublin Institute of Technology

26th March 2015

Abstract

Video games have struggled to represent interactive narrative in a meaningful way. Pen and paper games managed to do this for years. Examples of these include tabletop roleplaying games and Choose Your Own Adventure Games. The advantage that these games have over their digital counterparts is their ability to facilitate player agency and choice. To provide similar facilities, Tall Tales was developed as a game engine with interactive fiction in mind.

Tall Tales utilises natural language processing to accommodate for interactive fiction. The game can get the context of a user's written input and understand what they are attempting to do. When coupled with an expert system, the user's input can be evaluated. This evaluation can update the game world, like in a tabletop roleplaying game.

By integrating this as part of a game engine, an emphasis is placed on interactive narrative. During testing with a demo game, users were excited to see how they could interact with the world and how it would respond. The main limitation was the simplicity of the demo environment and how it does not showcase the system's functionality on a broad scale.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

____

Andrew Tully

26th March 2015

Acknowledgements

Firstly, I would like to thank my parents for providing me with this opportunity in the first place. I would also like to thank my friends for tolerating my absence at parties.

Table of Contents

1. Introduction

Project Overview & Background

Tall Tales is a game engine which using natural language processing to improve player agency and immersion within games. The idea for the project is inspired by tabletop role playing games (RPG). A tabletop RPG is a game in which there is no “winner” or end-goal. Instead, the game aims to provide a cooperative and dynamic storytelling experience. This is achieved by separating the game's players into two groups; the “party” and the “game master” (GM).

The GM is an individual who acts as an impartial judge of the rules, and a storyteller. They control the game world through established rules. They also control the flow of action-reaction within the game.

The party are the protagonists within the game. They dictate what actions they wish to take within the game to the GM. The GM determines if the action is plausible. By using an appropriate rule they determine whether the action fails or succeeds, and what the associated consequences are.

During a tabletop RPG, the rules and story can be adjusted as needed by the players. This is because here is no autonomous system in place determining the rules and pacing. This is another difference between RPG and traditional video games. This allows the game to adjust the story to unexpected outcomes. It allows for many more actions to be taken within the game through manipulation of the rules, or storytelling.

Project Objectives

The most important goal for this project is to ensure that it provides an ergonomic workflow for developers and is easy to use. The project must provide an interface for developers to incorporate natural language processing (NLP). This will be accomplished by abstracting the NLP functionality.

Project Risks

The largest risk associated with this project is whether it will be comparable to other game engines. Most industry standard game engines provide a robust suite of tools. Examples of which are; an editor, lighting utilities, physics utilities, an interface for scripting and the ability to render both 2D and 3D objects. These kinds of features are almost expected of a modern game engine. It is difficult to implement these along with meeting the project goals given the current time frame of the project.

Implementing NLP is a risk unto itself. By it's nature, NLP is not completely accurate. Incorrect language processing within the game could result in odd or outright incorrect output. This would break player agency and immersion.

Developing to an interface and not an implementation will be difficult to adhere to. This is because I will be developing a demo game alongside the engine. This could result in unintentionally making changes in the engine just to fit a specific use case of the demo game.

Risk Mitigation

To mitigate these risks several precautionary measures were taken. To manage the scope of a game engine, a set of core modules are established. These include the NLP system, the Knowledge Representation, and the Game Initialisation & Management. This guarantees that the project has a well-defined set of goals, and a reasonable workload for the given time frame. In the event of being ahead of schedule, additional components can be added to the project.

To reduce the risk of performing NLP, the scope of it's functionality is limited. This is to ensure that some form of functioning NLP system can is delivered on time, but can be extended if possible.

Unfortunately, there is no apparent way to ensure that I develop to an interface rather than an implementation. This is an accepted risk, which I hope to catch through rigorous testing.

2. Research

Researching Interactive Narrative: Problems & Potential Solutions

To determine a solution, it was imperative to research the architecture of game engines, and how narrative is represented in games.

To begin this research, I looked into what are considered to be the core elements of interactive storytelling, as according to Rational Games, a Swedish developer. Rational Games defined five different key elements; 1) Focus on Storytelling; 2) Spending Time Playing; 3) Maintaining Narrative Sense through Actions; 4) Minimise repetition; 5) No Major Progress Blocks (1). The first element identified is that the focus must be on storytelling, rather than the specific gameplay mechanics such as “puzzles, stacking gems or shooting moving targets” (1). One could argue that a game without a focus on gameplay is redundant, but I feel that the game needs to strike a balance between the two. While gameplay cannot be complete abandoned in favour of story, a good story can carry weak gameplay. This does not directly impact Tall Tales in a significant manner. Tall Tales does not provide an interface for story creation. While it would be possible to create an interface for crafting stories, like Twine (2) (which will be discussed later in the **“Research of Potential Technologies”** section), it is not necessary to the current scope of the project.

The second element is that “the core of the game should not be about reading or watching cut-scenes, it should be about playing” (1). While this is an important tenant, it is not exactly relevant to the development of a game engine. How the game is conducted and played is a decision of the game developers.

"Maintaining Narrative Sense through Actions" is the most important element that Tall Tales can help to reinforce. Rational Games define that the focus should be devoted more to the story than the gameplay. They also reinforce that it is important for the gameplay to be of value to the story. They establish that players must feel like an active participant in the story. This is achieved having gameplay actions which are closely tied with the progression of the story. This affords the player a great degree of agency. (1) Tall Tales manages to do this through supporting text-based NLP. Tall Tales allows the user to type their desired action with no limitation. This is offers more control than if they are told to choose one of three options. The use of text-based input means that a wide variety of potential actions can be supported to progress the story. The only issue is how well the individual developers will use these utilities to craft “important story moments”. (1)

Minimising Repetition is a difficult element to implement. By their nature, most games will have a set of core mechanics which are repetitious. Within the Tall Tales engine, the primary driving force of the NLP is being able to input text. The action of inputting text is incredibly repetitious. The only way this can be broken up is by the developer implementing additional mechanics into their game. However, Rational Games make an excellent point; “This does not mean that the core mechanics must constantly change, it just means that there must be variation on how they are used”. They even draw attention to games such as Limbo and Braid, where the core mechanics are simple, but must be used in increasingly different scenarios throughout the game. (1) This reinforces the versatility in text-based input as a driving force in gameplay. While the act of typing desired actions is repetitious, it offers a lot of scope for depth and diversity. During testing it was found that players of the demo game were not bothered by having to type in their desired action. Instead, they were excited to repeat the same process to see how the game would react to their input.

The final core element is avoiding major blocks to progression. Traditionally, games attempting to tell a story fell into the trap of “obscure puzzles, mastery-demanding sections and maze-like environments” (1) These would impede player progress and destroy the pacing of the story. To a degree, this is something which the developer must try to combat, but it is also a potential pitfall of Tall Tales. Previously, text-driven games have required the user to input specific words in order to progress past a particular section. This causes the user to repeat input until the desired keyword is found. By using NLP, I hope to combat this through the use of synonyms. This would put more of the onus on the developer to avoid major progression blocks.

Other papers which covered narrative representation in games were examined. This was necessary to gain a comprehensive view on the topic. One such paper is from Marie-Laure Ryan, a writer with several published articles on narratology. In one of her papers Ryan outlines three features which she feels are essential to game narrative; a natural interface, integration of user action and dynamic story creation. (3)

Ryan's paper bears some similarity to the article written by Rational Games; namely the integration of user action into the experience and narrative. By providing a natural interface for the user to communicate their desired actions they will experience a greater sense of immersion. Normally when video games offer the users narrative choices the choices take the form of a list of options. One of the most natural interfaces that could be incorporated is voice recognition. However, voice recognition is complicated and tends to suffer from it's own inaccuracies. This problem is exacerbated when incorporated with NLP due to NLP's own inaccuracies at times. As a result of this, Tall Tales only supports text input. I feel that this is the most natural interface, second to voice communication.

Ryan believes we need to provide physical actions which impact the world, and verbal actions which impact the inhabitants of the world in order to provide narrative richness and player immersion. (3) By having the player's actions directly impact the development of the game's narrative they afford a sense of agency. This allows the player to become more invested in the game. To attempt to facilitate this, Tall Tales uses an expert system in conjunction with the NLP system. This allows the developer to define a set of facts and rules about the game world. These can be changed and evaluated based on the actions the user takes. This will help create a dynamic environment for the player that can be influenced by their actions.

Dynamically creating a story is the most difficult feature to include, and is linked to integrating user actions into the story. There are two types of interactive narrative; emergent and top-down. Emergent story is a result of computing the effects of the user's actions and amending the model of the game world before responding to the player. Top-down relies on pre-scripted content that the player can progress through. (3) Top-down story is easy to create as the developer has complete control over it. The biggest downside to this is that it generally limits the interactivity and agency of the player. This reduces the replay value of the game. By having an emergent story the player can have a completely new experience each time they place. It will be almost impossible to guarantee that the same thing happens twice. Emergent story becomes relatively easy to implement in an environment with thousands of simultaneous players that have a large degree of freedom over what actions they can take. In an environment where there is only a small number of players emergent stories become reliant on robust AI systems. By adding expert system support I hope to help tackle this issue and make the development of emergent story easier.

To better understand these principals in action, I examined two games which rely on text input to progress the game and its story; Façade and Zork. Zork is an old text adventure game from the late 70s, while Façade is a more modern take on text adventure games.

Zork is a basic game which just presents the user with an input box for text and an area to display output text. It then presents the user with a scene, along with a set of cues for interacting with the environment. Zork fails to meet the key elements of interactive narrative presented by Rational Games. Zork relies on keyword spotting. During any specific scene in the game, there are a set of keywords that the game is looking for. These relate to a specific object and produce a specific result when invoked. This leads to frustration for players as they spend more time figuring out the correct keyword rather than the game interpreting what they want to do.

This hinders player progression through the game and detracts from the storytelling. The user becomes stuck on finding the exact word that the game is looking for at a particular moment. The user has to repeat input on the same object in order to progress. This also puts a strain on the progressing the story. Narrative sense is not preserved as legitimate actions are prevented. In terms of Ryan's principals, the game does provide a natural interface through text input. Although, it fails to have the user's actions impact the narrative in a significant way or present a dynamic story. The user does not remain a passive participant, but instead actively progresses the narrative. The only issue is that this impact the story is not particularly significant. The user only progresses the story because their options are limited. The game becomes more about puzzle solving. This is directly linked to the lack of dynamic story. In any given scenario there are only a few set actions the player can perform which will have the same result. They also fail to change the course of events in the game.

Façade is a more modern text driven game which was developed as an experiment for narrative driven by artificial intelligence. (4) During language processing the game attempts to figure out the context of the user's input and then progresses the game's narrative based off of that. (4)

Façade is more difficult to evaluate in terms of the previous established principals. The game maintains a focus on storytelling and manages to avoid repetition. There are no major progress blocks, but this is due to the narrative progressing without the player. The player can opt not to perform any actions and the story will progress, or they can influence the events occurring. This allows the players action, or inaction, to directly control the narrative, while avoiding major progression blocks. Unfortunately, Façade struggles to maintain narrative sense. This is as a direct result of the NLP. During my research I found that most of the time the game would fail to recognise my input and the narrative would just continue regardless. For the most part, the game's scenario puts you in a situation where there is no narrative sense in taking action.

After researching both Zork and Façade I have established that the following are important features for Tall Tales: comprehensive NLP that reduces the number of attempted inputs to perform a particular action; an interface for using expert systems to create dynamic narratives; facilitate text input in order to perform in-game actions.

Natural Language Processing is the computational analysis of human languages, such as English and is in the most crucial part of the project. This is a difficult field, as natural languages do not have easily identifiable explicit rules. (5) For my project I researched the primary areas of NLP that I will be using; part-of-speech tagging, WordNet & FrameNet.

Part-of-speech (POS) tagging is the processing of identifying a word's world class or lexical category (noun, verb, adverb, adjective etc.). (5) WordNet is a lexical database of English words which are linked from synonym-to-synonym (known as synsets) using semantic relations. (6) FrameNet is another lexical database which attempts to create semantics relationships with words by contextualising through a structured background experience, or definition. (7) For example, FrameNet categorises words into frames, such as the “Theft” frame. This frame is then triggered by semantically related verbs such as “steal”, “snatch”, “thieve”. Each frame then a set of elements that help to encapsulate the frame. These are; Agent, Undergoer and Instrument. (8) The benefit of FrameNet is that it allows for more than one subject to be taken into account, if the frame allows it. For example, two people could be Agents in the Theft frame, or two objects could undergo theft. Additionally, it provides a way to interface natural language with programming as it can parametrise verbs.

In order to perform the POS tagging, it is necessary to use multiple N-gram taggers. To tag the user input I use three N-Gram taggers; a unigram tagger, a bigram tagger and a trigram tagger. The difference between the taggers is the context they use for tagging a word. The unigram tagger examines the current word on its own, the bigram tagger examines the word in the context of the previous word, and the trigram examines the word in the context of the previous two words. (5) In order to make the analysis comprehensive, the taggers are chained together. If the trigram tagger fails to tag a word it gets passed off to the bigram tagger. If that fails the word is passed off to the unigram tagger, and then to the default tagger if that fails. Wordnet will be essential for error handling so that if a particular word fails, it's synonym can be checked.

A game engine is a set of classes and components, used in a game, that do not contain any logic or data specific to a particular game. This helps reuse and quickens the development time of any particular game. (9) While researching the most suitable architecture for a game engine, I found that the general consensus was that an Entity

Component System (ECS) is considered to be a more effective architecture for game engines, as opposed to traditional Object-Oriented (OO) approaches.

The benefit of an OO approach is that it promotes code reuse and organisation. The biggest issues that this has within a game is that the hierarchy is not flexible. It is difficult to add features to a few specific classes without code reuse, or adding it to a shared parent class, which results in the code be unnecessarily propagated to other classes. (9) Another huge problem with an OO approach is that it's hierarchy system does not promote expandability. Each child has a heavy reliance on it's parents. This makes adding or changing features difficult without having to reconstruct a lot of the code. (10) This issue impacts game development more than traditional software systems as features are constantly added and retracted during development.

An ECS architecture offers many benefits over an OO approach. ECS makes it easier to conceptualise each individual entity as a composition of features. Constructing an entity becomes much easier as it is simply just adding or removing features without any major code reconstruction, and classes become smaller and easier to maintain. (9) (10) In ECS, a database-like approach is taken. Logic is decoupled from data. Components become collections of related properties, which are then managed by a relevant system. This results in major benefits such as a high degree of cohesion and a low degree of coupling within the engine. (9)

ECS does have its disadvantages. Some people argue that the isolation and modularity of each feature can serve as a downside due to the fact that it complicates cross-system communication. (10) This can be accomplished by using an external manager which can be seen Cupcake and Artemis Framework (which will be discussed later). However, this leads to increased overheard.

Pure ECS isn't always a common occurrence. A pure ECS system assumes that all of an entities properties will be contained within its components. (9) This is not necessarily a bad thing, but it can be incredibly limiting. A solution would be to include an abstract component and system which can be sub-classed to create tailor-made systems and components. This is a step towards a hybrid ECS model in which it is assumed that there are certain properties and methods that all entities will share at some point.

There are multiple ECS architectures. One proposed implementation I found throughout my research was forgoing the creation of an actual entity class. Instead, whenever a logical entity is created, the relevant components are instantiated and is given an ID which represents what entity it belongs to. (9) Two other approaches are using the Cupcake and Artemis frameworks.

The Cupcake ECS architecture is designed to be a completely modular user-defined game engine. The system relies on highly independent modules which can be combined into a customisable user engine. Upon start up, the user can choose from a list of plugins to include which will manage the engine's functionality. This results in a game engine customised for the end user. (10) Cupcake offers some interesting solutions to potential efficiency problems. In Cupcake, systems are separated into two lists; processing and idle. Processing systems are updated at each game loops, while idle systems aren't updated unless moved into processing. (10) In order to solve the shared components problem, Cupcake maintains a set of components outside the rest of the game engine which are passed to a system upon creation. This helps to maintain system independence, and avoids the need to sync the systems with the shared components. The cross system communication problem is solved by using an external messaging system. If the appropriate trigger is set off, then the message is sent to the messaging system specifying the appropriate course of action. (10) This solution comes with some major downsides, such as reducing the level of decoupling in the architecture while also increasing the amount of overhead required. By having messaging integrated as part of the engine, it could possibly improve the feasibility of the messaging system. Additionally, due to components being stored within the actual systems it becomes much more difficult to remove entities from the systems. (10)

Artemis is an ECS framework developed in Java that treats entities as numeric Ids rather than classes containing data. (10) There is a major downside with numeric Ids, and that is that there is a limit on the amount of Ids for components that can be created. Artemis uses an Entity Manager to track a list of Ids and associated components. This Entity Manager is essential in attempting the solve the shared component problem. By attaching all components to the Entity Manager all systems have access to all components, and no syncing is required. (10) Unfortunately, Artemis does not have a realistic solution for cross-system communication as the only object shared between systems are components, and creating/deleting components is costly.

For this project I feel that booth Cupcake and Artemis do not quite satisfy the requirements. Instead, I will opt for a hybrid ECS model. This assumes that all components and systems will share certain core functionality.

Research of Potential Technologies

In order to select what technologies to do, I evaluated them based on the following criteria; ease of handling strings, support for knowledge representation, support for game development and support for natural language processing.

Adrift

Adrift is a GUI driven application for easily producing interactive fiction games. (11) Adrift allows users to rapidly develop interactive fiction games, however they are extremely limited. Adrift allows you to define connected nodes on a graph which contain text. These nodes are traversed by typing in the appropriate command (“Look, Take etc.”).

This does not meet many of the project requirements. The requirement it does manage to meet is that it allows a user to rapidly develop an interactive fiction game. Unfortunately there are many downsides. The text recognition is only keyword spotting. Keyword spotting is not robust enough to fully immerse the player in the game world. This is due to the disconnect between what the user enters, and what is actually recognised. Secondly, the nodes are extremely static and cannot be updated as the game progresses. In order to “update” a node, there must be a similar node with slightly different text that is reached by a different path. Adrift also does not allow for customisation by the user developing the interactive fiction. It is not possible to define new game elements within the engine, such as physics, 2D/3D models etc.. This results in Adrift only being able to handle simple text input and output.

Fungus

Fungus is a Unity extension for creating story-based games. (12) Fungus allows Unity developers to easily add a focus on story. By doing this, it expands upon one of the problems presented by Adrift. It allows the developer to incorporate more complex gameplay elements that progress the story.

Like Adrift, Fungus meets the requirement of allowing developers to create story-based games, with the added bonus of being able to include other gameplay features. However, fails to include natural language processing; a key aspect of the project problem. Unlike Adrift, it does not allow the user to input text. Instead, the user is presented with a set of choices which they can choose from that will progress the story.

This breaks immersion even more than Adrift's mono-verbal user input. The player is not even able to accurately verbalise their exact intentions. Instead, the user needs to find an option which best fits how they want to approach the situation. This leads to confusion as players and the developer may interpret the option in different ways.

Twine

Twine is an open-source tool for developing interactive stories. (13) Twine is an open-source tool for developing interactive stories. (13) Twine is almost identical to Adrift, except for the fact that it presents the same problem that Fungus does. It lacks written user input, only selection from a set of options. This is the worst of both worlds as it doesn't even incorporate keyword spotting, or allow a large degree of customisation. The main benefit of Twine is that it is entirely open-source. This could mean several major improvements and interesting/useful packages being developed for it.

Python

Python was selected due to how easily it can handle strings and indexed lists, which will be the main data types used in the project. Additionally, it is the only language that NLTK is compatible with.

I have a small amount of Python experience prior to this year. My only experience comes from developing a RESTful API using Flask. This will prove useful as it gave me experience in developing an API. Additionally, we used a Bootstrap GUI with Flask. The experience with adding a GUI in Python will be useful if there is time to add a GUI to the project. Flask will also be useful if any networking is required.

In order to further familiarise myself with Python I read through the API documentation that concerned any functionality which may be relevant to my project.

Natural Language Toolkit (NLTK)

The Natural Language Toolkit is an interface for various lexical resources and libraries such as WordNet and FrameNet (14)

There was not a lot of criteria to consider when selecting this particular technology. It is the largest comprehensive resource for natural language processing. It is also developed using Python which makes it a perfect fit into my project.

I have no prior experience with NLTK. To familiarise myself with this I read 'Natural Language Processing with Python' by Stephen Bird. It written particularly for NLTK. I also got hands-on experience by developing a simple proof of concept prototype for the project.

