Matt Filer
Department of Computer Science and Creative Technologies
University of the West of England
Coldharbour Lane
Bristol, UK
matthew2.filer@live.uwe.ac.uk

Developing a tool to produce modern narrative-driven text adventures

The purpose of this report is to outline the process of creating designer-focussed tools that can be used as a pipeline to develop modern narrative-driven text adventures. This report will discuss the project's inspirations and explain the creation of an agnostic tool to drive three logic implementations across different engines, appraising and comparing each result along with its suitability in a larger game project.

1) Starting out

Back in a time where processing power was at an all-time low, it can be argued that there was no better way to allow users to interact with and explore a fictional place than through the medium of an interactive fiction game. This could be due to the fact that such a simple platform provided a far richer experience than reading a book and allowed users to make their own choices within the story. The first computer-based text adventures emerged in the late 1960s to early 70s before the advent of personal computers. These games were playable on 'mainframe' systems, the first of which being heralded as *Wander*, developed in 1974 – although the true first text adventure is somewhat debatable (Tanbusch, 2015). Many early text adventures were created by individuals and undocumented, or simply lost over time as technology has evolved.

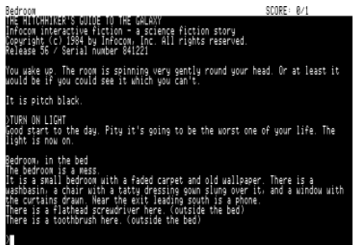
Figure 1 Colossal Cave Adventure is played on a *VT100* serial console by Meutia (2017).

In modern gaming culture text adventures are seen as a dying breed, as easy access to professional game engines such as Unreal and Unity make it easier than ever for indie developers to create fully fledged 3D titles, as explained by Haas (2014). Years prior, the cost of an engine license and extra development time required for a 3D experience may have favoured creating an interactive fiction game, however this is simply no-longer the case. Tied with ever shortening attention spans of a modern audience, text adventures are slowly becoming a thing of the past. Despite this, the *Interactive Fiction Competition* is an annual competition for text adventures, currently in its 24th year, which still boasts an impressive entry pool (Alexander, 2014). Modern adventure engines such as *Inform 7* also still attract a wide audience, while web-based text adventures prove to be a popular genre to a niche audience capitalising on cross-platform play, allowing access to a new mobile audience that traditional command line based adventures have trouble accessing (Zurawel, 2017).

In 2016, text adventures were brought back into the spotlight with the release of *The House Abandon* during the 36th *Ludum Dare* game jam. This game, created with Unity, was set in a 3D environment and progression in the text adventure affected the player's surroundings, revealing more about the story (Ludum Dare, 2016). The unique interaction between text game and 3D environment allowed it to become a metagame where the user played a 2D text adventure of the 3D game they were currently in. This unique experience quickly became noticed and was eventually picked up by publisher *Devolver Digital*, turning into a 4-part full release named *Stories Untold*, featuring a series of different computer-based puzzle games (Gamasutra, 2017).

Figure 2 The environment of *The House Abandon* during its boot sequence, by MacLeod (2017).

2) A narrative driven approach



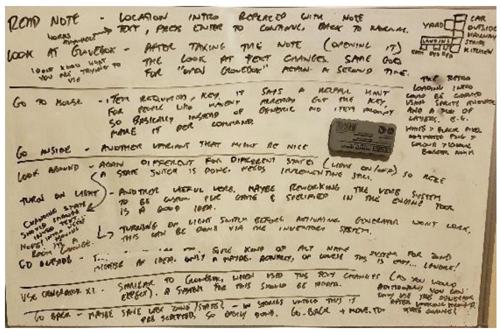
```
Bedroom                                                    SCORE: 0/1
THE HITCHHIKER'S GUIDE TO THE GALAXY
Infocom interactive fiction - a science fiction story
Copyright (c) 1984 by Infocom, Inc. All rights reserved.
Release 56 / Serial number 841221

You wake up. The room is spinning very gently round your head. Or at least it
would be if you could see it which you can't.

It is pitch black.

>TURN ON LIGHT
Good start to the day. Pity it's going to be the worst one of your life. The
light is now on.

Bedroom, in the bed
The bedroom is a mess.
It is a small bedroom with a faded carpet and old wallpaper. There is a
washbasin, a chair with a tatty dressing gown slung over it, and a window with
the curtains drawn. Near the exit leading south is a phone.
There is a flathead screwdriver here. (outside the bed)
There is a toothbrush here. (outside the bed)

>
```