Pyke

Pyke is a Prolog-inspired knowledge engine for Python. (15)

Initially, I was planning on using the rudimentary knowledge representation provided by NLTK. However, this might not fully capture the use cases required, and as a result could require a lot more work to be effective. This would be extremely time consuming and detract from the main goal of the project. Instead I settled on using Pyke as it already provides a suite for forward and backward chaining goals, and a complete framework for creating a logical model.

To familiarise myself with the library I read through the API documentation and got hands-on experience by developing a simple proof of concept prototype for the project.

PyGame

PyGame is a game framework for Python that will be used to draw renderable entities to the screen. (16)

This was library was a necessity for the project as it is the most versatile and widely-used Python game framework. Implementing the features provided by PyGame would be quite difficult and time consuming. PyGame was selected over Pyglet as it has been around longer, resulting in better documentation and a larger community.

Pyglet

PyGlet is an open-source game framework for Python. (17)

Pyglet provides much of the same functionality as PyGame as is often considered to be more “Pythonic”. Unfortunately, Pyglet hasn't had any new builds released in two years, and has a very small, inactive community. This renders it unsuitable to use for the project.

Love2D

Love2D is a Lua-based game development framework. (18)

The reasons for not selecting this over the previous technologies was because I am more familiar with Python than Lua. There is also a lot more support for natural language processing and logical model representation in Python.

Unity

Unity is a game engine which supports rapid 2D and 3D development. (19)

Unity is easily one of the most robust and widely supported game engines with a massive community. It solves the problem of developing a comprehensive game engine,but is poor at handling knowledge representation and natural language processing. Additionally, C#/JavaScript/Boo (all Unity scripting languages) and Unity do not contain libraries for natural language processing, unlike Python.

3. Design

Design Methodology

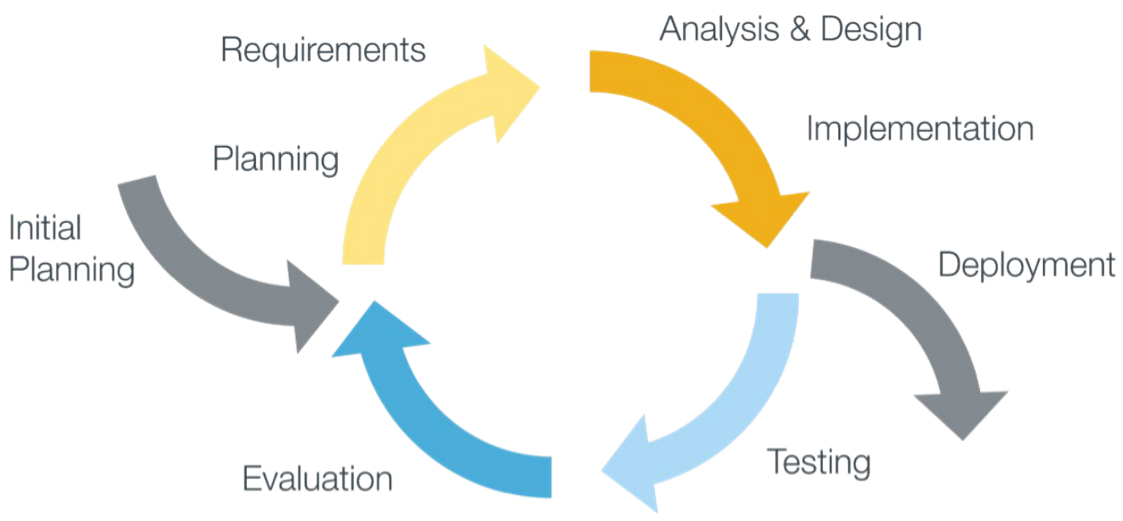


Figure 3.1: Design Methodology

The approach I have chosen for the project is an iterative design methodology. Iterative design is a widely used design philosophy which involves a cycle of design, creation, testing and redesign. A system within the program is chosen, designed in detailed, created and then tested before redesigning it, if necessary. This allows for the developer to catch design flaws early on in development, rather than when every system is fully implemented. (20) The main reason for adopting this methodology is that it allows for the project to be adapted in response to user feedback and unplanned setbacks (such as API conflicts, hardware issues etc.). The ability to adapt to user feedback is key in software development, especially within game and game engine development. By isolating particular features of the project when developing them, they can be carefully tested in a vacuum before integration occurs.

Component Design

The central concept with each of these components is that they are to be subclassed when used. For example, in order to create a new game you need a new instance of the game engine. That engine will contain instances of the NLP and Knowledge systems which will handle their relevant game entities.

NLP System

The premise behind the NLP system is relatively straightforward. The input is tokenised and then POS tagged. Then the find the first verb in the sentence is found and recorded. Once the verb is found, the first noun which occurs after the recorded verb is recorded. These can then be used with the knowledge system.

If there is no verb found, the first token of the input is checked. If the first token is a personal preposition (“I”, “You”, “They” etc.) then it's assumed that the second token is a verb. Otherwise, it's assumed that the first token is a verb.

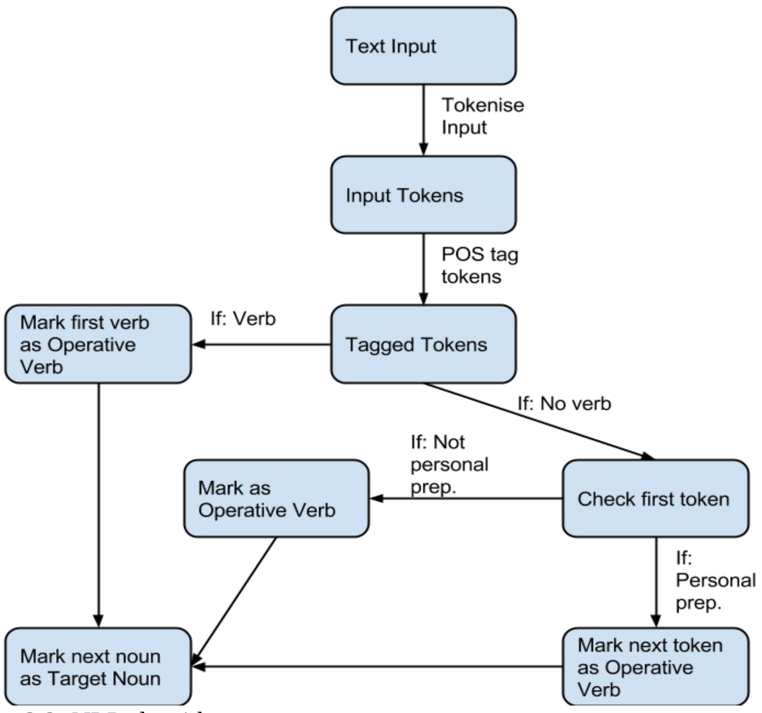


Figure 3.2: NLP algorithm

Knowledge System

The Knowledge System utilises Pyke rule bases in order to evaluate whether a user's action succeeds or not. The developer defines these rule bases for their specific use case. When used, the Knowledge System checks if the named rule base exists, and then if the named rule exists within that rule base. If there is a match, it attempts to evaluate the rule and will return whether or not it was successful. This can be used in conjunction with the Knowledge System where the Target Noun is used to check for a rule base while the Operative Verb is used as the rule.

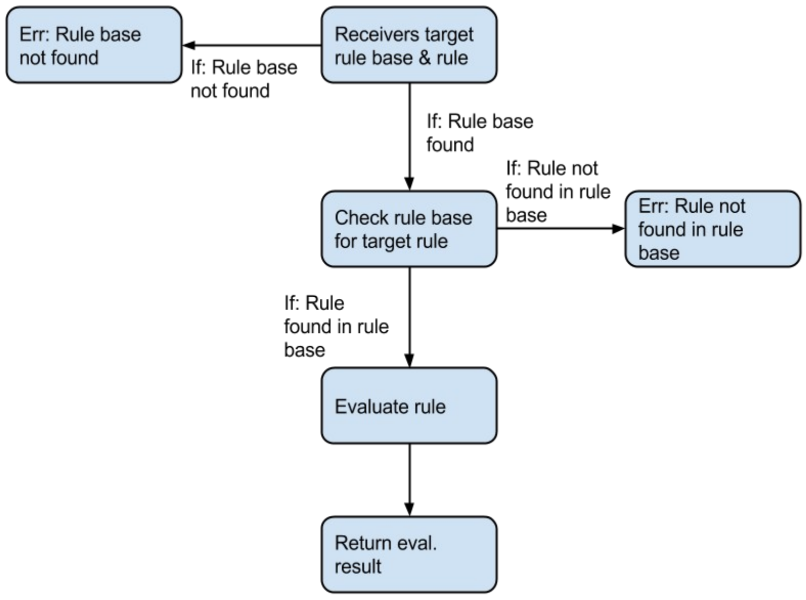

Receivers target rule base & rule
If: Rule base not found
Err: Rule base not found
If: Rule base found
Check rule base for target rule
If: Rule not found in rule base
Err: Rule not found in rule base
If: Rule found in rule base
Evaluate rule
Return eval. result

Figure 3.3: Rule evaluation algorithm

<u>Scene</u>

The Scene abstract class defines a set of methods that are useful for developing and managing game scenes. This class is designed to be subclassed. Once subclassed, the Scene can then be populated with entities and systems.

The Scene updates all its Systems and contains an interface for adding entities and systems to the scene. It also allows the developer to define conditions for transitioning from one scene to another

<u>Entity</u>

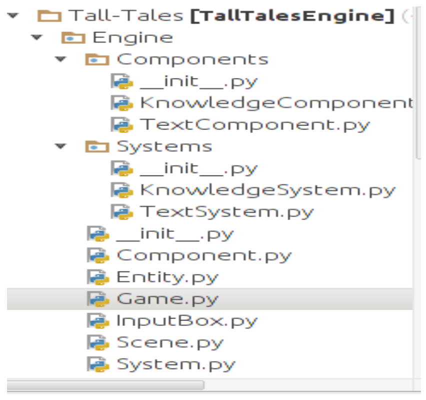
The Entity abstract class defines a set of methods that are useful for developing and managing game entities. This class is designed to be subclassed. Once subclassed, the Entity can be populated. Additionally, it is possible to check if an entity has a particular type of component (E.G: text component, knowledge component etc.).

<u>Game Engine</u>

The game engine is designed to be subclassed. This allows flexibility for the developer to fit it to specific use cases. To begin development, the game engine must be subclassed into a new Game object. Then, the Scene template class must be subclassed in order to create a new scene. This may be further subclassed in order to provide more specific temples (E.G: creating a Town scene which defines generic features of a town, which can then be subclassed again into individual towns). Once a scene is created in can then be set as the initial active scene in the game.

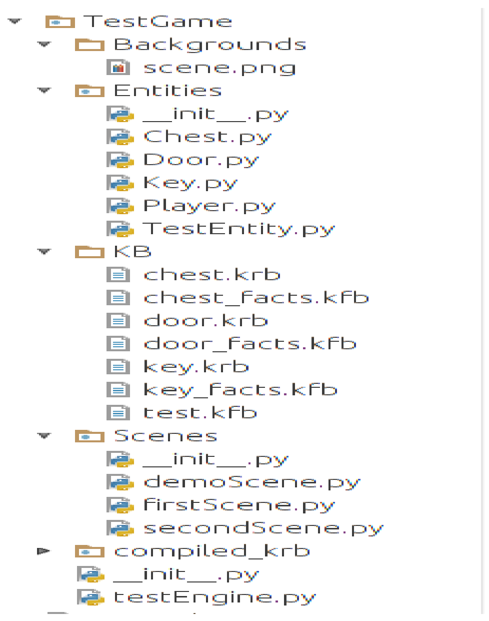
An empty scene isn't very useful, however. Once a scene has been created, the developer can subclass the engine's systems and add them to the scene. Developers can also create entities in a similar manor to scenes. The Entity super-type is subclassed. Within this subclassed Entity instances of various game engine components are added. These entities can then be added to the scene where a system will manage the relevant components (E.G: If an entity has a Text Component the Text System will manage any text-related attributes for that entity). This allows for a high degree of modularity and customisation for developers.

<u>Source Code Layout</u>



```
Tall-Tales [TallTalesEngine]
  Engine
    Components
      __init__.py
      KnowledgeComponent
      TextComponent.py
    Systems
      __init__.py
      KnowledgeSystem.py
      TextSystem.py
    __init__.py
    Component.py
    Entity.py
    Game.py
    InputBox.py
    Scene.py
    System.py
```

Figure 3.4: Engine source code



```
TestGame
  Backgrounds
    scene.png
  Entities
    __init__.py
    Chest.py
    Door.py
    Key.py
    Player.py
    TestEntity.py
  KB
    chest.krb
    chest_facts.kfb
    door.krb
    door_facts.kfb
    key.krb
    key_facts.kfb
    test.kfb
  Scenes
    __init__.py
    demoScene.py
    firstScene.py
    secondScene.py
  compiled_krb
  __init__.py
  testEngine.py
```

Figure 3.5: Demo game source code

Project Features and Use Cases

Project Features

- Interface for NLP
- Interface for rule-based expert systems using Pyke
- Interface for using PyGame, a 2D SDL wrapper
- Established development workflow
- Flexible and customisable templates for defining new game engine systems.

Project Use Cases

- Games which utilise NLP
- Games which utilise expert systems
- Python-based game development
- 2D game development
- Text-driven game development through the use of NLP
- Development of dynamic game worlds through the use of expert systems

4. Architecture and Development

Overview of System Architecture

The architecture is relatively standard as far as game engines go; an engine object contains a collection of scenes. These scenes contain various systems and entities. Entities are aggregates of components, with system that manage their relevant components on a per-entity basis. During development, a developer can add whatever systems and components they feel are necessary, without needing to include any in particular.

Figure 4.1 shows not only the architecture, but the workflow for the game engine. All follows this basic structure. Systems manages related components. A game is controlled by a game engine. This game engine has a collection of scenes. Each scene has a collection of systems and entities. Each entity has a collection of components.

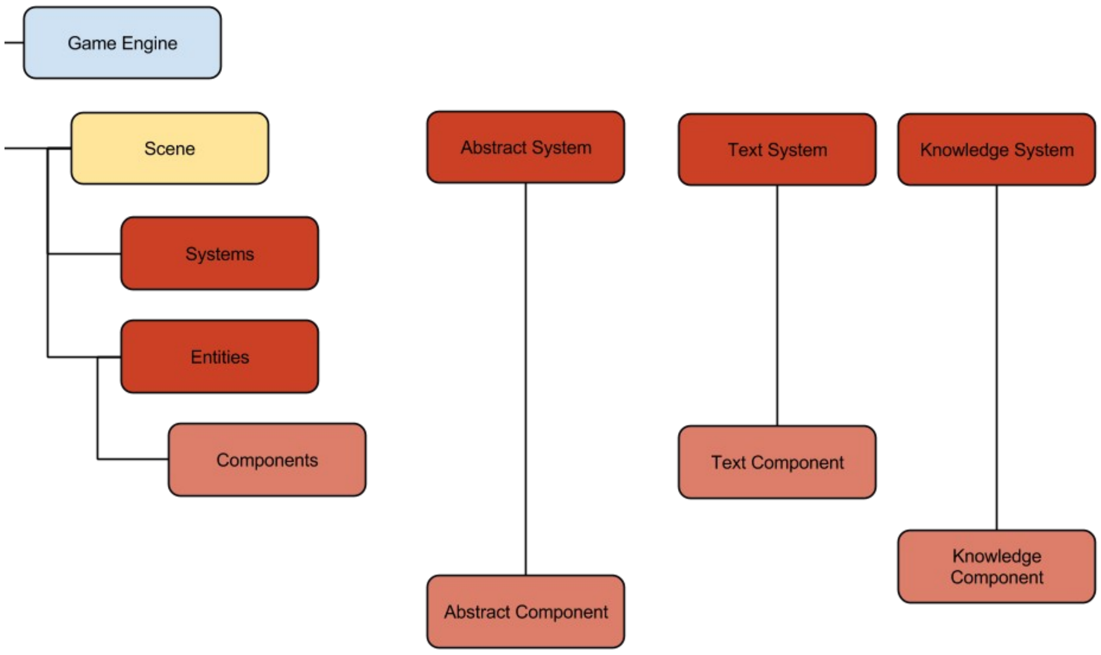


Figure 4.1: System Architecture

Development & Problem Resolution of the Game Engine

Entity

The Entity class is a template which allows developers to create entities. The Entity class defines that all entities have a componentDict and contain methods for adding, removing and checking components. The key of the componentDict represents the type of component (E.g.: text, knowledge etc.). The value for that key is a list of all components of that type which belong to the entity.

The add_components() method accepts a key, and a list of components. It then attempts to add the components to the componentDict at the given index, which is given by the key. The remove_components() method accepts the same parameters. Instead it attempts to remove the components for a given key, if the key exists within the componentDict.

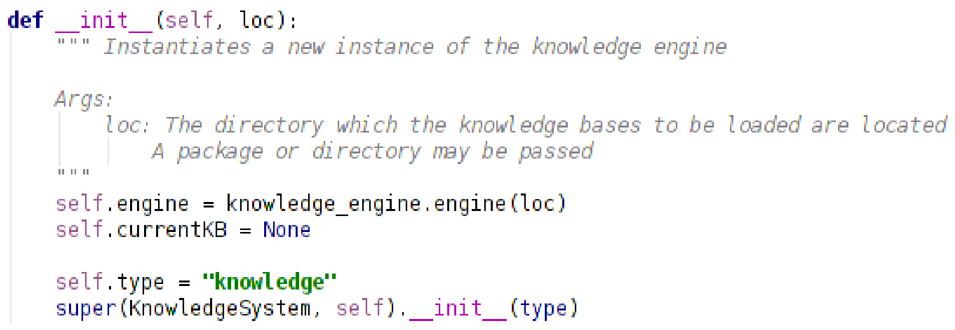
The does_contain() method accepts a key and checks if the key is contained within the componentDict. This helps to determine whether or not the entity contains a component of a particular type and is used by the System class.

System

The System class is a template which is used to define specific types of game systems. This defines that all Systems have an entityList and a check_entity() method. The check_entity() method accepts an entity and and entity type (E.G: knowledge, text etc.) and checks whether the entity contains a component of the given type. If it does, this entity is added to the System's entityList, with a key to denote it's priority. By default, priority is assigned in the order that the entities are added to the system. This allows the system to manage the it's relevant entities.