Figure 3 The opening section of the *Hitchhiker* text adventure from My Abandonware (2011).

One thing that set *The House Abandon*'s mechanics apart from the rest is how story driven its approach was. In comparison to traditional text adventures that use a grid system the player navigates through using cardinal directions picking up items, *The House Abandon* had a space that the player navigated through with commands. These commands also acted as an ability to look around the environment, interact with objects and pick up items. This method of interaction is similar in style to 1984's *Hitchhiker's Guide to the Galax*y text adventure, which was also a narrative driven adventure where the player was in the universe of the book series (Webster, 2014). It could be argued that this is a far more engaging experience than the cardinal design, as the user feels less constrained by the game logic and are instead navigating through known areas rather than simply commanding "NORTH" or "SOUTH". After speaking with developer of *The House Abandon* Jon McKellan about his thought process behind this decision, he explained that the idea for the command-based system was to make the gameplay less "sterile" and to allow the player to build up the world in their mind rather than have it effectively spelled out for them through a direction system. This idea was referred to as "taking the house off the grid" and although received backlash from fans of traditional text adventures, the mechanic allowed the player to effectively create their own idea of the game's environment - as Jon says, "its *your* house in many ways" (McKellan, 2019). There is also proof that this system works - the *Hitchhiker* text adventure sold over 300k copies, large for the time (Ciraolo, 1985). The success of *The House Abandon* and its subsequent acquisition by *Devolver Digital* is also further evidence to this point.

"We have a hall that definitely leads to a kitchen, but what that looks like is down to the player." – Jon McKellan

The requirements for this narrative approach include the need for the ability to: set up zones in an environment with large text outputs in response to entering the area and performing actions, easily modify the level layout and parameters, view an overview of the game logic in a simple interface that is designer friendly. This modular approach leads into the question of how it should be implemented. Should there be an implementation that is engine agnostic? Should each implementation be engine-specific and utilise the advantages of its features?



READ NOTE - LOCATION INTRO REPLACED WITH NOTE TEXT, PRESS ENTER TO CONTINUE, BACK TO NORMAL.

LOOK AT GLOVEBOX - AFTER TAKING THE NOTE (OPENING IT) THE LOOK AT TEXT CHANGES. SAME GOES FOR "OPEN GLOVEBOX" AGAIN A SECOND TIME.

GO TO HOUSE - ITEM REQUIRED, KEY. IT SAYS A HELPFUL HINT FOR PEOPLE LIKE [illegible] ALREADY GOT THE KEY, SO BASICALLY INSTEAD OF GENERIC NO ITEM PROMPT MAKE IT PER COMMAND

GO INSIDE - ANOTHER VARIANT THAT MIGHT BE NICE

LOOK AROUND - AGAIN DIFFERENT FOR DIFFERENT STATES (LIGHT ON/OFF) SO MORE A STATE SWITCH IS DONE. NEEDS IMPLEMENTING STILL.

TURN ON LIGHT - ANOTHER USEFUL VERB. MAYBE REWORKING THE VERB SYSTEM TO BE CUSTOM PER GAME & SPECIFIED IN THE ENGINE TOOL IS A GOOD IDEA.

↳ TURNING ON LIGHT SWITCH BEFORE ACTIVATING GENERATOR WONT WORK. THIS CAN BE DONE VIA THE INVENTORY SYSTEM.