Knowledge System

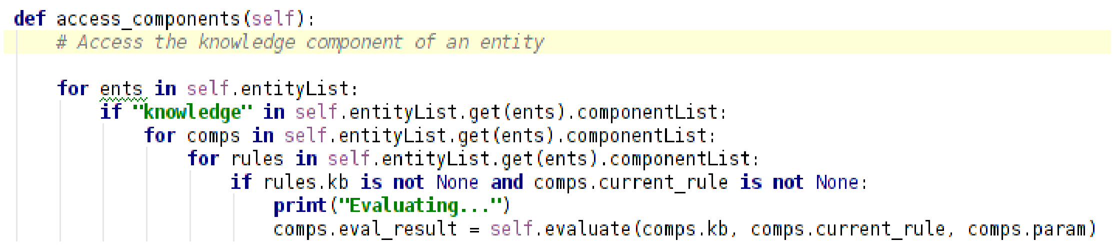
In order to function, the Knowledge System contains three methods; access_components(), update() and evaluate(). When a new Knowledge System is initialised, it accepts a directory for all the fact and rule bases that will be used by the knowledge engine.



```
def __init__(self, loc):
    """ Instantiates a new instance of the knowledge engine

    Args:
        loc: The directory which the knowledge bases to be loaded are located
            A package or directory may be passed
    """
    self.engine = knowledge_engine.engine(loc)
    self.currentKB = None

    self.type = "knowledge"
    super(KnowledgeSystem, self).__init__(type)
```

Figure 4.2: __init__() method of the Knowledge System

The access_components() method will iterate through each entity in the system's entityList. It attempt to evaluate them based on their currently attributed rule and rule base.



```
def access_components(self):
    # Access the knowledge component of an entity

    for ents in self.entityList:
        if "knowledge" in self.entityList.get(ents).componentList:
            for comps in self.entityList.get(ents).componentList:
                for rules in self.entityList.get(ents).componentList:
                    if rules.kb is not None and comps.current_rule is not None:
                        print("Evaluating...")
                        comps.eval_result = self.evaluate(comps.kb, comps.current_rule, comps.param)
```

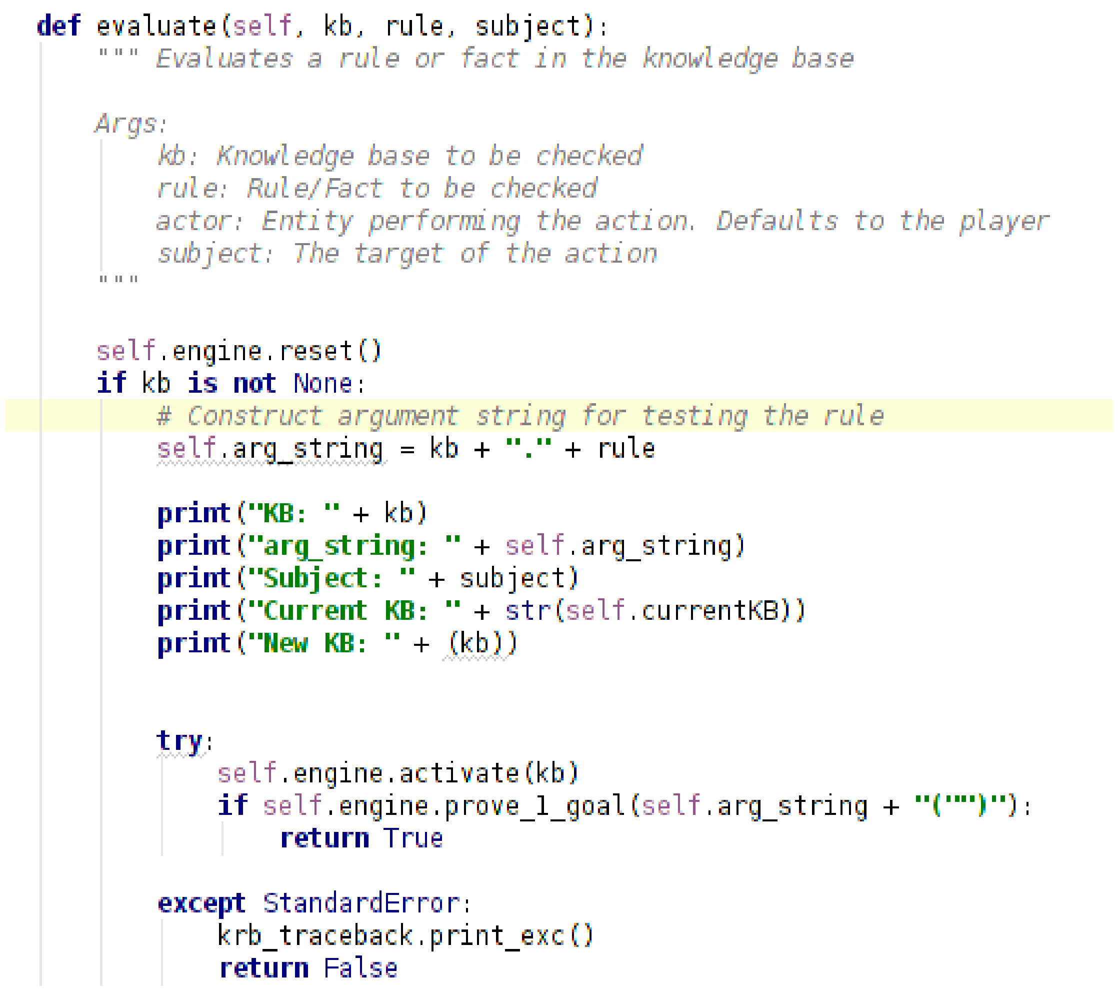
Figure 4.3: access_components() method of the Knowledge System

The update() method for the Knowledge System doesn't perform any action. This is because evaluating a rule within a knowledge base is relatively costly, and is not necessary at each update cycle in the engine.

The evaluate() method accepts a knowledge base, a rule and a subject. The subject is often the same as the knowledge base. This method is used to determine whether or not a particular entity can be interacted with, and whether the interaction is valid. This is done by using the knowledge engine which is instantiated when the instance of the Knowledge System is created.

To begin evaluation, the engine is reset. This deactivates the currently active rule base. Prior to this change, if a player interacted with an object (I.E: loaded the object's rule base) the first evaluation attempt would succeed, but all other subsequent evaluation attempts would fail. The knowledge engine could not activate a rule base that was already active. Additionally, various checks to prevent to engine from activating the same rule base twice did nothing to prevent to error. It also led to issues where the engine would try to activate increasingly nested rule bases that didn't exist. For example, “Kick the chest” would run the “kick” rule from the “chest” rule base. This would succeed. A subsequent rule, such as “Open the chest” should run the “open” rule in the “chest” rule base, but would instead try to run a blank rule within the “chest.open” rule base.

After resetting the knowledge engine, the evaluate() method sculpts its input into a format that the Pyke API can understand. This is used for accessing rule bases. To activate the rule of a rule base in Pyke it must be in the format: “rule_base.rule”. The evaluate() method takes the rule base and rule it was given when called and joins them using a “.” to delineate the two individual attributes. This newly formed string is used as the input. Once this is done, it attempts to activate the rule base and evaluate the rule.



```
def evaluate(self, kb, rule, subject):
    """ Evaluates a rule or fact in the knowledge base

    Args:
        kb: Knowledge base to be checked
        rule: Rule/Fact to be checked
        actor: Entity performing the action. Defaults to the player
        subject: The target of the action
    """

    self.engine.reset()
    if kb is not None:
        # Construct argument string for testing the rule
        self.arg_string = kb + "." + rule

        print("KB: " + kb)
        print("arg_string: " + self.arg_string)
        print("Subject: " + subject)
        print("Current KB: " + str(self.currentKB))
        print("New KB: " + (kb))


        try:
            self.engine.activate(kb)
            if self.engine.prove_1_goal(self.arg_string + "("")"):
                return True

        except StandardError:
            krb_traceback.print_exc()
            return False
```

Figure 4.4: evaluate() method from the KnowledgeSystem class

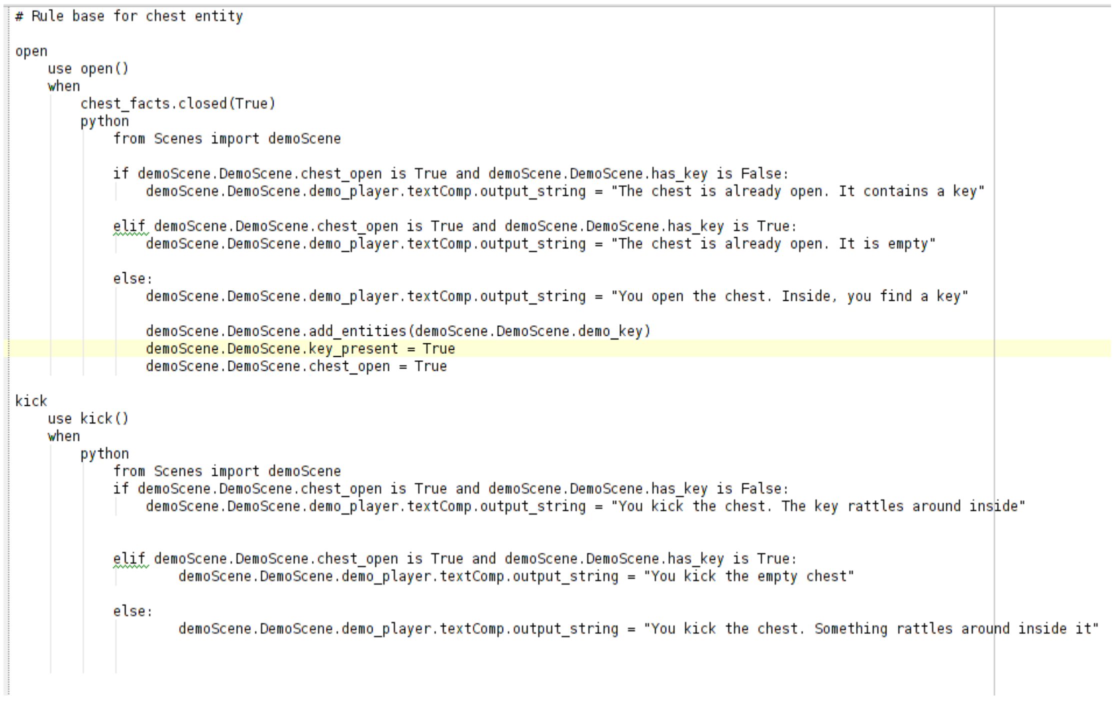
Rule Base Formatting

The rule bases must be formatted in a particular way in order for them to give the desired output. This is due to problems with the Pyke API. Upon initial inspection, and from developing a proof of concept, Pyke seemed like the perfect solution, but caused more problems than it solved.

All rules within the rule base must be backward chaining rules. This is because any forward chaining rules that are present are evaluated and fired upon activation of the rule base. This leads to another problem; forward chaining rules are able to assert new facts, but backward chaining rules are unable to assert new facts. The result of this is that these facts now need to be represented as attributes within the scene in order to facilitate facts about entities in the scene changing.

Normally, Python calls are required within a rule in order to return the string describing the result of the rule evaluation, but because the majority of the facts are represented as attributes in the scene, the majority of the rule becomes one big Python call. This makes the use of an expert system almost completely redundant.

Rule bases must also be named after the object that they are representing. For example, the rule base for a door must be called door.krb otherwise the Knowledge System won't find it. This could have been fixed by using a dictionary to hold the rule bases, and using a string as the key. If no match is found, then the key could be checked against it's synonyms using the Text System. Due to time constraints I did not have enough time to figure out an efficient method of doing this while maintaining the ECS architecture. Having the Text System directly interact with the Knowledge System would violate the architecture by creating cross-system dependencies.

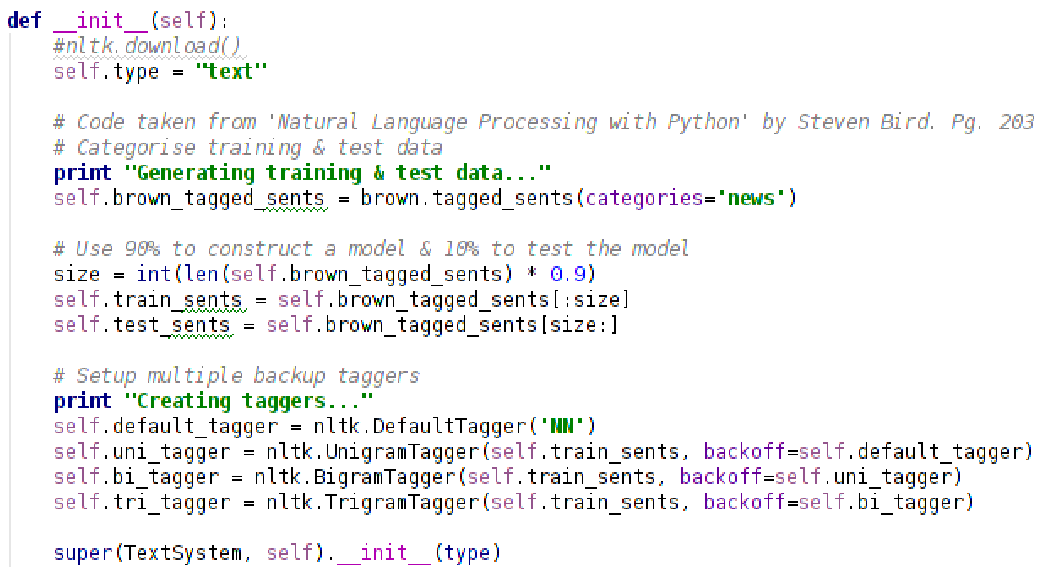


```
# Rule base for chest entity

open
    use open()
    when
        chest_facts.closed(True)
        python
            from Scenes import demoScene

            if demoScene.DemoScene.chest_open is True and demoScene.DemoScene.has_key is False:
                demoScene.DemoScene.demo_player.textComp.output_string = "The chest is already open. It contains a key"

            elif demoScene.DemoScene.chest_open is True and demoScene.DemoScene.has_key is True:
                demoScene.DemoScene.demo_player.textComp.output_string = "The chest is already open. It is empty"

            else:
                demoScene.DemoScene.demo_player.textComp.output_string = "You open the chest. Inside, you find a key"

                demoScene.DemoScene.add_entities(demoScene.DemoScene.demo_key)
                demoScene.DemoScene.key_present = True
                demoScene.DemoScene.chest_open = True

kick
    use kick()
    when
        python
            from Scenes import demoScene
            if demoScene.DemoScene.chest_open is True and demoScene.DemoScene.has_key is False:
                demoScene.DemoScene.demo_player.textComp.output_string = "You kick the chest. The key rattles around inside"


            elif demoScene.DemoScene.chest_open is True and demoScene.DemoScene.has_key is True:
                    demoScene.DemoScene.demo_player.textComp.output_string = "You kick the empty chest"

            else:
                    demoScene.DemoScene.demo_player.textComp.output_string = "You kick the chest. Something rattles around inside it"
```

Figure 4.5: Sample rule base chest.krb

Text System

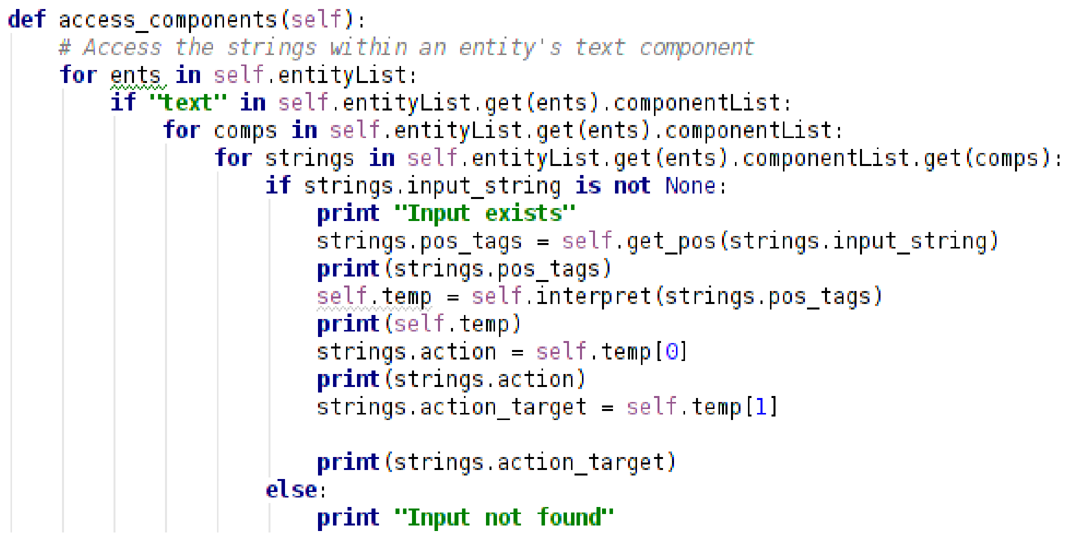
The Text System is the most essential system in the engine. It has four important methods; the initialise method, access_components(), get_pos(), get_syns() and interpret() method. The initialise method generates the taggers (described earlier in **2. Research**) and trains them. The taggers are trained using the Brown corpus' “news” category. The initialise method splits the corpus into training and testing data; ninety percent is used for training, while the other ten percent is used for testing. The three taggers are trained, their back-off taggers are set and the default is set.



```python
def __init__(self):
    #nltk.download()
    self.type = "text"

    # Code taken from 'Natural Language Processing with Python' by Steven Bird. Pg. 203
    # Categorise training & test data
    print "Generating training & test data..."
    self.brown_tagged_sents = brown.tagged_sents(categories='news')

    # Use 90% to construct a model & 10% to test the model
    size = int(len(self.brown_tagged_sents) * 0.9)
    self.train_sents = self.brown_tagged_sents[:size]
    self.test_sents = self.brown_tagged_sents[size:]

    # Setup multiple backup taggers
    print "Creating taggers..."
    self.default_tagger = nltk.DefaultTagger('NN')
    self.uni_tagger = nltk.UnigramTagger(self.train_sents, backoff=self.default_tagger)
    self.bi_tagger = nltk.BigramTagger(self.train_sents, backoff=self.uni_tagger)
    self.tri_tagger = nltk.TrigramTagger(self.train_sents, backoff=self.bi_tagger)

    super(TextSystem, self).__init__(type)
```

Figure 4.6: __init__() method of the Text System

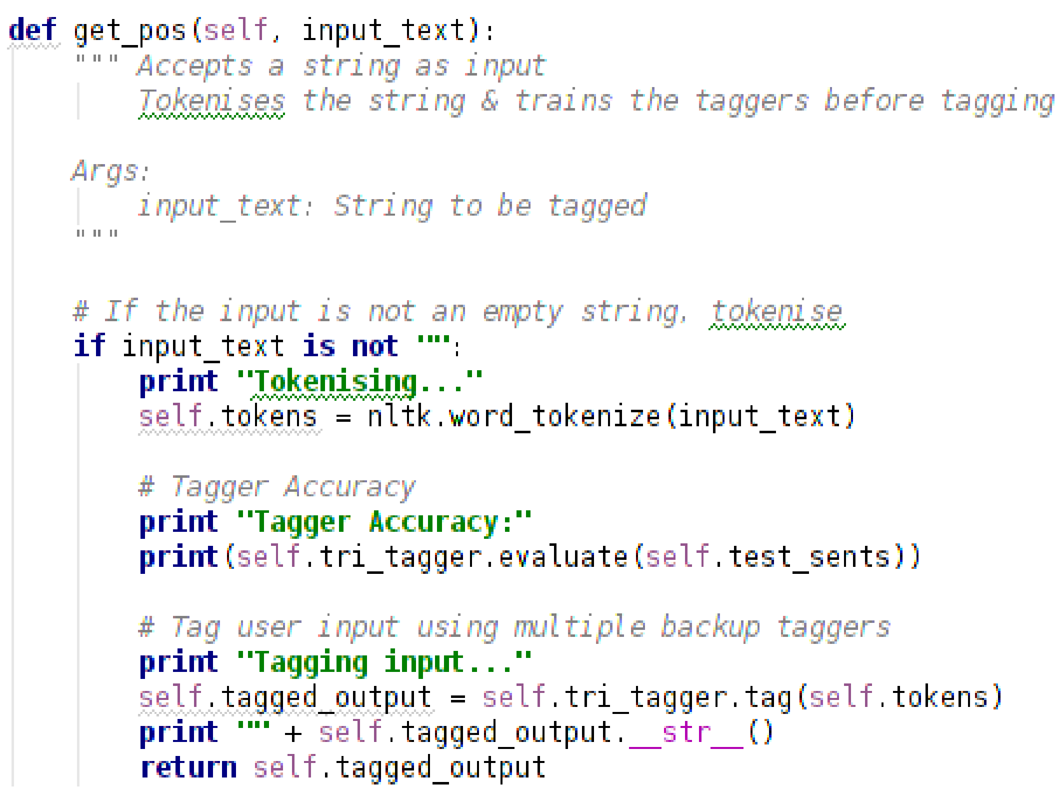
The access_components() is similar to Knowledge System's. The main difference is that it interprets the string for each component rather than evaluating a rule base. Once again, the update() method does not doing anything by default as it is inefficient to perform NLP on each frame; it is only necessary when new input is received.



```python
def access_components(self):
    # Access the strings within an entity's text component
    for ents in self.entityList:
        if "text" in self.entityList.get(ents).componentList:
            for comps in self.entityList.get(ents).componentList:
                for strings in self.entityList.get(ents).componentList.get(comps):
                    if strings.input_string is not None:
                        print "Input exists"
                        strings.pos_tags = self.get_pos(strings.input_string)
                        print(strings.pos_tags)
                        self.temp = self.interpret(strings.pos_tags)
                        print(self.temp)
                        strings.action = self.temp[0]
                        print(strings.action)
                        strings.action_target = self.temp[1]

                        print(strings.action_target)
                    else:
                        print "Input not found"
```