GO OUTSIDE - [illegible]

USE GENERATOR X1 - SIMILAR TO GLOVEBOX, WHEN USED THE TEXT CHANGES (AS YOU WOULD EXPECT). A SYSTEM FOR THIS SHOULD BE [illegible]

GO BACK - MAYBE SAVE LAST ZONE/STATES? - [illegible] GO BACK + MOVE TO [illegible]

Figure 4 The logic of *The House Abandon* is deconstructed for a second time.

To answer these questions, the logic of *The House Abandon* was dismantled to see how it could possibly utilise the idea of bespoke implementations over multiple engines. Playing through the game every possible input was tested for five starting areas and the responses to each were recorded. The basic player movement mechanics of the game were then analysed and informed guesses of how the systems were operating under the hood were made. Drawing all of these findings out into a tree led to answer the question of how this should be approached – an engine agnostic tool that produces a parse-able resource able to be loaded into a number of logic implementations with the same experience across each. Bespoke implementations seemed like it would overcomplicate the solution when an agnostic tool could provide enough logic to each. Utilising 2D and 3D engines could allow for different visual outputs, such as a 3D environment or flat 2D screen, but the logic could easily be the same across all the implementations, handled by an agnostic tool that outputs an importable resource.

3) An agnostic implementation

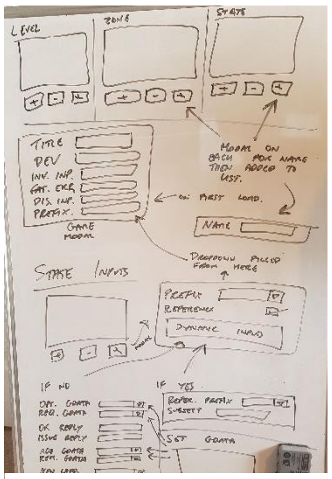


Figure 5 Editor ideas are expanded upon for potential workarounds to *Brainiac*.

Initially, XML was planned to be used to implement this agnostic logic script. An open source tool for modifying behaviour trees known as *Brainiac Designer* by Kollmann (2008) was picked as the graphical XML editor because its linear node interface was perfect for creating zones and scripting logic per commands in an organised fashion. Unfortunately however, after attempting this solution an issue was encountered; XML is considered somewhat outdated at this point and turned out to be far bulkier than needed for the project. Additionally, XML support in C++ is very poor and requires confusing library workarounds such as *'RapidXML'* by Kalicinski (2006) to parse, which were hard to implement.

This XML appraisal eventually led to JSON, which is effectively the modern replacement of XML that is lightweight and far easier to read which is a bonus as it allows for easier debugging, as explained by Nurseitov (2009). The major pros to JSON are that it is supported natively in Unity with extended features via an additional script, and can easily be parsed in C++ with a plugin known as '*JSON for Modern C++*'. There is no native support for JSON in Unreal Engine, however a plugin named '*JSONParser*' by Socke (2017) provides basic enough functionality for this project's requirements.

To test these solutions out, a basic JSON script was created in '*Notepad++'* by Ho (2003) and logic written in each of the three engines to parse it and display a line of text.

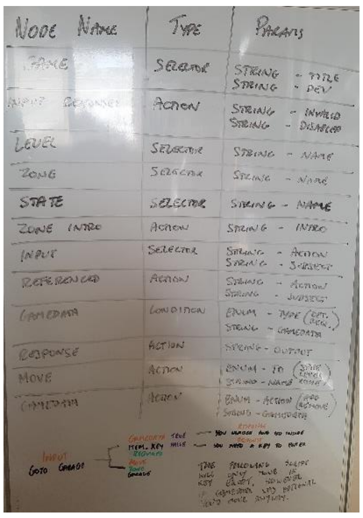
This worked perfectly, and each logic implementation read from the same JSON file with no issues. The JSON script was then extended to allow for options such as "LOOK AROUND" and "GO TO" within each location, implementing logic to suit in each engine – this played well, giving the same experience in each.

This then left the issue of the JSON editor GUI. Due to time restraints on the project, effort wasn't planned to be wasted to build a node editor from scratch. A series of online solutions such as '*JSONViewer*', '*JSONmate*', and '*JSON GUI*' were tried, but none of these provided the designer-focused interface that the project required. Most gave only a series of text inputs that build up a JSON structure. It could be argued that there is very little originality with the interfaces and that the "one size fits all" approach that they take therefore didn't seem right therefore a resolve to this could be an easy to read, neat, bespoke, and simple tool that is not an ugly programmer-focused design was key for the aim of the project. This tool is intended to be used by game designers and writers to create narrative-driven adventures, not programmers. As explained at *GDC 2012* by *Valve* developer Ruskin (2012), writers should be given "tools, not homework".

"The intersection between good programmers and good writers, while non-zero, is pretty small." – Elan Ruskin

Brainiac Designer kept seeming like the only option for this solution for the reasons mentioned previously, and eventually it was decided to write a converter to parse *Brainiac*'s XML output into a JSON file that could be read into the game logic implementations. This script worked effectively, even allowing for very basic validation of the user's *Brainiac* node layout to catch any issues before running the game.

Unfortunately, using this converter felt like a strange workflow, to the point where it felt like a hacky solution. The experience was intended to feel more consolidated, so a launcher application was developed that would handle the creation of *Brainiac Designer* projects, allow localisation of strings, and additionally allow for a more streamlined workflow for switching projects and converting the XML script to JSON for any implementation. Using C# the launcher was created which handles the aforementioned tasks, and utilising the open source nature of *Brainiac Designer*, the program was modified to load the active project from the launcher immediately – giving a unified experience across the tools.

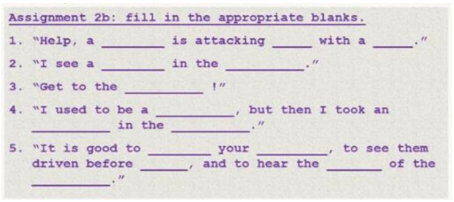


NODE NAME	TYPE	PARAMS
GAME	SELECTOR	STRING - TITLE STRING - DEV
INPUT RESPONSES	ACTION	STRING - INVALID STRING - DISABLED
LEVEL	SELECTOR	STRING - NAME
ZONE	SELECTOR	STRING - NAME
STATE	SELECTOR	STRING - NAME
ZONE INTRO	ACTION	STRING - INTRO
INPUT	SELECTOR	STRING - ACTION STRING - SUBJECT
REFERENCED	ACTION	STRING - ACTION STRING - SUBJECT
GAMEDATA	CONDITION	ENUM - TYPE (GET, REQ.) STRING - GAMEDATA
RESPONSE	ACTION	STRING - OUTPUT
MOVE	ACTION	ENUM - TO (STATE, LEVEL, ZONE) STRING - NAME
GAMEDATA	ACTION	ENUM - ACTION (ADD, REMOVE) STRING - GAMEDATA

Figure 6 A plan of *Brainiac* nodes and their parameters.

With the working launcher, converter, and Brainiac interface; the functionality of the game logic began to be extended. The idea of "game data" was introduced, allowing a string to be saved into the game's memory which can then be queried through a condition node in order to allow for item pickups and state changes depending on the user's prior actions. To handle user input processing, initially a node was used which allowed a selection from a set of input commands such as "GO TO", "LOOK AROUND", "OPEN"; with a following child node that allowed a text input for the 'subject' – for example, the parent node could be "OPEN" and the child could be typed as "GLOVEBOX". This system worked well, but through playtests of the game it felt restrictive in nature and often left people unsure of what commands to use as they were not bespoke to the implementation.