Figure 4.7: access_components() method of the Text System

The get_pos() method performs part-of-speech tagging on the input. This method accepts a string to operate on. If the input is not null, it will tokenise it and then tag each of those tokens.



```
def get_pos(self, input_text):
    """ Accepts a string as input
        Tokenises the string & trains the taggers before tagging

    Args:
        input_text: String to be tagged
    """

    # If the input is not an empty string, tokenise
    if input_text is not "":
        print "Tokenising..."
        self.tokens = nltk.word_tokenize(input_text)

        # Tagger Accuracy
        print "Tagger Accuracy:"
        print(self.tri_tagger.evaluate(self.test_sents))

        # Tag user input using multiple backup taggers
        print "Tagging input..."
        self.tagged_output = self.tri_tagger.tag(self.tokens)
        print "" + self.tagged_output.__str__()
        return self.tagged_output
```

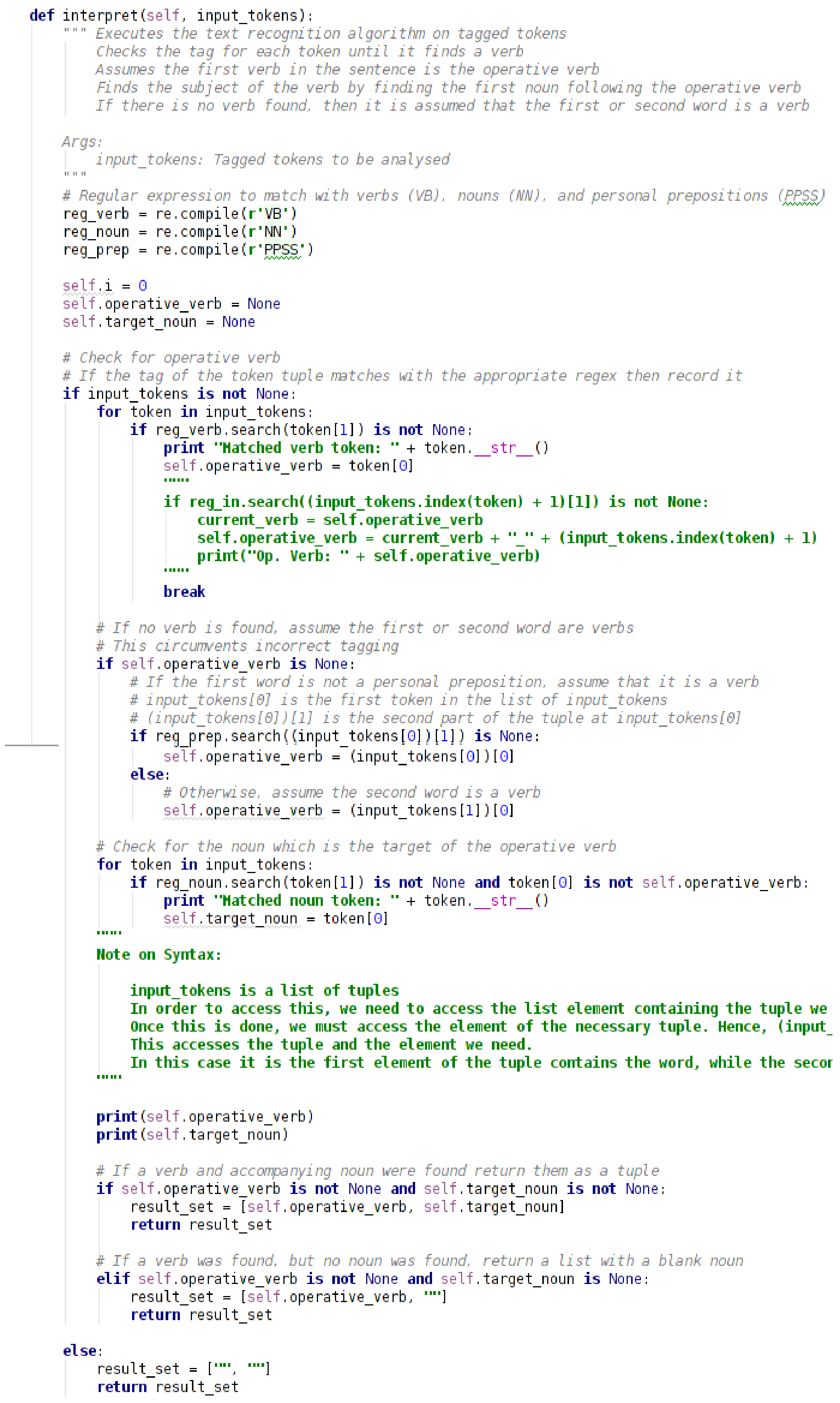
Figure 4.8: get_pos() method of the Text System

The interpret() method performs the NLP for the Text System by using regular expressions. Three regular expressions are used to catch each of the tags that the system is looking for. reg_verb attempts to extract the 'VB' tag which represents a verb, the reg_noun attempts to extract the 'NN' tag which represents a noun and the reg_prep attempts to extract the 'PPSS' tag which represents a personal preposition.

The interpret() method begins iterating through the tokens. It's goal is to search for the first verb token, and then found a following verb. Each of the tokens is a tuple, with the second part holding the token's tag. During iteration, only the second part of the tuple is checked by reg_verb. If it is a match, the operative_verb (I.E: the verb used to represent the rule we will search for in the Knowledge System) is set to that token.

Initially, if there was no verb found then the game would crash. In order to fix this an assumption is made about the input. The assumption is that if the first word in the sentence is a personal preposition, then the following word must be a verb. Otherwise, it's assumed that the first word in the sentence is a verb. To do this the first token is checked to see if it is a personal preposition, using reg_prep. If it is then the first token is set to the operative_verb. If not, then the second token is set to the operative verb.

To find the noun the tokens are iterated through once again. If the second part of the tuple is not null and the part of the tuple (I.E: the word itself) is not the same as the operative_verb, then reg_noun is used to check if the token is a noun. If it is, it's set to the target_noun (I.E: the noun which is considered the subject of the action). If both a verb and noun were found then a list containing both is returned. Otherwise, if a verb was found and a noun was not found then a list containing the verb and a blank element are returned. This is done as returning a list which contained a null element resulted in the game crashing when performing operations on the list. However, if there were no tokens to work on, a list with two blank elements are returned.

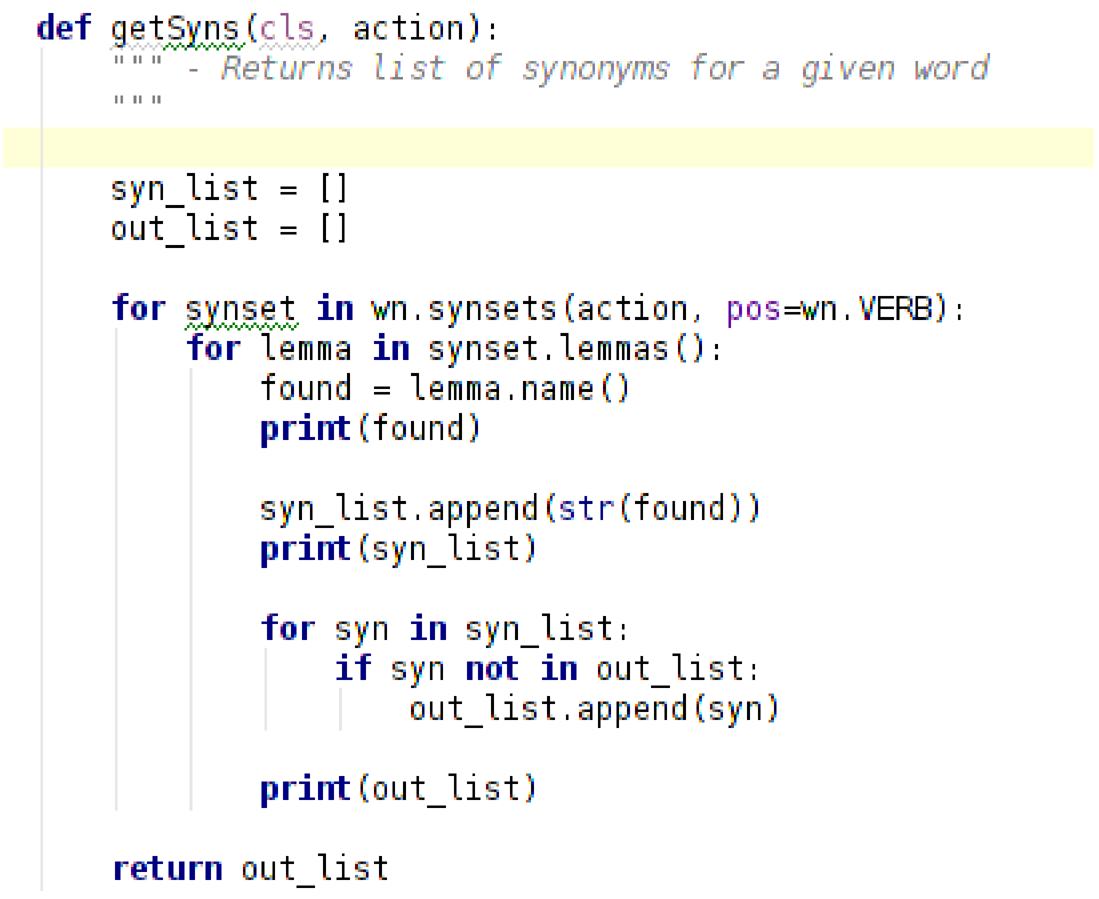


```python
def interpret(self, input_tokens):
    """ Executes the text recognition algorithm on tagged tokens
        Checks the tag for each token until it finds a verb
        Assumes the first verb in the sentence is the operative verb
        Finds the subject of the verb by finding the first noun following the operative verb
        If there is no verb found, then it is assumed that the first or second word is a verb

    Args:
        input_tokens: Tagged tokens to be analysed
    """
    # Regular expression to match with verbs (VB), nouns (NN), and personal prepositions (PPSS)
    reg_verb = re.compile(r'VB')
    reg_noun = re.compile(r'NN')
    reg_prep = re.compile(r'PPSS')

    self.i = 0
    self.operative_verb = None
    self.target_noun = None

    # Check for operative verb
    # If the tag of the token tuple matches with the appropriate regex then record it
    if input_tokens is not None:
        for token in input_tokens:
            if reg_verb.search(token[1]) is not None:
                print "Matched verb token: " + token.__str__()
                self.operative_verb = token[0]
                """
                if reg_in.search((input_tokens.index(token) + 1)[1]) is not None:
                    current_verb = self.operative_verb
                    self.operative_verb = current_verb + "_" + (input_tokens.index(token) + 1)
                    print("Op. Verb: " + self.operative_verb)
                """
                break

        # If no verb is found, assume the first or second word are verbs
        # This circumvents incorrect tagging
        if self.operative_verb is None:
            # If the first word is not a personal preposition, assume that it is a verb
            # input_tokens[0] is the first token in the list of input_tokens
            # (input_tokens[0])[1] is the second part of the tuple at input_tokens[0]
            if reg_prep.search((input_tokens[0])[1]) is None:
                self.operative_verb = (input_tokens[0])[0]
            else:
                # Otherwise, assume the second word is a verb
                self.operative_verb = (input_tokens[1])[0]

        # Check for the noun which is the target of the operative verb
        for token in input_tokens:
            if reg_noun.search(token[1]) is not None and token[0] is not self.operative_verb:
                print "Matched noun token: " + token.__str__()
                self.target_noun = token[0]
        """
        Note on Syntax:

            input_tokens is a list of tuples
            In order to access this, we need to access the list element containing the tuple we
            Once this is done, we must access the element of the necessary tuple. Hence, (input_
            This accesses the tuple and the element we need.
            In this case it is the first element of the tuple contains the word, while the secor
        """

        print(self.operative_verb)
        print(self.target_noun)

        # If a verb and accompanying noun were found return them as a tuple
        if self.operative_verb is not None and self.target_noun is not None:
            result_set = [self.operative_verb, self.target_noun]
            return result_set

        # If a verb was found, but no noun was found, return a list with a blank noun
        elif self.operative_verb is not None and self.target_noun is None:
            result_set = [self.operative_verb, ""]
            return result_set

    else:
        result_set = ["", ""]
        return result_set
```

Figure 4.9: interpret() method of the Text System

The get_syns() method is used to get the synonyms of a verb. It is intended to be used if the Knowledge System fails to find a match using the operative_verb. If the operative_verb fails it's synonyms will be checked instead. In order to do this the get_syns() methods accepts an “action” parameter. WordNet is used to get all the synonyms of the “action” parameter that are verbs. These are saved into a list. The duplicates are removed from this list, resulting in an output list of all verb synonyms being returned.

One issue to be noted is that the personal preposition “I” is not recognised correctly. This is a joint problem between the InputBox API used to receive the text input and NLTK. NLTK fails to tag “I” unless it is capitalised. The problem is that the InputBox API doesn't allow for upper case characters. This problem is a result of an API limitation and a suitable fix could not be found in the given time frame.



```
def getSyns(cls, action):
    """ - Returns list of synonyms for a given word
    """

    syn_list = []
    out_list = []

    for synset in wn.synsets(action, pos=wn.VERB):
        for lemma in synset.lemmas():
            found = lemma.name()
            print(found)

            syn_list.append(str(found))
            print(syn_list)

            for syn in syn_list:
                if syn not in out_list:
                    out_list.append(syn)

            print(out_list)

    return out_list
```

Figure 4.10: get_syns() method from the TextSystem class

Component

The Component class is incredibly simple and only contains one method; get_type(). The Component class is another template class which is designed to be sub-classed by developers in order to create specific implementations of it. The get_type() method is used to fetch a default type if the “type” attribute is not already specified within the component. This is accomplished by retrieving the component's name. When using components it is important to remember that their relevant system should alter their attributes when performing operations.

Knowledge Component

The Knowledge Component contains attributes required for interacting with the Knowledge System. In contains a dict to store rule bases and fact bases, should that be needed. In addition, it contains a “kb” field to denote the current knowledge base that is active for that component, a current_rule field, and an eval_result field which is used to store whether an evaluation was successful or not.

The recommended way for utilising this in conjunction with the Knowledge System is to set the values of kb and current_rule within the scene, and then use those attributes when calling methods from the Knowledge System.

Text Component

The Text Component contains attributes required for interacting with the Text System. It stores the input and output strings, the current action being taken (I.E: the verb found to be the operative_verb in the Text System), the action target (I.E: the noun found to be the target_noun in the Text System), and the POS tags of the current input.

Once again, it is recommended to use these attributes in conjunction with the Text System. Any input and output for an entity should be set using the input_string and output_string fields. The same is applicable for action and action_target.

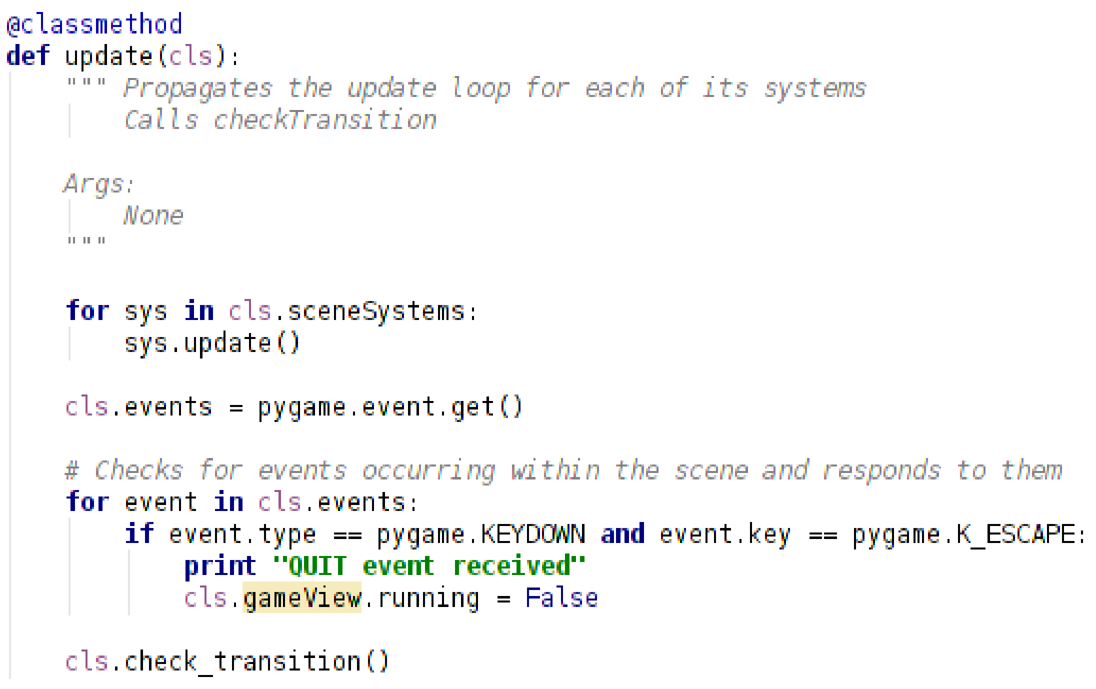
Scene

The Scene class is used as a template for future scenes, to be designed by a developer. The developer can use the Scene template class to create more specific templates. The Scene is designed to hold references to all entities and systems which are present in the scene, in addition to checking conditions for transitioning to a new scene, and drawing relevant backgrounds and objects to the screen. Each scene also contains a gameView attribute. This is a reference to the current running instance of the game engine. This is used to render images and text to the screen.

The Scene class contains the following methods; add_systems(), add_initial_entities, add_entities(), update(), check_transition() and draw(). The draw() method is used to draw backgrounds and entities to the screen. This is empty as it must be defined by the developer. The check_transition() method is also an empty method which must be defined by the developer, but it is used to check if particular conditions are met which cause the game to switch from the current scene to the specified new scene.

The add_systems() method is extremely straightforward; it accepts a list of systems and adds them to the scene's sceneSystems list. Adding entities was not as simple. Adding entities to the scene when the scene is initialised was straightforward, however, attempting to add entities while the scene was active resulted in the game crashing or entities not being added. In order to fix this, adding entities is done through two different methods. The add_initial_entities() method is designed to be used when initialising the scene in order to populate it with entities. The add_entities() method is used to add further entities to the scene while it is active and running.

The update() method is designed to be called by all subclasses of the Scene. This superclass update() method will call the update() method for all systems belonging to the class. It will also call the class' check_transition() method and check if the game is being quit.



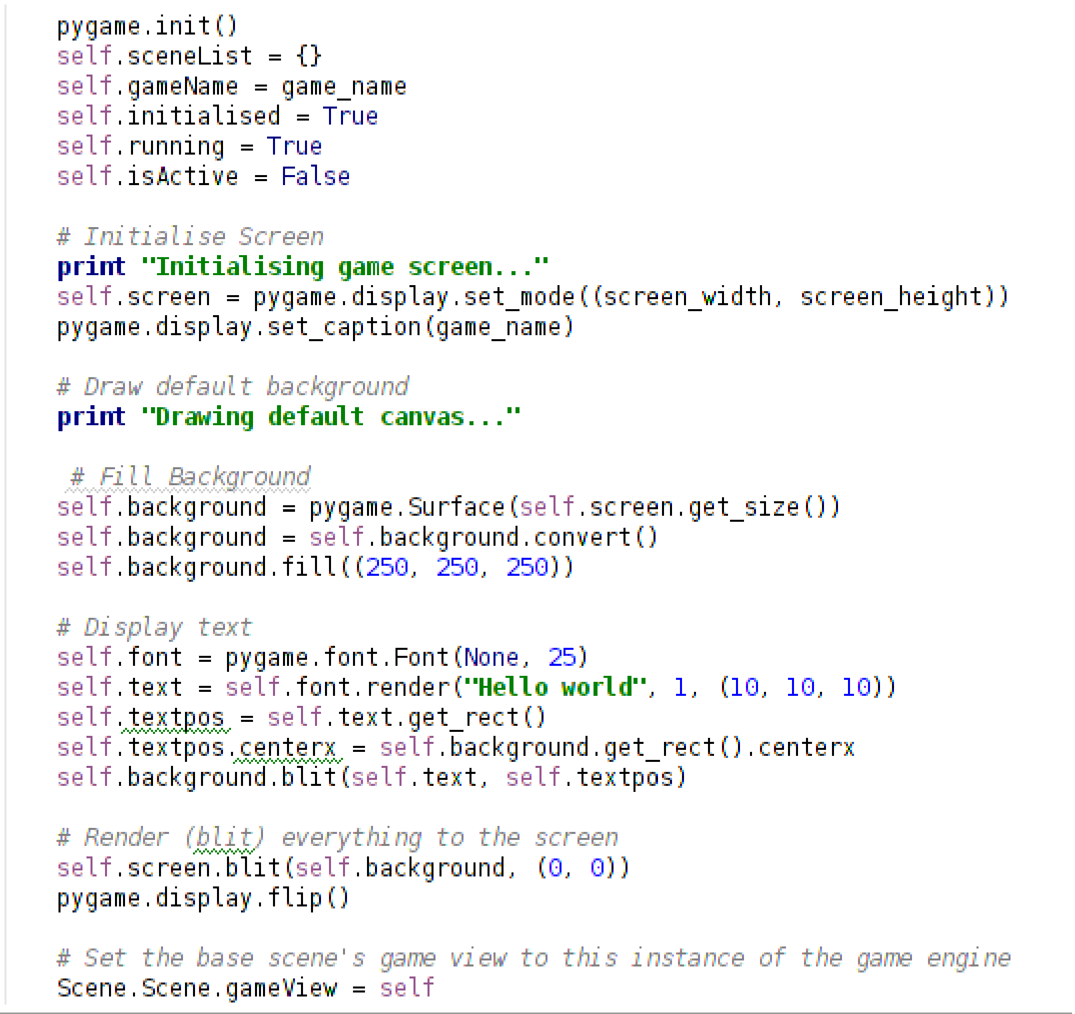
```
@classmethod
def update(cls):
    """ Propagates the update loop for each of its systems
        Calls checkTransition

    Args:
        None
    """

    for sys in cls.sceneSystems:
        sys.update()

    cls.events = pygame.event.get()

    # Checks for events occurring within the scene and responds to them
    for event in cls.events:
        if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
            print "QUIT event received"
            cls.gameView.running = False

    cls.check_transition()
```

Figure 4.11: update() method from the Scene class

Game Engine

The Game class is the most important part of the project, and is used as a container to run the game. The Game class initialises, runs and stops the game. It does this by initialising the PyGame module and updating the active scene until it is instructed to stop. This class is designed to be subclasses and contains the following methods; __init__(), run(), update(), set_active_scene().

The __init__() class is used to create a new instance of the Game class and accepts a name for the game and dimensions for the screen. When called, this method initialises the PyGame module, sets the game to be initialised and running, and draws the default background. Afterwards, it sets the game view of the Scene template class to this current instance of the game engine.



```
pygame.init()
self.sceneList = {}
self.gameName = game_name
self.initialised = True
self.running = True
self.isActive = False

# Initialise Screen
print "Initialising game screen..."
self.screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption(game_name)

# Draw default background
print "Drawing default canvas..."

 # Fill Background
self.background = pygame.Surface(self.screen.get_size())
self.background = self.background.convert()
self.background.fill((250, 250, 250))

# Display text
self.font = pygame.font.Font(None, 25)
self.text = self.font.render("Hello world", 1, (10, 10, 10))
self.textpos = self.text.get_rect()
self.textpos.centerx = self.background.get_rect().centerx
self.background.blit(self.text, self.textpos)

# Render (blit) everything to the screen
self.screen.blit(self.background, (0, 0))
pygame.display.flip()

# Set the base scene's game view to this instance of the game engine
Scene.Scene.gameView = self
```

Figure 4.12: __init__() method of the Game class

After creating a new instance of the game engine, an initial active scene must be set. This is done using the set_active_scene() method. This accepts a scene as a parameter. Once called, it sets the received scene to the active scene then calls it's initialise() and update() methods.

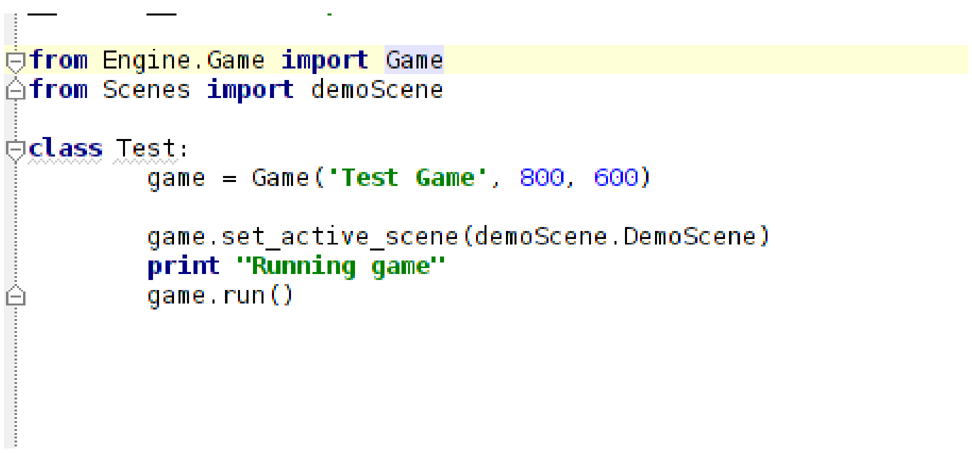
In order to start the game engine after initialisation, the run() method must be called. This continues to called the update() method until a QUIT message is received from the PyGame module.

The update() method is called until the game is quit. As long as there is an active scene, the update() method will called the update() of the active scene and update the screen.

Development & Problem Resolution of the Demo Game

Test Engine

The Test Engine is a simple class. It contains a new instance of the Game class, sets the active scene and runs the game. In order to set the active scene, the class must first import it. This allows the Test Engine to be aware that the scene it's trying to activate exists.



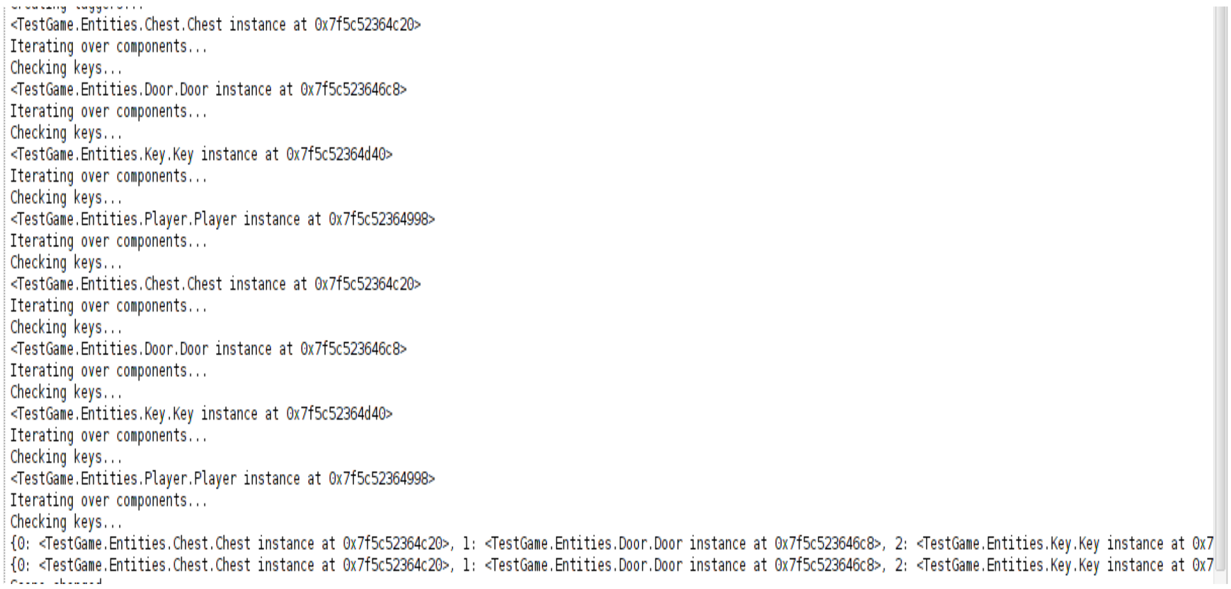
```
from Engine.Game import Game
from Scenes import demoScene

class Test:
        game = Game('Test Game', 800, 600)

        game.set_active_scene(demoScene.DemoScene)
        print "Running game"
        game.run()
```

Figure 4.13: Screenshot of the Test Engine class

Demo Scene

The Demo Scene is a working example of how to use the utilities provided by Tall Tales. The Demo Scene provides the user with a simple scenario. The user is in a room that contains a locked door and a chest. They must figure out how to escape. To create this scenario, it uses Tall Tales' Text System and Knowledge System.

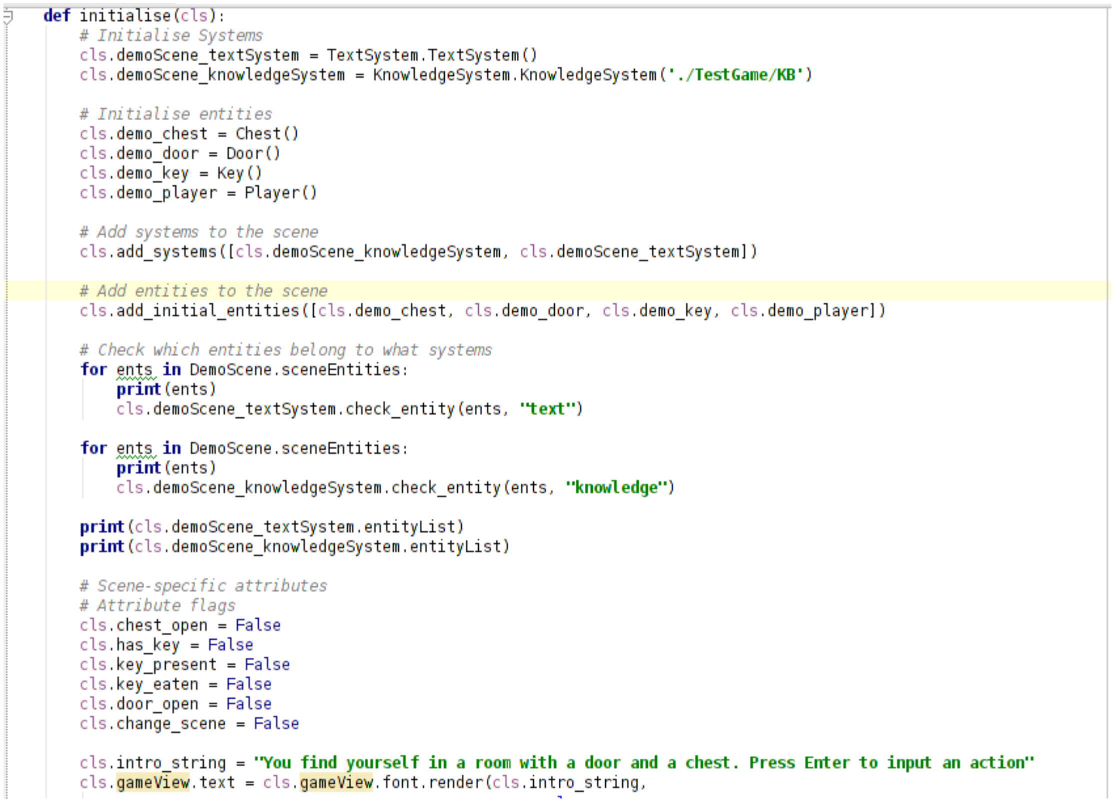
In the initialise() method the scene is created. An instance of the Text System and Knowledge System are created and added to the scene. Four entities are then created; a door, a chest, a key, and a player. These are also added to the scene. A debug check is performed to ensure that the scenes and entities belong to the scene (Figure 4.7)



```
Creating taggers...
<TestGame.Entities.Chest.Chest instance at 0x7f5c52364c20>
Iterating over components...
Checking keys...
<TestGame.Entities.Door.Door instance at 0x7f5c523646c8>
Iterating over components...
Checking keys...
<TestGame.Entities.Key.Key instance at 0x7f5c52364d40>
Iterating over components...
Checking keys...
<TestGame.Entities.Player.Player instance at 0x7f5c52364998>
Iterating over components...
Checking keys...
<TestGame.Entities.Chest.Chest instance at 0x7f5c52364c20>
Iterating over components...
Checking keys...
<TestGame.Entities.Door.Door instance at 0x7f5c523646c8>
Iterating over components...
Checking keys...
<TestGame.Entities.Key.Key instance at 0x7f5c52364d40>
Iterating over components...
Checking keys...
<TestGame.Entities.Player.Player instance at 0x7f5c52364998>
Iterating over components...
Checking keys...
{0: <TestGame.Entities.Chest.Chest instance at 0x7f5c52364c20>, 1: <TestGame.Entities.Door.Door instance at 0x7f5c523646c8>, 2: <TestGame.Entities.Key.Key instance at 0x7
{0: <TestGame.Entities.Chest.Chest instance at 0x7f5c52364c20>, 1: <TestGame.Entities.Door.Door instance at 0x7f5c523646c8>, 2: <TestGame.Entities.Key.Key instance at 0x7
Scene changed
```

Figure 4.14: Debug data showing the presence of entities in the demo scene

Lastly, a set of scene-specific flags are defined. These present to compensate for the poor functionality of the Pyke API, as mentioned earlier. Then, an introductory message is displayed. This message informs the user of the basic instructions to play the game.



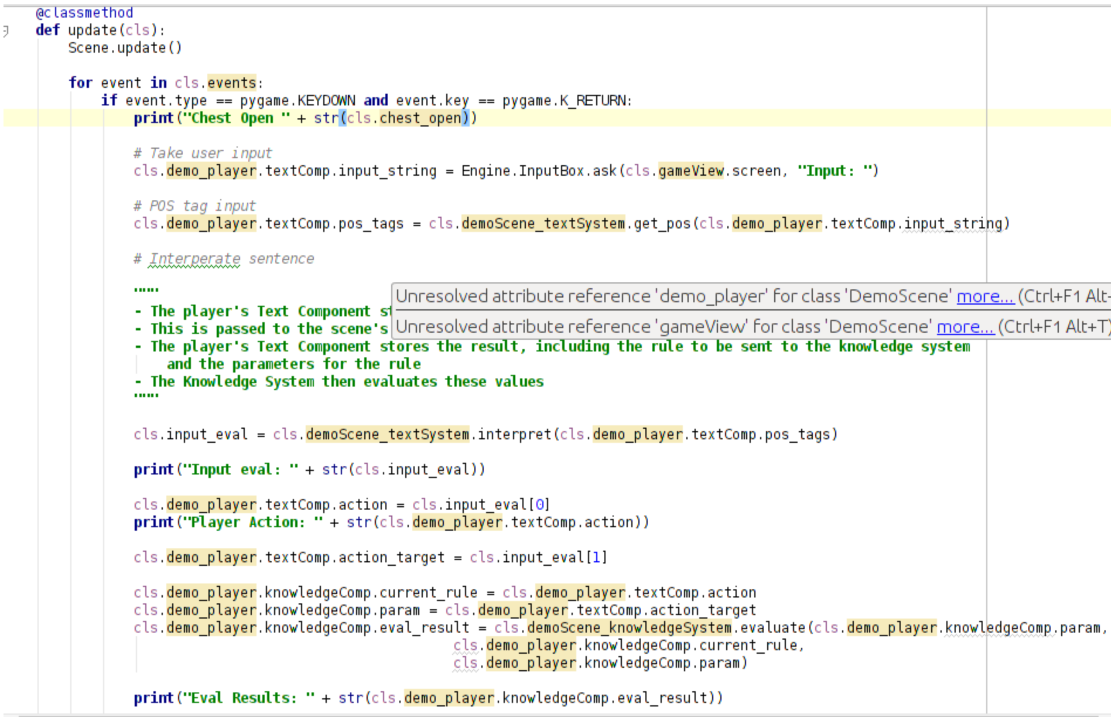
```
def initialise(cls):
    # Initialise Systems
    cls.demoScene_textSystem = TextSystem.TextSystem()
    cls.demoScene_knowledgeSystem = KnowledgeSystem.KnowledgeSystem('./TestGame/KB')

    # Initialise entities
    cls.demo_chest = Chest()
    cls.demo_door = Door()
    cls.demo_key = Key()
    cls.demo_player = Player()

    # Add systems to the scene
    cls.add_systems([cls.demoScene_knowledgeSystem, cls.demoScene_textSystem])

    # Add entities to the scene
    cls.add_initial_entities([cls.demo_chest, cls.demo_door, cls.demo_key, cls.demo_player])

    # Check which entities belong to what systems
    for ents in DemoScene.sceneEntities:
        print(ents)
        cls.demoScene_textSystem.check_entity(ents, "text")

    for ents in DemoScene.sceneEntities:
        print(ents)
        cls.demoScene_knowledgeSystem.check_entity(ents, "knowledge")

    print(cls.demoScene_textSystem.entityList)
    print(cls.demoScene_knowledgeSystem.entityList)

    # Scene-specific attributes
    # Attribute flags
    cls.chest_open = False
    cls.has_key = False
    cls.key_present = False
    cls.key_eaten = False
    cls.door_open = False
    cls.change_scene = False

    cls.intro_string = "You find yourself in a room with a door and a chest. Press Enter to input an action"
    cls.gameView.text = cls.gameView.font.render(cls.intro_string,
```

Figure 4.15: initialise() method from the demo scene

After the initialise() method is the update() method. This is where the game engine's systems are used to create gameplay. At the beginning of the update() method the Scene superclass' update() is called. This performs general “housekeeping” such as updating all the scene's systems (if applicable), and checking if the game has been quit.

Next, the update method checks the PyGame event queue. The only event checked is whether the return key has been pressed. This is used to invoke the InputBox. The value received by the InputBox is then set as the player's input_string in it's Text Component. The player's input_string is the POS tagged, which is stored in the player's Text Component. The player's POS tagged input is then interpreted by the Text System. This operation returns a list which is stored in the scene's input_eval attribute. The input_eval attribute is used to determine the player's action, and the target of that action. The first element (input_eval[0]) is the action. The second element (input_eval[1]) is the action_target. This is defined by the Text System.

After NLP has been performed in the update() method, the Knowledge System evaluates the input using the rules for that scene. Attributes contained within the player's Knowledge Component are set. The current_rule is set to the player's action, and the param is set to the player's action_target. These are then passed to the Knowledge System's evaluate() method. The result of this is stored in the eval_result field of the player's Knowledge Component. See Figure 4.9 to further illustrate this.



```
@classmethod
def update(cls):
    Scene.update()

    for event in cls.events:
        if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN:
            print("Chest Open " + str(cls.chest_open))

            # Take user input
            cls.demo_player.textComp.input_string = Engine.InputBox.ask(cls.gameView.screen, "Input: ")

            # POS tag input
            cls.demo_player.textComp.pos_tags = cls.demoScene_textSystem.get_pos(cls.demo_player.textComp.input_string)

            # Interperate sentence

            """
            - The player's Text Component st
            - This is passed to the scene's
            - The player's Text Component stores the result, including the rule to be sent to the knowledge system
              and the parameters for the rule
            - The Knowledge System then evaluates these values
            """

            cls.input_eval = cls.demoScene_textSystem.interpret(cls.demo_player.textComp.pos_tags)

            print("Input eval: " + str(cls.input_eval))

            cls.demo_player.textComp.action = cls.input_eval[0]
            print("Player Action: " + str(cls.demo_player.textComp.action))

            cls.demo_player.textComp.action_target = cls.input_eval[1]

            cls.demo_player.knowledgeComp.current_rule = cls.demo_player.textComp.action
            cls.demo_player.knowledgeComp.param = cls.demo_player.textComp.action_target
            cls.demo_player.knowledgeComp.eval_result = cls.demoScene_knowledgeSystem.evaluate(cls.demo_player.knowledgeComp.param,
                                                        cls.demo_player.knowledgeComp.current_rule,
                                                        cls.demo_player.knowledgeComp.param)

            print("Eval Results: " + str(cls.demo_player.knowledgeComp.eval_result))
```

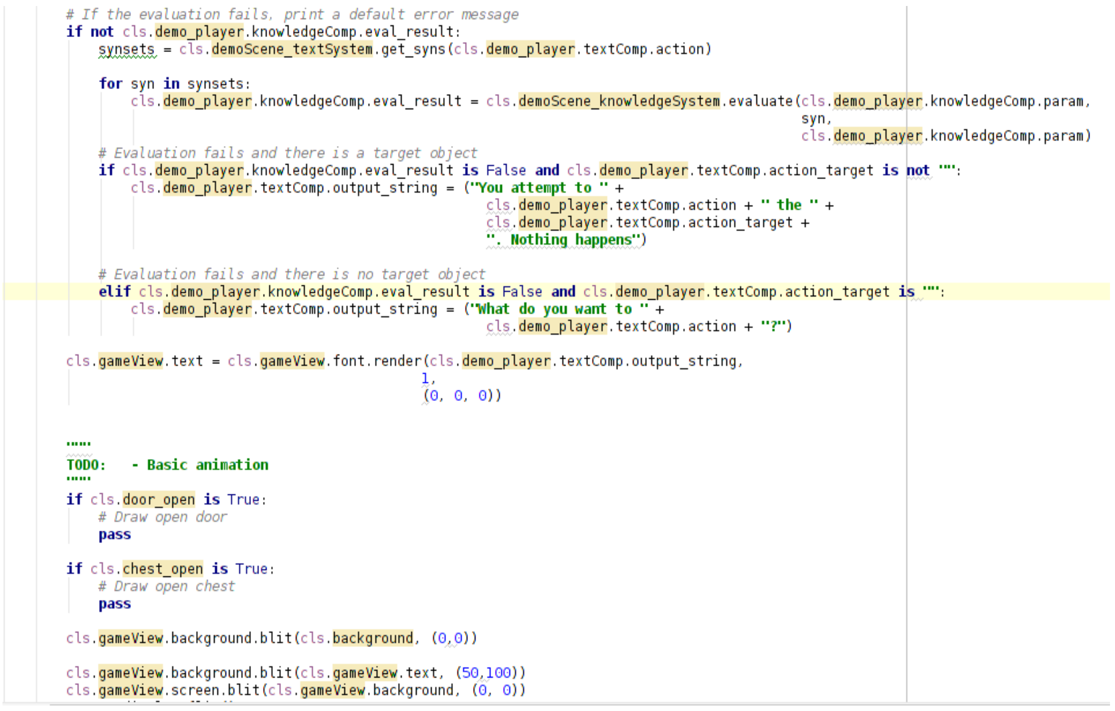
```
Unresolved attribute reference 'demo_player' for class 'DemoScene' more... (Ctrl+F1 Alt-
Unresolved attribute reference 'gameView' for class 'DemoScene' more... (Ctrl+F1 Alt+T)
```

Figure 4.16: Demo scene update() method performing NLP and knowledge evaluation

If the rule is successfully evaluated, that rule will set its own unique output to be drawn to the screen. Otherwise, the update() method catches it and attempts to re-evaluate the rule using synsets. If that fails then a default output message is used.