To solve this issue, the parent node was changed from a pre-defined set of options, to another text input; this allows any 'action' to be paired with any 'subject' - for example, "DRIVE" or "EXPLORE" which were previously not able to be used. Handling this extended 'action' set required an update to the XML to JSON converter to pick out every 'action' keyword and push it to an array in the JSON data structure. This array is then read in by each implementation and used to match the user's input, rather than having a pre-defined set of 'action' keywords. Going back to Ruskin's *GDC* talk (2012), writers should not be cornered into the way a system works, they should be free to be creative – "empower your writers". The use of this system needs to keep in mind inconsistencies, however, between the hint of the command to use and the actual command, as explained in a report by Gabsdil (2001).



Assignment 2b: fill in the appropriate blanks.

1. "Help, a _______ is attacking _____ with a _____."
2. "I see a _______ in the _________."
3. "Get to the _________ !"
4. "I used to be a _________, but then I took an _________ in the _________."
5. "It is good to _______ your _________, to see them driven before ______, and to hear the _______ of the _________."

Figure 7 Writers should not be driven by programmers, from Ruskin (2012).

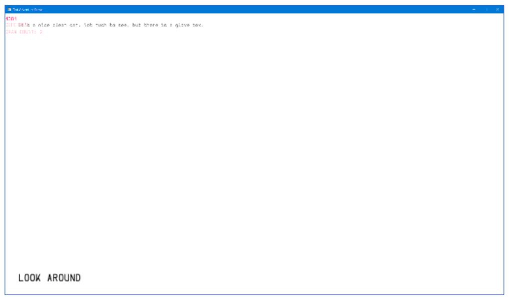
In testing the implementation, the opening of *The House Abandon* was re-created to prove that the toolset works effectively. This worked well after the further additions to the tools, being very quick and simple to produce four to five zones from the game within a short space of time. Following this, an original short game based in the *Alien* universe was created to form an original demo piece for the toolset.

4) Implementing in OpenGL

To implement game logic in OpenGL, the barebones engine framework *ASGE* by Huxtable (2017) was used. This framework allows easy access to basic initialisation, update, and render functions as well as event listeners for key presses and simple sprite handling. Sprites were first created for both the foreground and background of the game, which were planned to be used across all three implementations. Importing these assets was easy as *ASGE* uses a framework known as '*FileIO*' which produces a virtual filesystem from a compressed archive file at game initialisation. From this filesystem, assets are read-in and *ASGE* allows support for a number of popular file formats. If working with just OpenGL itself, there are a number of libraries such as *'libjpg'* and *'lodepng'* that can provide similar basic support to load images into engine memory for use [Image Libraries 2018].

Positioning these sprites in *ASGE* is as simple as calling a function for X and Y position. For both background sprites these positions were set as default 0 and the game window was then resized to the size of the sprites making them cover the whole game viewport.

Adding text instances in *ASGE* is also somewhat simple although positioning them without a graphical interface can become a pain, and without the ability to align the text left or right this aspect was the most limiting part of the whole implementation experience. Graphical engine development tools available with the other two implementations in Unreal and Unity were absolutely more useful for this. Assigning a font to the text was possible through *ASGE*'s *loadFontFromMem* function which loads a TTF font from the virtual filesystem into the engine's memory which can then be assigned as the current active font for displaying text. Again, while not as simple as a more modern engine's solution, the process is straightforward and reliable.



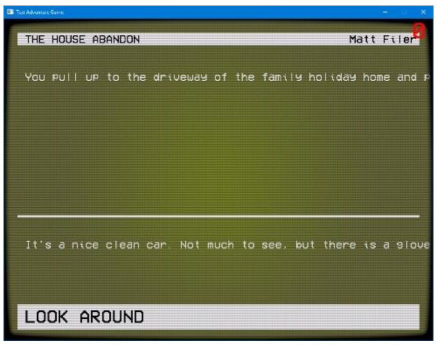
It's a nice clean car. Not much to see, but there is a glove box.