If the evaluation of the player's action fails, the Text System's get_syns() method is used. In this scenario, get_syns() is provided with the player's action. From here, it constructs a list of synonyms for the player's action, known as a synset. Each element of the synset is then evaluated using the Knowledge System. If these fail as well, then the update() method uses one of it's default output messages.

There are two different outputs for a failed evaluation. One is if the rule fails and the player has an action_target. This will produce the message “You attempt to X the Y. Nothing happens”. The second is if the rule fails the player does not have an action_target. This produces the message “What do you want to X?”. No matter which output message is used, they all change the player's output_string in its Text Component. In order to draw this output to the screen, the text field of the game view must be set to the desired output. Figure 4.10 highlights this process.

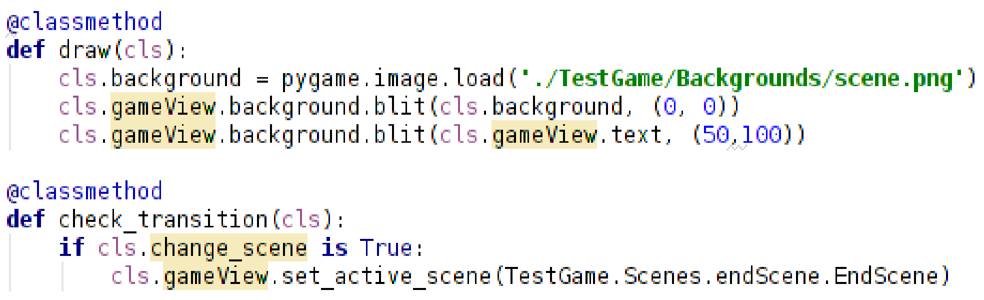


```python
# If the evaluation fails, print a default error message
if not cls.demo_player.knowledgeComp.eval_result:
    synsets = cls.demoScene_textSystem.get_syns(cls.demo_player.textComp.action)

    for syn in synsets:
        cls.demo_player.knowledgeComp.eval_result = cls.demoScene_knowledgeSystem.evaluate(cls.demo_player.knowledgeComp.param,
                                                                                           syn,
                                                                                           cls.demo_player.knowledgeComp.param)
    # Evaluation fails and there is a target object
    if cls.demo_player.knowledgeComp.eval_result is False and cls.demo_player.textComp.action_target is not "":
        cls.demo_player.textComp.output_string = ("You attempt to " +
                                                  cls.demo_player.textComp.action + " the " +
                                                  cls.demo_player.textComp.action_target +
                                                  ". Nothing happens")

    # Evaluation fails and there is no target object
    elif cls.demo_player.knowledgeComp.eval_result is False and cls.demo_player.textComp.action_target is "":
        cls.demo_player.textComp.output_string = ("What do you want to " +
                                                  cls.demo_player.textComp.action + "?")

cls.gameView.text = cls.gameView.font.render(cls.demo_player.textComp.output_string,
                                             1,
                                             (0, 0, 0))


"""
TODO:   - Basic animation
"""
if cls.door_open is True:
    # Draw open door
    pass

if cls.chest_open is True:
    # Draw open chest
    pass

cls.gameView.background.blit(cls.background, (0,0))

cls.gameView.background.blit(cls.gameView.text, (50,100))
cls.gameView.screen.blit(cls.gameView.background, (0, 0))
```

Figure 4.17: Handling failed rule evaluations in the demo scene

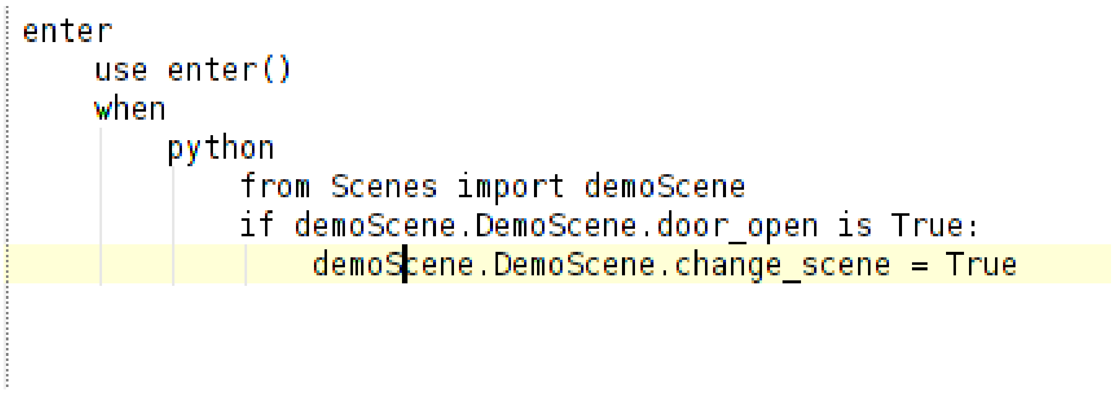
The last operation to occur in the update() method is to call the check_transition() method. This checks the game engine needs to change scene, and what scene to change to.

The last two methods in the Demo Scene are the draw() method and the check_transition() method. The draw() method is only called on initialisation of the scene. It renders a background image and displays any introductory text. The check_transition method in the Demo Scene is incredibly simple. It checks if the change_scene flag is set to True. If it is, the scene is changed to the End Scene, which displays a different background. The change_scene flag is only set to True of the "Enter" rule for the door entity is successfully evaluated.



```python
@classmethod
def draw(cls):
    cls.background = pygame.image.load('./TestGame/Backgrounds/scene.png')
    cls.gameView.background.blit(cls.background, (0, 0))
    cls.gameView.background.blit(cls.gameView.text, (50,100))

@classmethod
def check_transition(cls):
    if cls.change_scene is True:
        cls.gameView.set_active_scene(TestGame.Scenes.endScene.EndScene)
```

Figure 4.18: draw() and check_transition() methods from the demo scene-specific



```
enter
    use enter()
    when
        python
            from Scenes import demoScene
            if demoScene.DemoScene.door_open is True:
                demoScene.DemoScene.change_scene = True
```

Figure 4.19: Rule to trigger a scene transition

Entities

There are four entities in the demo game; the door, the chest, the player, and the key. Each of these entities are identical in content. All four contain a Text Component, and a Knowledge Component. During development, only the player's Text Component and Knowledge Components are actually used. This is due to the fact that the player is the entity in the game which is entering text and attempting evaluate rules in the scene.

Key Development Components

The following components of the project where key to development:

- Text System
- Knowledge System
- Game Engine

Text System

The Text System is the most important part of the project. This handles all the NLP, which is the core aspect of the project. Without this component there would be no project. This system can have its functionality expanded and make the NLP more comprehensive and robust.

Knowledge System

While the Knowledge System is flawed due to the use of an ill-suited API, it is still important. Irrespective of the problems, the main algorithm used to identify whether a user's action is valid can be applied to a better designed system, and can even be adapted and expanded to fit more complicated iterations of the system.

Game Engine

The Game Engine is crucial to the project as it holds everything together and ensures that all the individual pieces run together. Having a functional Game Engine component provides a solid base for the project to by changed and expanded on in almost any way necessary.

List of Classes & Related Usage

The following are the classes that were used as part of the engine:

- System.py
- Scene.py
- InputBox.py
- Game.py
- Component.py
- KnowledgeSystem.py
- TextSystem.py
- KnowledgeComponent.py
- TextComponent.py
- Entity.py

Here is the resulting API for using these classes to develop a game. The following should be used with the Figure 4.1 as a reference.

System

- To be sub-classed to create game systems
- __init__(self, system_type)
    - Accepts a system type from it's child classes
    - Initialises an empty entityList
- check_entity(self, entity, ent_type)
    - Checks if an entity contains ent_type
    - If it does, add it to the entityList
- update(self)
    - Empty method

    - Defined by developer on a per-system basis

<u>Scene</u>

- To use, subclass it to create game scenes
- initialise(cls)
    - To be defined by the developers
- add_systems(cls, systems=[])
    - Accepts a list of systems
    - Adds them to the scene
- add_initial_entities(cls, entities=[])
    - Accepts a list of entities
    - Adds them to the scene
    - To be used during initialisation of the scene
- add_entities(cls, *args)
    - Accepts multiple entities
    - Extends the list of entities in the scene
    - Used during run-time
- update(cls)
    - Updates all systems in the scene
    - Checks if the game has been exited
    - Checks for transition
    - Should be called at the start of each scene's update()
- check_transition(cls)
    - Checks the conditions for the scene to be changed
    - Defined by the developer in subclasses
- draw(self)
    - Renders initial objects to the screen
    - Defined by the developer in subclasses

InputBox

- get_key()
  - Polls event queue for a key press
  - Records key press
  - Called by the ask() method
- display_box(screen, message)
  - Accepts a view of the current game screen and a message to display
  - Displays the input box
  - Called by the ask() method
- ask(screen, question)
  - Accepts a view of the current game screen and a question to ask
  - To be called when written input is desired

Game

- __init__(self, game_name, screen_width, screen_height)
  - Accepts a name for the game, and screen dimensions
  - Initialises a new instance of the game engine
  - Creates a blank game canvas
- run(self)
  - Called to begin running the game engine
  - Updates the game engine until the game has been exited
- set_active_scene(self, scene)
  - Accepts a scene and sets it to be the active scene
  - This calls the scene's initialise and draw methods
- update(self)
  - Updates the active scene as long as the game is running, and an active scene is present
  - Updates the game display

Component

- To be sub-classed to create game components
- __init__(self)
  - To be defined by the developer
- get_type(self)
  - Returns the component type
  - If no type is defined, the class name is used

Knowledge System

- To use, create a new instance and add it to a scene
- __init__(self, loc)
  - Accepts a directory for the knowledge base files
  - Creates new instance of the knowledge engine
  - Sets the System type to “knowledge”
  - Calls the System parent class' __init__ method
- access_components(self)
  - Accesses the knowledge component of each entity it manages
  - Attempts to evaluate these entities' rules
- add_kb(self, entity, kb_name, kb)
  - Adds a reference to a knowledge base to an entity's Knowledge Component
  - Accepts an entity, a key for the knowledge base, and the knowledge base itself
- evaluate(self, kb, rule, subject)
  - Evaluates a given rule in a particular rule base
  - Accepts a knowledge base to check, a rule, and the target of that rule
  - If it evaluates the rule it returns true
  - Otherwise, it returns false

Text System

- To use, create a new instance and add it to a scene
- __init__(self)

    - Downloads the appropriate NLTK data
    - Generates POS taggers
    - Calls the System parent class' __init__()
- access_components(self)
    - Accesses the text component of each entity it manages
    - Attempts to interpret these entities' text input
- get_pos(self, input_text)
    - Accepts a string
    - Attempts to POS tag the string by tokenising it, then tagging it with the taggers created in __init__()
    - This returns a list of tagged tokens
- interpret(self, input_tokens)
    - Accepts a list of tokens
    - Attempts to interpret the action being taken, and the target of the action
    - Performs this based on the tags of the input tokens
    - This returns a list of two strings
        - One string is the verb which represents an action (operative_verb)
        - The other is the target of that action (target_noun)
- get_syns(cls, action)
    - Accepts a string
    - Generates a list on synonyms for the given word
    - Returns this list of synonyms

<u>Knowledge Component</u>

- To use, create a new instance and add it to an entity
- type
    - Set to “knowledge”
    - Identifier for the component
- kb_dict
    - Dictionary of knowledge bases for the component

- kb
    - Currently active knowledge base
- current_rule
    - Rule which is currently being evaluated
- eval_result
    - Result from the Knowledge System's evaluation

<u>Text Component</u>

- To use, create a new instance and add it to an entity
- type
    - Set to “text”
    - Identifier for the component
- input_string
    - Holds the string received from the event queue in a scene
- output_string
    - Holds the string to be written to the screen
- action
    - Used to hold a verb
    - Holds the operative_verb from the Text System's interpret
- action_target
    - Used to hold the target of the action attribute
    - Holds the target_noun from the Text System's interpret
- pos_tags
    - Holds a list of tagged tokens from the get_pos() method in the Text System

<u>Entity</u>

- To use, subclass it and add it to a scene. Then, add components to it
- Defines a componentDict to hold its components
    - Stored based on component type

    - Key represents component type
- __init__(self)
    - Defined by the developer for particular entities
- add_components(self, key, components=[])
    - Accepts a key, and a list of components
    - Attempts to add the components at a given key in the componentDict
- remove_components(self, key)
    - Accepts a key
    - Removes all components at a specified key
- does_contain(self, type)
    - Accepts a component type
    - Checks the componentDict for components of the given type

The following are classes that were used as part of the test game. Use these as a reference point when developing games using Tall Tales.

- testEngine.py
- demoScene.py
- Player.py
- Key.py
- Door.py
- Chest.py

Identification of APIs

The following external APIs are used in the project:

- PyGame
- Pyke
- NLTK
- InputBox

PyGame

PyGame is used in the Scene and Game classes. PyGame contains a suite of game development utilities that are used for handling input, event management and 2D rendering. In both the Scene and Game classes PyGame is used to receive keyboard events and to render backgrounds, sprites and text to the screen. PyGame API calls are present in the draw() method of Scene subclasses and in the __init__() of the Game class. It is also used in their update() methods to check for keyboard events, such as the game being quit, or a particular key being pressed.

Pyke

Pyke is used by the Knowledge System to facilitate an expert system. Pyke manages logic for evaluating forward and backward chaining rules using a knowledge engine. As a result, the Knowledge System only needs to know what rules and rule bases are being checked. It is not concerned with how the check occurs. When an instance of this system is created a new instance of a knowledge engine is created. When a rule needs to be evaluated the Knowledge System formats the input in a manor which is accepted by Pyke, then calls the relevant Pyke method to evaluate the rule.

NLTK

NLTK is one of the most important and is used extensively in the Text System. In the __init__() method of the Text System nothing but the NLTK API is used. The Brown Corpus is accessed from NLTK and used for training data. The NLTK API is then used to create the taggers. These taggers are trained by the NLTK API. The Text System is unaware of how the these are trained, it just calls the taggers when necessary.

In the get_pos() method NLTK is used to tokenise the input and to call the taggers that are created when the system is initialised. Calling tri_tagger.tag(tokens) uses NLTK to perform the tag() method on the input tokens.

The get_syns() method is the only other part of the Text System which uses the NLTK API. The synset() method is called to retrieve a list on synonyms for the input. The return synset is a list of lemmas. These lemmas are a string which contain the word, the type of word it is, and the context of the word. By calling lemma.name() it is possible to extract just the word from the entire lemma. The lemma's name is what is then later returned from get_syns

InputBox

This is an API found on the PyGame website which facilitates a text input field. This is called within individual game scenes in order to receive text input when desired. In the case of the test game, the enter key invokes the InputBox.

5. System Validation

System Testing

In order to test the project, two methodologies were used; black-box unit testing and integration testing. Unit testing is performed after each individual module of the project has been developed. This module is undergoes a cycle of testing and development until it is deemed to be complete. Afterwards, integration testing is performed using this module.

Black-box unit testing was chosen as it ensures that the individual components of the project work in a scenario where the user does not have in-depth knowledge of how the system functions. Black-box unit testing also helped to prevent me from developing to an implementation (the demo environment), rather than an interface (the game engine). This method of testing was applicable to all parts of the project. Each individual module of the project underwent black-box unit testing.

For each class in the project, black-box testing was performed. Each of the methods were given fixed test values. This was done to provide a degree of consistency. Due to the nature of the ECS architecture, some of this testing was just part of development. When developing the Text System and the Text Component, for example, that allowed me to test the System and Component classes by attempting to utilise their facilities.

To test the Text System, each method was passed the string “Take the hat”. This was used to check if the string could be POS tagged, and interpreted. After this succeeded, the interpret() method was given the string “Take eat the up hat house”. This was done to check if the algorithm worked when passed multiple tokens with the same tag. The result was the interpret() method recording “Take” as the operative verb and “house” as the target noun. The algorithm functions correctly, but has the possibility to correctly evaluate a nonsensical sentence. It was difficult to find a solution to this without limiting the ability of the Text System. One attempt involved automatically failing the interpret() if two verbs were in the same sentence. This caused more issues as terms such as “running shoes”, and “jumping jacks” would cause the interpret() to fail. The only viable solution was to allow this evaluation to occur. There was not enough time to fix this bug and continue development on other parts of the project.

When testing the Knowledge System a simple rule base was used. The rule base, chest.krb, contained the rule “open”. To test this, the evaluate() method was given the string “open the chest”. When this succeeded a new rule “kick” was added. The evaluate method was then given the string “open the chest”, followed by “kick the chest”. This led to the issue outlined in **4.a Development & Problem Resolution of the Game Engine** section, where the Knowledge System attempted to access increasingly nested rule bases.

In order to test the project as a whole, integration testing was necessary. To perform this a demo environment was set up. This demo environment doubled as a platform for demonstrating the project's functionality and for testing the components as a whole. Integration testing is needed when developing a game engine because while each individual part of the engine may work perfectly in isolation, it is paramount that they function correctly when combined together. After unit testing, each module is added to the demo environment and tested once again in conjunction with all other modules in the demo environment. If the tests fail the module is iterated upon and then goes through more unit testing and integration testing. This process continues until the module works in the demo environment with other modules.

User testing was carried out after several cycles of integration and unit testing. User testing was imperative as a game engine must be usable by developers by providing a sensible workflow and speeding up the development process by a reasonable amount. User testing highlighted several issues involving the presentation of the Text System's output and interactions between the Text System and the Knowledge System.

After multiple iterations of development and testing it was concluded that the system performed the intended functions, but not necessarily as well as desired. The Text System managed to correctly tag the input and interacted perfectly with the Knowledge System.

User testing on the demo environment highlighted issues with the Text System is the default error messages used within the demo environment. Under particular circumstances the error messages produce an output that is not authentic and does not make a lot of sense. In the original iteration of the error message read “You unsuccessfully X the Y”. This led to inaccurate error message, such as when the user input the adjective “unsuccessfully” before their input, causing it to read “You unsuccessfully unsuccessfully X the Y”, which implies that the action was actually successful. After a process of trial and error, the most acceptable default error message reads “Nothing happens when you X the Y”. This caught most incorrect inputs in an authentic and understandable manor. One input which does not work with this is if the user attempts to perform an action on an object that does not exist. Instead of the denying that the non-existent object was in the scene, the message would only say that a particular action could not be performed on the object. This implied that the non-existent object was in the scene.

During integration testing it was found that the Knowledge System would fail to work when presented with a blank field, as it would try to load a null rule base or evaluate a null rule. To fix this, the Text System had to be modified so that if one of the return parameters was empty, it would return an empty string instead of a null value. This resulted in the Knowledge System checking rules and rule bases with the name “ “ rather than checking a null value. This prevented the system from crashing.

However, this led to another problem with the error messages; parts of the sentences would being blank. To fix this, multiple scenarios were set up to cater to different evaluations and inputs; one for a successful evaluation, one for unsuccessful evaluation where a field is blank, and one for unsuccessful evaluation where a field is provided.

System Evaluation

After extensive testing it was found that the NLP functionality was performing to a reasonable degree and was able to successfully interact with the Knowledge System. There were several issues, however. Due to the poor usability of the Pyke API the expert system didn't perform particularly well and felt as though it limited the performance of the system. A more flexible alternative to using Pyke would have been to track facts about an entity by using it's Knowledge Component. Instead of Pyke rule bases which caused more problems than they solved, Python scripts could have been used to provide similar functionality.

Currently the NLP functionality performs as well, but is a somewhat naïve implementation. For most use cases the Text System will wok correctly off the assumption that the sentence is “... Verb … Noun …”, but this doesn't allow for more complicated sentence structures. As established in the **System Testing** section of this chapter, inputs with multiple verbs and nouns can have undesirable outputs. Attempts to correct this either opened up new problems or limited the capabilities of the Text System.

Even the error outputs could not catch all incorrectly evaluated inputs, as highlighted in the **System Testing** section. The only plausible way to check for this without implementing major changes to the Scene class. The most obvious way to do this is to check the scene's entityList, but there is a problem with that. The entityList stores references of the entity and it's memory address, meaning the only realistic way to access this is through regular expressions in the scene (which is undesirable), or by changing the scene class. The scene class would need to be changed so that the entityList was a dictionary, with the name of the entity as the key. Unfortunately, this creates new problems. If this route was to be taken then entities of the same type would have to be stored as a list inside the dictionary. This would make selecting specific ones difficult. This issue was a major development oversight on my behalf. If this was caught earlier in testing it may have been possible to fix it. Instead, it was caught towards the end of the testing cycle during development.

As a game engine, Tall Tales is quite simplistic. Currently, it only supports NLP, scene management and an ECS interface for developing new Systems and Components. It lacks systems that would be considered standard or mandatory, such as; physics systems, camera systems, and 2D & 3D rendering utilities. Instead, Tall Tales relies on the developer utilising PyGame and their own talents to implement these features. As it stands, Tall Tales is a functional game engine that provides developers the ability to easily add systems which are missing, but doesn't provide basic facilities that are considered necessary to speed up game development. This downside is somewhat alleviated by the ECS interface provided. This makes integrating new systems into the game engine faster than it normally would be.

There are some performance issues with Tall Tales during NLP. This is caused whenever an initial rule evaluation fails and the synsets have to be checked. The time for re-evaluation varies greatly depending on the size of the synset that is being re-evaluated. I could not find any obvious way to improve the perform for this as NLP is computationally expensive.

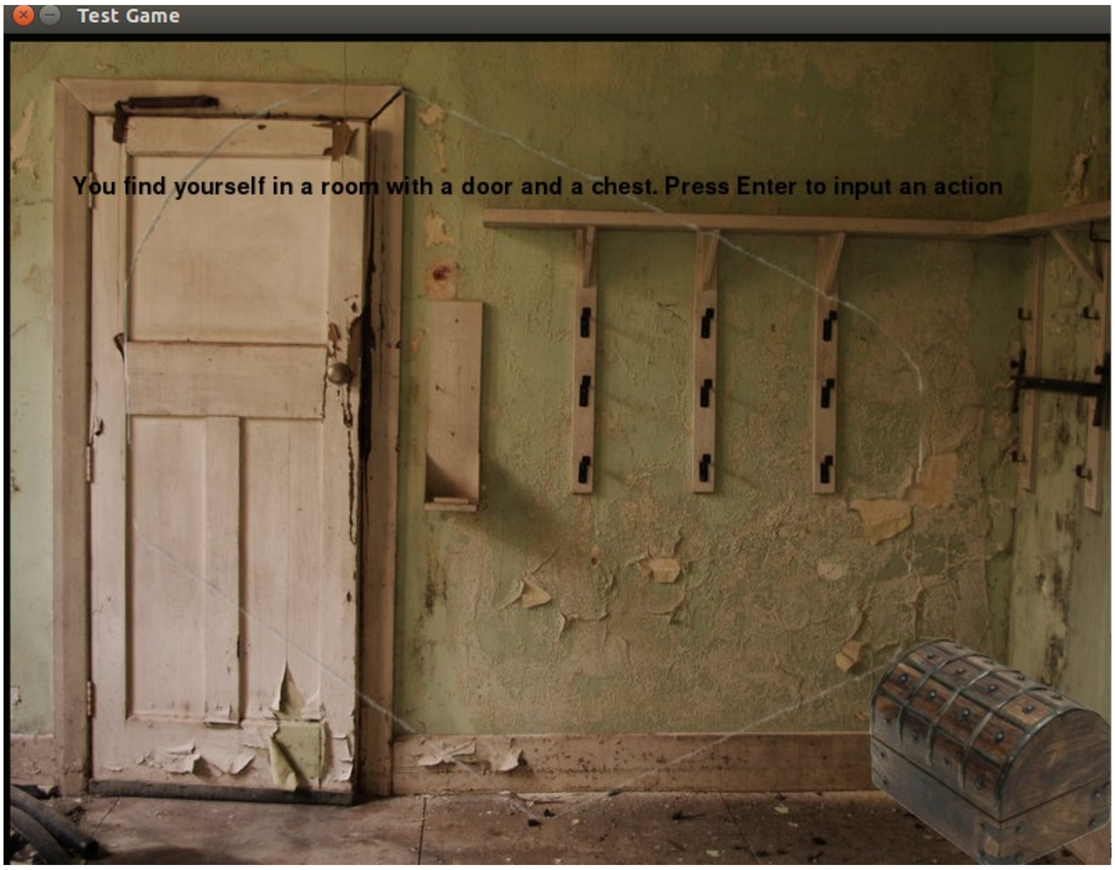
In recent months, the games industry has shifted from using “in-house” game engines (game engines developed for specific games) to licensing existing engines, such as Unreal and Unity. This shift is beneficial for developers of game engines. By having large studios license externally developed engines, the demand for new public domain game engines has increased. This does carry it's own downsides, however. With demand increasing, there is also an increase in the quality expected. Most developers expect a game engine to be as feature-complete as Unity or Unreal. For Tall Tales, this means that development of a fully-fledged usable game engine has become increasingly unrealistic. This is because more is expected of it. At the same time, however, game engines that provide something which is missing from a competitors engine will pique the interest of potential developers. This is Tall Tales' strong point, as no other engine provides NLP utilities. Although, in its current state it would not have any industry appeal, and would be hard-pressed to garner appeal without the support of a large development team.

Demonstration

Currently the demo environment is able to demonstrate that the Tall Tales game engine can accept written input, perform NLP on that input, and then evaluate rules in a knowledge base using the results of the NLP. The source code for the demo environment also demonstrates the usability of the engine and provides a guideline for how to develop using Tall Tales.

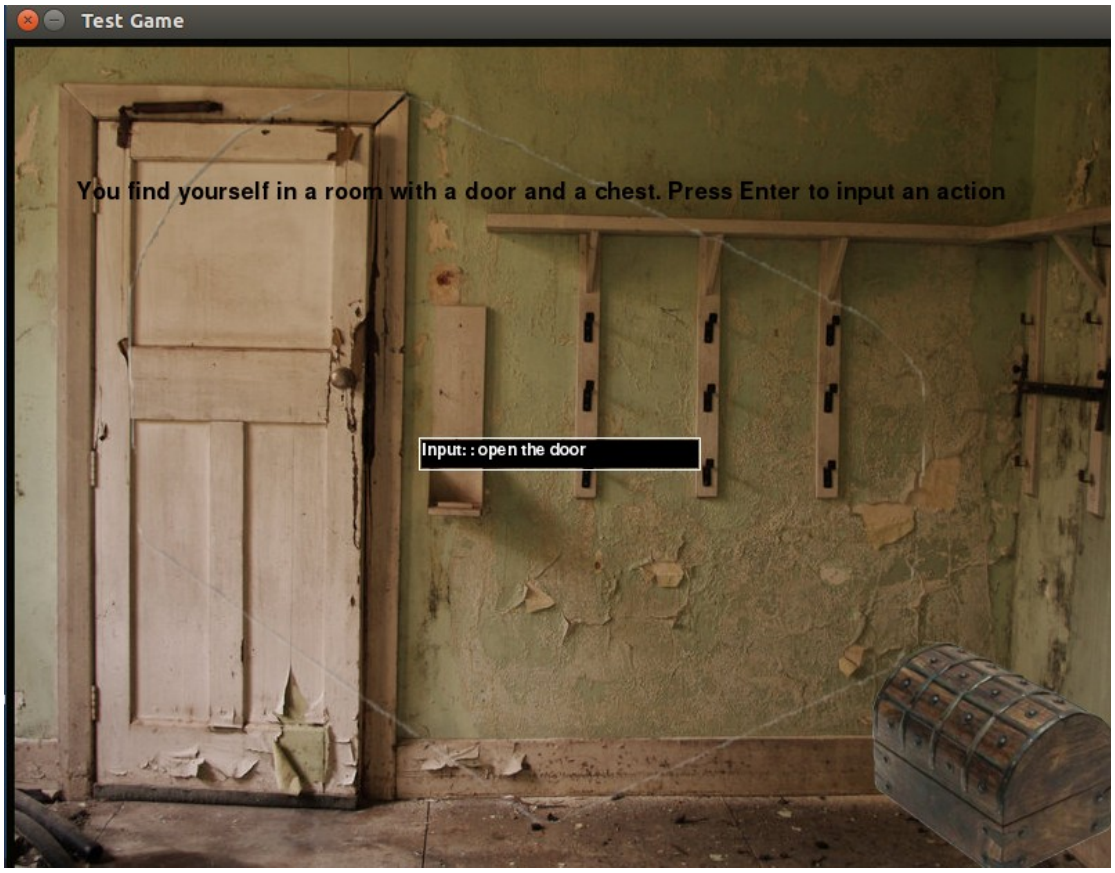
Link to demonstration: https://www.youtube.com/watch?v=BsKD6g6Sw58&feature=youtu.be

The following are a selection of screenshots to demonstrate functionality in the event that the video is inaccessible. These do not provide the full cover that the video provides.



You find yourself in a room with a door and a chest. Press Enter to input an action

Figure 5.1: Demo screenshot 1



You find yourself in a room with a door and a chest. Press Enter to input an action

Input: : open the door

Figure 5.2: Demo screenshot 2

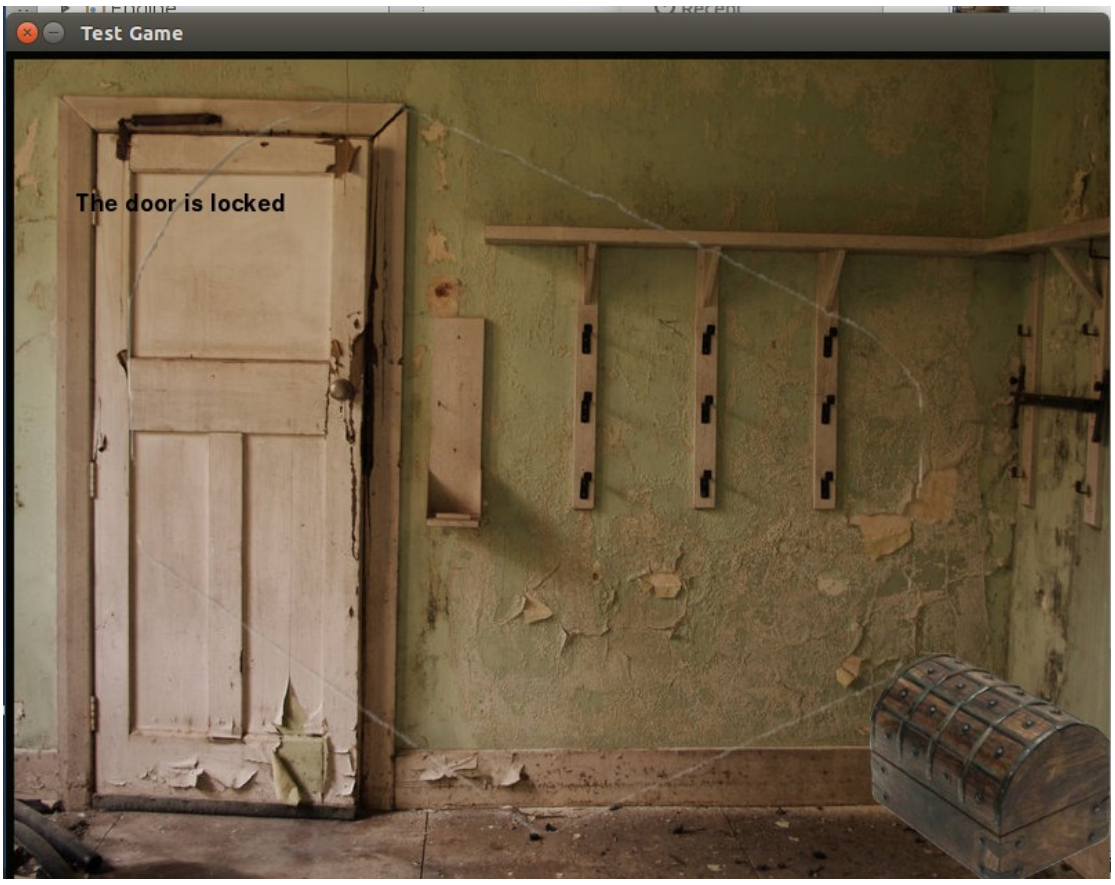
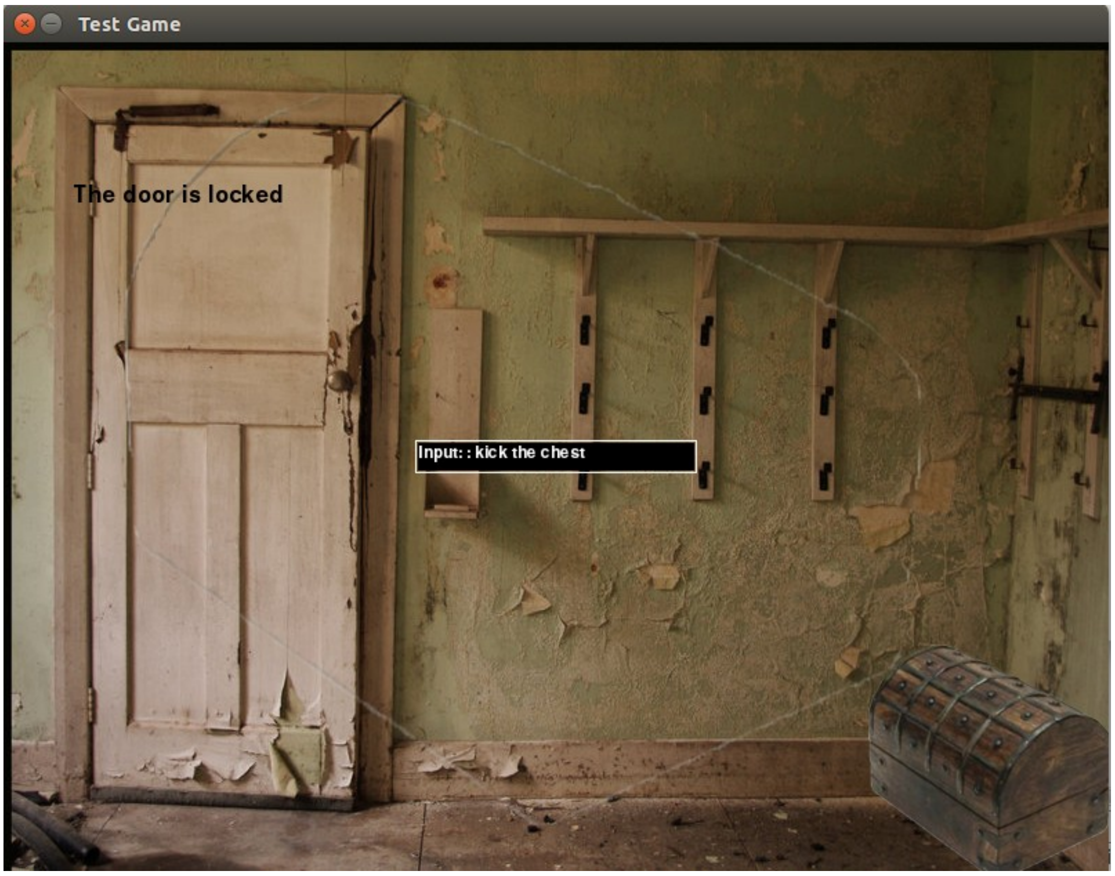
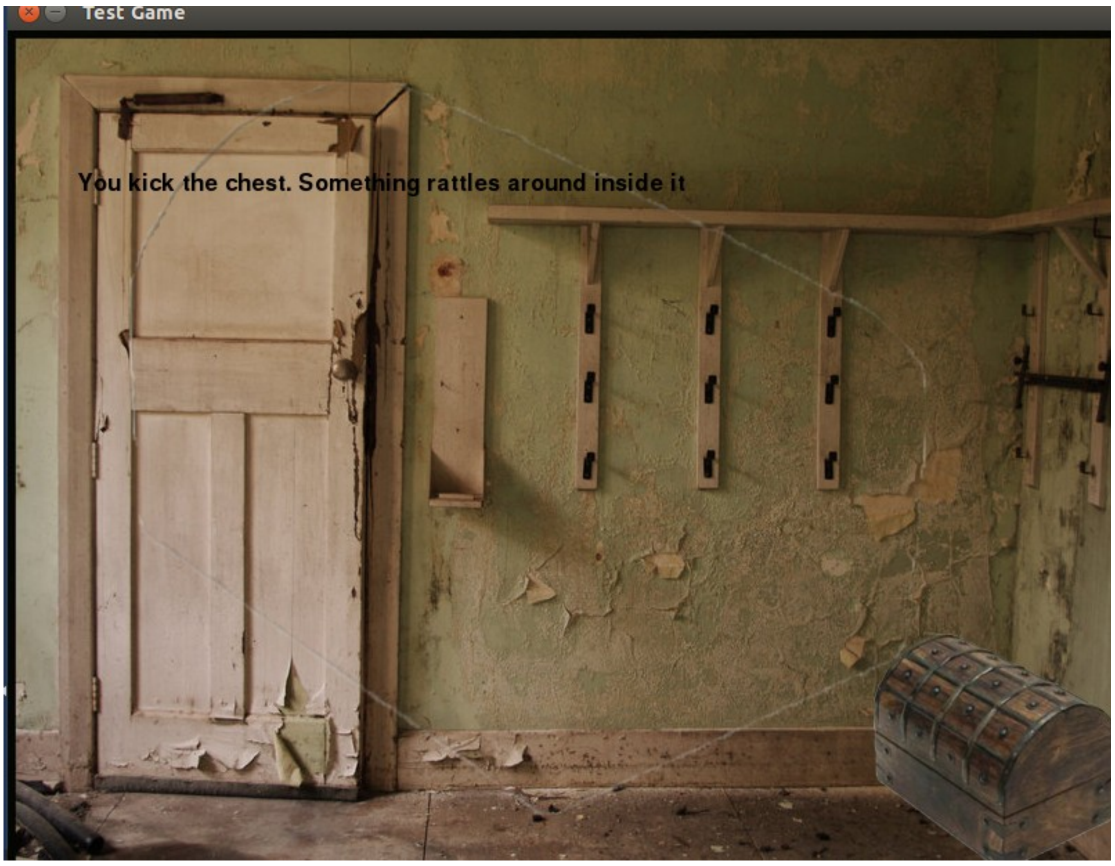