LOOK AROUND

Figure 8 An early XML parsing test in *ASGE.*

Another issue with this low-level engine is that the text support doesn't have any build-in word wrapping like Unity or Unreal can provide. Experiments with implementing automated line wrapping were tried, but as the font used doesn't have uniform characters this became tricky to perfect, so eventually the tools were adjusted to make it simple to understand where to manually enter line breaks. These line breaks are then encoded to the newline character *\n* which allows *ASGE* (along with the other implementations) to continue text on a new line.

As *ASGE* has no notion of text inputs like the more complete modern engine solutions have, a key handler event was relied upon along with a character array to build up the user's text input. The character value of the key is added to the character array, with exception of the backspace key which removes the last array entry, and the enter key which triggers the game logic to begin and clears the array on completion. The array is then displayed on screen in the same way that the other text outputs function. This low-level solution for handling input was arguably better to work with as you get far more control than a more higher-level engine provides, also having to manually manage the input's character array allows you greater flexibility with memory management and what action is given to certain inputs.



THE HOUSE ABANDON Matt Filer

You pull up to the driveway of the family holiday home and p

It's a nice clean car. Not much to see, but there is a glove

LOOK AROUND

Figure 9 The first interface design in *ASGE.*

To parse the JSON logic, a C++ library known as '*JSON for Modern C++*' by Lohmann (2014) was utilised. This library provides a robust solution for parsing a JSON file stream into an easily navigable data structure through an array-style system. All logic was handled within a separate class file in the *ASGE* implementation in preparation for a wider game implementation. This is good as C++ is a pretty standard language to use in industry, so the likelihood of this script working in a range of low-level implementations is high with minor modifications. Loading the JSON file into memory for the library to read was simple, the file was read into a buffer from the virtual filesystem, converted to a character array, and then converted to a stream using the "stringstream" functionality from C++'s standard IO library. This functionality may have to be changed if the script was used in another low-level engine, as most differ with their ways of reading in data.

Parsing user input was achieved through C++'s standard string library. Input actions were matched against the user's input by trimming it with "substring" to the lengths of each entry in the action array pulled from the JSON file. If the trimmed string matched the action array, that subsequent logic was then performed. A modifier to this logic was added to prioritise longer action matches - the reason being that since the action array is dynamic and could be anything, one entry could be "GO TO" and another could simply be "GO", yet the script could match the shorter action as the input by mistake.

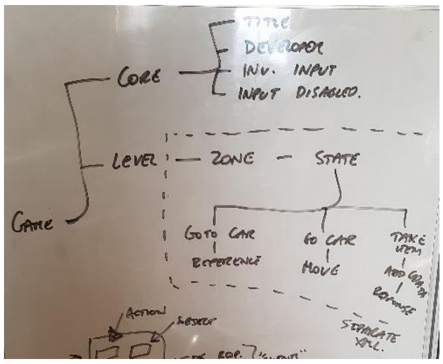


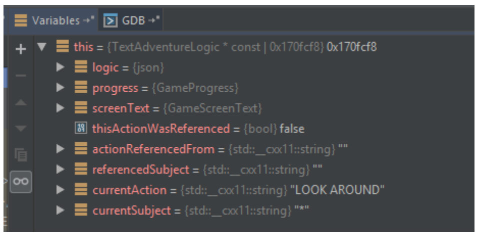
Figure 10 An early outline of the XML/JSON structure to use.

The logic is then acted upon by matching the remaining text in the user's input (after the action is removed) with the JSON branch off of the matched input action. If no branch is found, a message is shown to the player that the input is an invalid command. If logic is found however, it is acted upon. Firstly, a decision node is checked for to see if any "game data" is needed to be queried. "Game data" is stored in a fixed-size array which is looped through to check for existing entries and the first empty space to store new data within. Depending on if the array contains the queried data depends on what happens next - if the data was required and not found, the appropriate message is output to the player and no further logic is run. If the data is found however, or not required to continue, the additional logic is run on the branch. This logic includes adding/removing "game data", moving location, or signifying a game over.