Figure 5.3: Demo screenshot 3



```
Input: : kick the chest
```

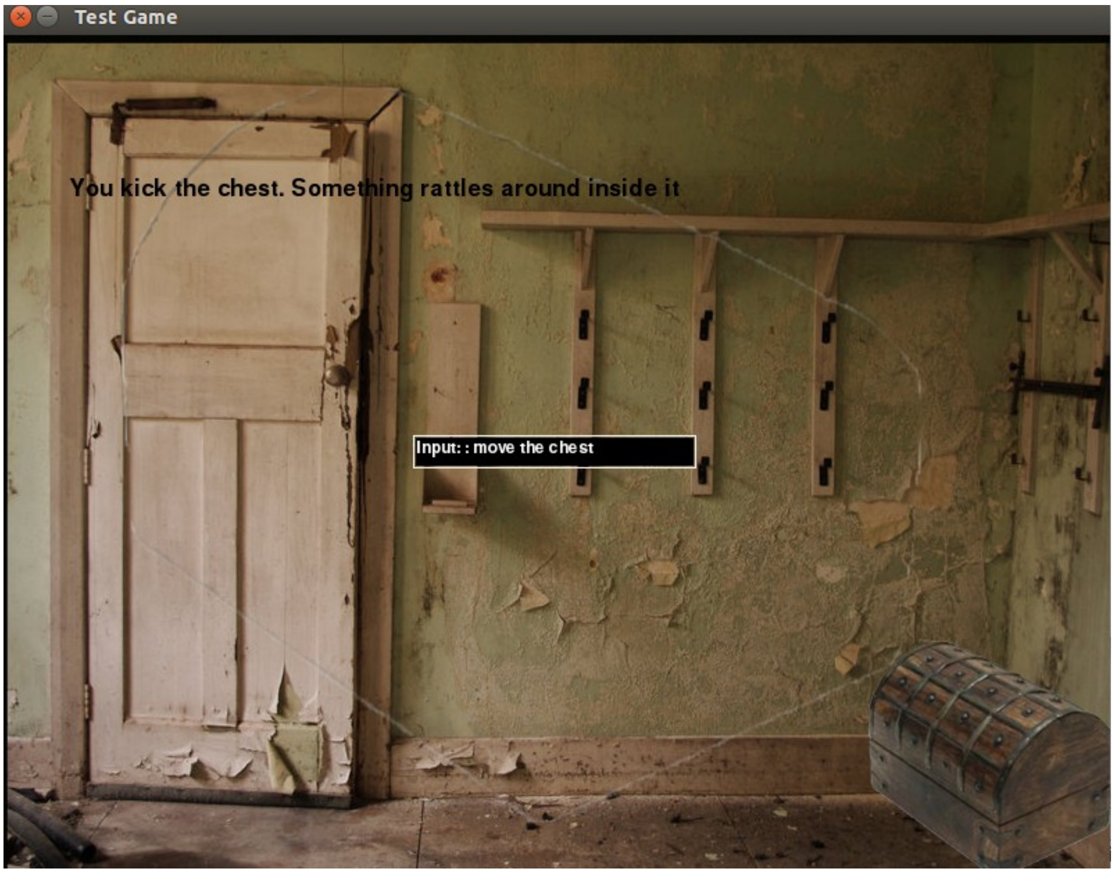
Figure 5.4: Demo screenshot 4



Test Game

You kick the chest. Something rattles around inside it

Figure 5.5: Demo screenshot 5

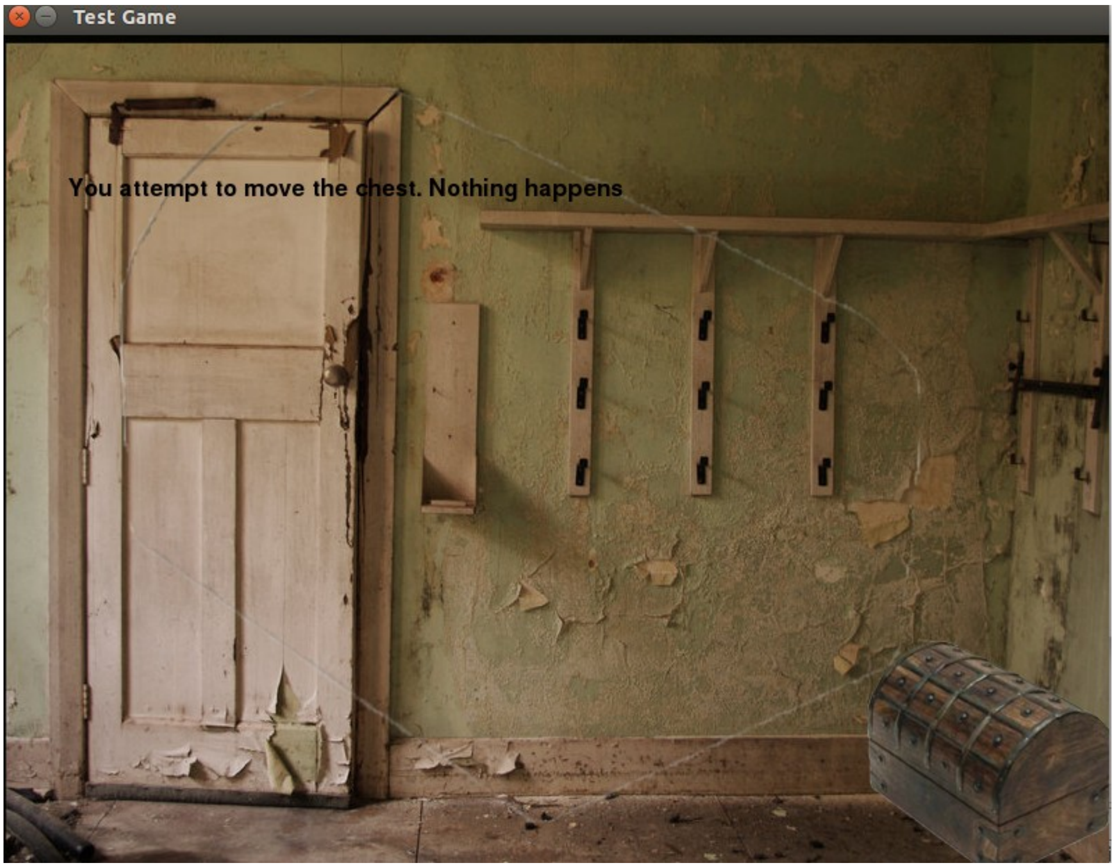


Test Game

You kick the chest. Something rattles around inside it

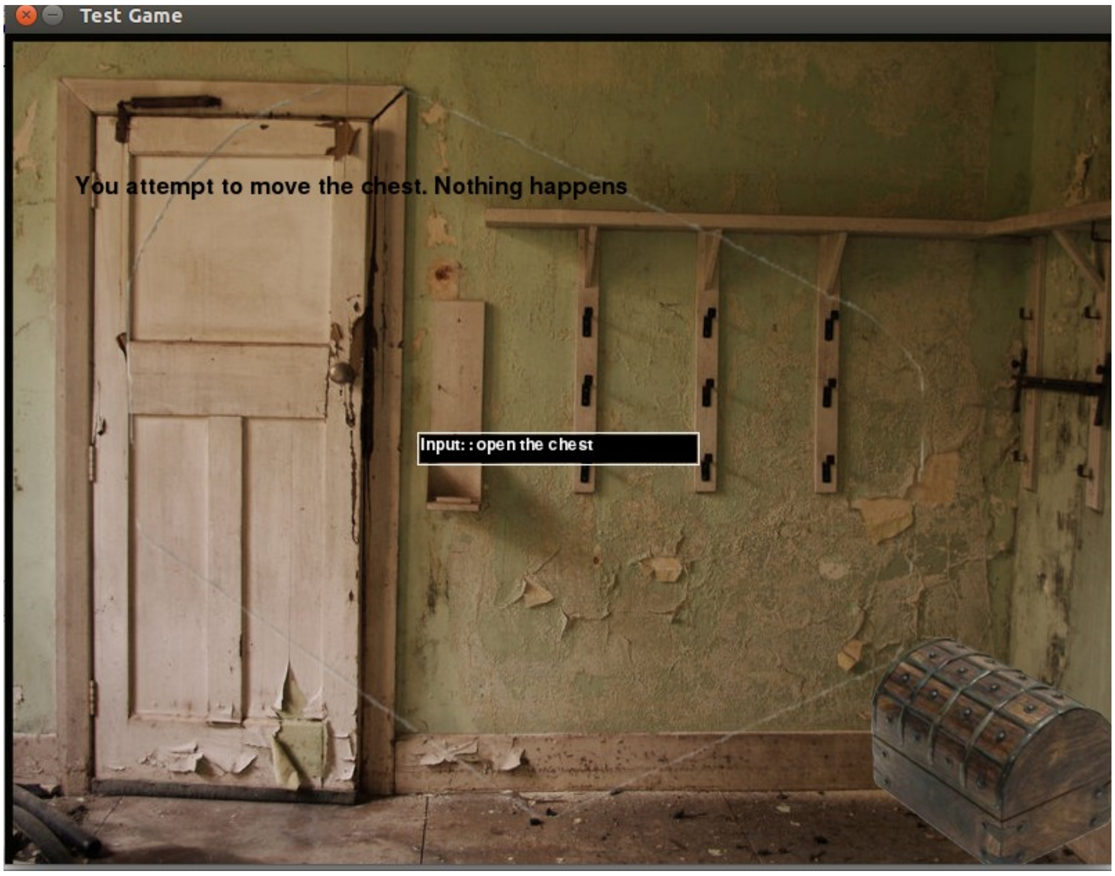
Input: : move the chest

Figure 5.6: Demo screenshot 6



You attempt to move the chest. Nothing happens

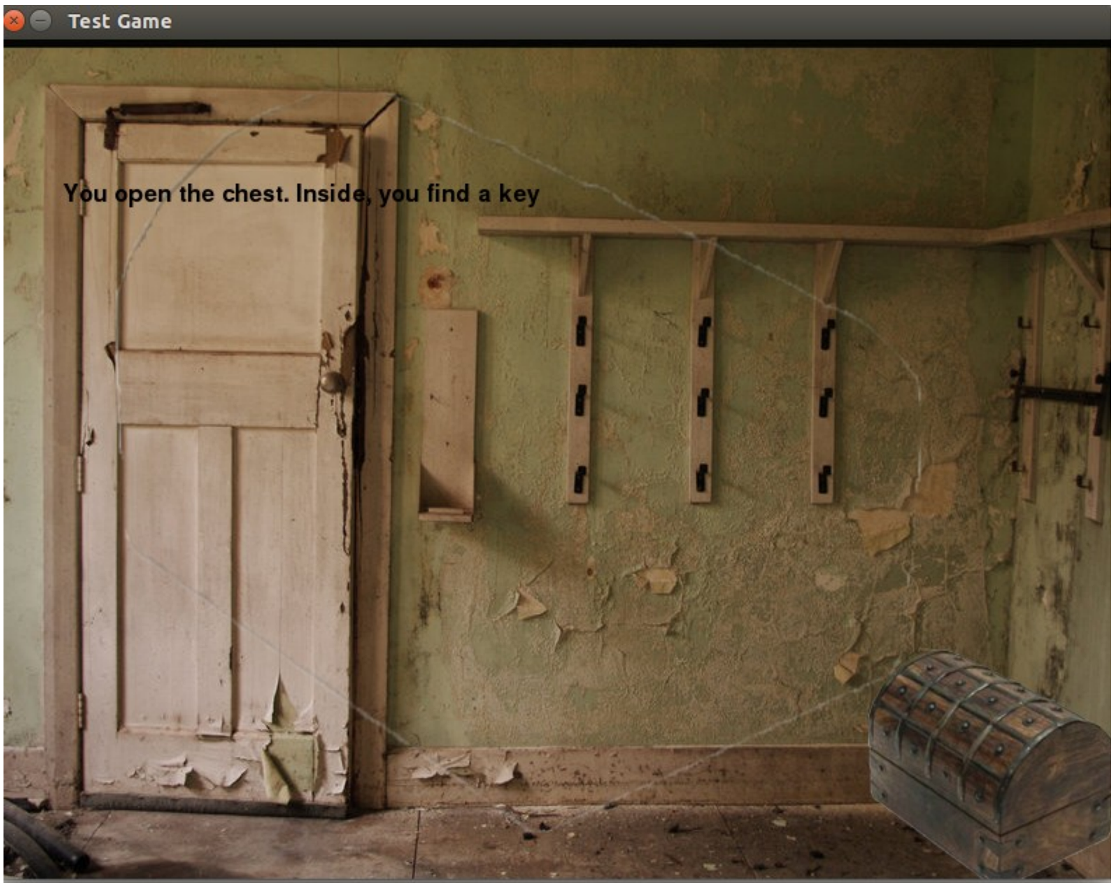
Figure 5.7: Demo screenshot 7



You attempt to move the chest. Nothing happens

Input: : open the chest

Figure 5.8: Demo screenshot 8



You open the chest. Inside, you find a key

Figure 5.9: Demo screenshot 9

6. Project Plan

Analysis of Original Plan & Proposal

Initially Tall Tales was intended to be a game that provided NLP as the main input source rather than a game engine that supported NLP. The change from a game to a game engine was made because there was no support for NLP in industry-standard game engines. This led to the requirement for using Python, as it supported NLP libraries. However, using Python meant that instead of relying on a game engine to develop the game, everything would have to be built from the ground up using a game development library such as PyGame or Pyglet. Because of this, the logical decision was to propose a game engine with NLP instead of a game as the systems used in a game engine would be required to make the original game.

Analysis & Evaluation of Final Project Plan

Table 6.1: Finalised project plan

Deliverable	**Deadline**	**Requirements**	**Testing**
Game Initialisation & Management	18th January	• Start, run and close game • Support for ECS architecture	• Create new game • Print start-up message to console • Print close message to console • Create simple entities on engine start-up • Print contents of entities to console
File system	18th January	• Organise file structure of engine source code	N/A
Scene Management	28th January	• Modify Game class to manage active scenes • Create & define Scene class	• Create two new scenes • Automate changing between two scenes
User Input	3rd February	• Modify Scene superclass to handle events • Modify scene test classes to receive user input	• Switch between two test scenes based on user input
NLP	24th February	• Create TextCompone nt & TextSystem classes • Add NLP functionality to TextSystem	• Enter test string • Print NLP results to console
Knowledge Representation	6th March	• Create KnowledgeCo mponent & KnowledgeSy stem classes	• Create simple knowledge base files • Attempt to access a

		• Add expert system functionality to KnowledgeSystem	specified knowledge base and evaluate specified rules in that knowledge base
Rendering System	TBD (after development of core deliverables)	• Create an interface for rendering 2D objects to the screen • Create RenderSystem and RenderComponent • RenderSystem should draw all entities with a RenderComponent in the current scene	• Utilise Render System to draw simple images to the currently active scene • Switch to a second scene, draw its entities and switch back to the original scene
Physics System	TBD	TBD	TBD
Collision System	TBD	TBD	TBD
Camera System	TBD	TBD	TBD

For the most part, the project managed to stick to the project plan. Deadlines were met ahead of schedule for the first set of deliverables, namely; Game Initialisation, File Systems, Scene Management, and User Input. These provided no major problems. Both Knowledge Representation and NLP went over a week past the deadline due to unexpected errors found during unit testing. NLP was intended to be finished by the 24th of February, but had to be extended until the 4st of March due to several crucial errors that needed to be fixed. This impacted the Knowledge Representation module. Knowledge Representation was meant to be finished 6th of March, but development was extended until the 13th March. This was necessary as there were multiple issues with integrating the Pyke API into the engine. In order to facilitate this stretch goals such as the Rendering System and the Physics System had to be cut from the project as there was no feasible way to implement them and perform testing. The Text System and Knowledge System also had to be developed in tandem due to the scheduling delay. This proved to be useful as results from the Text System's unit tests could be used as input for the Knowledge System's unit tests. This helped to perform a portion of the integration testing slightly ahead of schedule.

The testing schedule was on time. This was a result of performing unit tests throughout the development lifecycle. This meant that the testing period during March was focused on integration, and user, testing.

If I were to repeat the project the Knowledge Representation would be researched more thoroughly in order to avoid complications with the API. Assuming this would minimise problems, there would be development time left to implement various stretch goals of the project, such as a Rendering System.

7. Conclusion

In conclusion, Tall Tales succeeds at interfacing NLP with game development, but it would be hard to consider it a game engine. While Tall Tales provides a standard game engine architecture for developers, it fails to deliver certain facilities than are expected of modern game engines such as; a graphical user interface (GUI) editor, rendering utilities, lighting utilities, and physics utilities. These shortcomings are a result of how time consuming implementing NLP is.

The main learning outcome from this is that NLP is an incredibly complex field. It is extremely difficult to have a a broad NLP system cater to all potential use cases. As a result, the scope of an NLP system must be limited and fine-tuned to match a specific environment. If I were to extend or redo the project I would research more sophisticated methods of performing NLP. While the current NLP used in the project performs as intended and provides an enjoyable gameplay experience, it is not as diverse as I originally desired, and has a very limited use case.

Additionally, I learned that game engines are massive systems that often require a lot of time, or a team of multiple developers, in order to fully realise them. These limitations prevented me from expanding the project beyond the core modules as the original scope was too large for the given time frame. This limitation did provide me with excellent experience in time and risk management. Because of the time constraint I learned to manage time effectively in order to maximise development of a large system in a small time frame, and I learned to mitigate risks by adjusting the scope of the project as needed by cutting unnecessary components.

After the issues with using the Pyke API in the project I realised the importance of extensive prototyping during the research period of the project. On paper Pyke seemed like a perfect fit for the project, but ended up causing more problems than it solved. During user testing testers felt that Pyke hindered development and the same goal could be accomplished by utilising Python scripts and storing an entity's facts in it's Knowledge Component. If more time was spent rigorously prototyping Pyke then a more viable solution could have been found before Pyke began to cause problems. When continuing development I will take the user feedback into account and incorporate it into the Knowledge System.

When testing the demo environment, users were excited to see how the game world would respond to their input, and to see what kind of capabilities they had within the game. This proved that this kind of text-driven gameplay has a lasting appeal with users, especially if it were extended and fully realised.

In future work I will attempt to adapt Tall Tales to existing industry-standard game engines. This is due to the games industry switching from “in-house” game engines to commercial, public domain engines. It is not feasible to develop Tall Tales as a fully functional game engine which can compete with the likes of Unity. Additionally, doing this allows for a focus on NLP within the engine, rather than developing efficient rendering utilities which exist in other engines.

8. Bibliography

Research Sources

1. Frictional Games. 5 Core Elements of Interactive Storytelling. In the Games of Madness. 2013.
2. Klimas C. Twine [Internet]. Available from: http://twinery.org/
3. Ryan M-L. From Narrative Games to Playable Stories: Toward a Poetics of Interactive Narrative. Univ Neb Press. 2009;1:43–59.
4. Mateas M, Stern A. Integrating Plot, Character and Natural Language Processing in the Interactive Drama “Façade.” Carnegie Mellon University; 2003.
5. Bird S, Klein E, Loper E. Natural Language Processing with Python.
6. Bird S, Klein E, Loper E. Natural Language Processing with Python.
7. Princeton University. WordNet.
8. Boas HC. From Theory to Practice: Frame Semantics and the Design of FrameNet. [Austin]: University of Texas; 2005.
9. Godrey LB. The Design and Implementation of a Lightweight Game Engine for the iPhone Platform. [Fayetteville]: University of Arkansas; 2014.
10. Hall D. ECS Game Engine Design. [San Luis Obispo]: California Polytechnic State University; 2014.
11. Wild C. Adrift [Internet]. Available from: http://www.adrift.co/cgi/adrift.cgi

12. Snozbot. Fungus [Internet]. Available from: http://fungusgames.com/

13. Klimas C. Twine [Internet]. Available from: http://twinery.org/

14. NLTK Project. Natural Language Toolkit [Internet]. Available from: http://www.nltk.org/

15. Frederiksen B. Pyke [Internet]. Available from: http://pyke.sourceforge.net/

16. Open Source. PyGame [Internet]. Available from: http://www.pygame.org/news.html

17. Open Source. Pyglet [Internet]. Available from: http://www.pyglet.org/

18. Open Source. Love2D [Internet]. Available from: http://love2d.org/

19. Unity Technologies. Unity [Internet]. Available from: http://unity3d.com/

20. Sotirovski D. Heuristics for Iterative Software Development. Inst Electr Electron Eng. 2012 Jan 1;18(3):66–73.

9. Appendix

Abbreviations

1. NLP: Natural Language Processing
2. API: Application Programming Interface
3. GUI: Graphical User Interface
4. POS: Part of Speech

Figures

<u>Figures 3</u>

<u>Figures 4</u>

Tables