One issue with *ASGE* is its lack of in-built audio support. Using engines such as Unity and Unreal adding audio is simple, however with *ASGE* this was not the case. Trying to find a lightweight audio library that supported *CMake* proved difficult, and in the end audio support was dropped from the game. Prior to this, an older version of *ASGE* was used which was compatible with *Microsoft Visual Studio*. Having the increased flexibility of *Visual Studio* allowed for a number of sound libraries to be supported, and as a result functionality for the library '*irrKlang'* was added which worked well to provide a simple multi-layer sound API which was easy to call through a single function after initial instantiation. Unfortunately, with the updated *ASGE* version using *CMake*, '*irrKlang*' was unsupported due to a change to 64-bit from 32-bit.

The new *ASGE* update was chosen over the *Visual Studio* build due to its '*FileIO'* implementation providing the virtual filesystem which makes the game more portable than the previous version through a single archive file.

The IDE chosen to be used with this new *CMake* build of *ASGE* is *CLion* by JetBrains (2016). *CLion* provides a range of helpful features such as Git integration, smart code navigation (the ability to jump to declarations or other uses of functions, as well as the definition of variables and their uses), easy refactoring, and the ability to link to documentation for smart code hinting. The debugger in *CLion* is also quite helpful, providing you with all stored memory at the time of a breakpoint in an easy to navigate table – it can be argued that this is an advantage over *Visual Studio* as that IDE requires you to hover over each variable to see its current value at a breakpoint. *CLion* also provides the option to run without the debugger attached for better performance. Navigation through project source files is easy with *CLion*'s system folder integration, allowing you to see all files within a folder at once, and create new ones if required. This is different to *Visual Studio*'s method of splitting header and source files, and arguably makes code easier to navigate if folders are used effectively. With a large project, navigating through the header/source tabs can become a nuisance which this cuts out.



```
this = {TextAdventureLogic * const | 0x170fcf8} 0x170fcf8
  logic = {json}
  progress = {GameProgress}
  screenText = {GameScreenText}
  thisActionWasReferenced = {bool} false
  actionReferencedFrom = {std::__cxx11::string} ""
  referencedSubject = {std::__cxx11::string} ""
  currentAction = {std::__cxx11::string} "LOOK AROUND"
  currentSubject = {std::__cxx11::string} "*"
```

Figure 11 A look at part of *CLion*'s breakpoint debug output.

5) Implementing in Unity

Unity was chosen early on for its in-built JSON serialising feature, however in practice this turned out to be somewhat useless for the intended purpose. Unity's default serialising functionality relies on the production of C# structures to match the JSON structures, which for this project's dynamic JSON file turned out to be confusing to understand and hard to reliably match. After a short time browsing the Unity wiki however, a script known as *'SimpleJSON'* by Bunny83 (2012) was found which allows very similar functionality to *'JSON for Modern C++'* in the *ASGE* implementation. After importing this script, the process of creating the logic for this game was very similar to that of the *ASGE* implementation. The *ASGE* code was almost replicated line for line to create the Unity game, however some elements were changed to utilise C#'s features, such as using lists for "game data" instead of a fixed array. The advantages to this are obvious with support by default to search the list, add/remove data through a function call, among others.

Unity's interface for editing the UI layout is quite intuitive although feels outdated compared to Unreal Engine. It can be argued that the anchor points are particularly confusing within the editor, and eventually it was opted for the project to run at a fixed resolution so that responsive anchoring was not required. General ability to create the UI is quite simple, acting similar to Unreal Engine in that you place a canvas and populate it with the required elements. For the Unity implementation, UI Text elements were used for all dynamic text which worked well as it is simple to call from the C# script to update via Unity's drag-and-drop referencing system. Creating a public instance of the Text class within a C# script allows you to simply drop a UI Text element onto the script's GUI counterpart in the editor and have access to that element within the script. This functionality is very easy to learn and is a nice visual way to see what elements you are accessing when scripting.

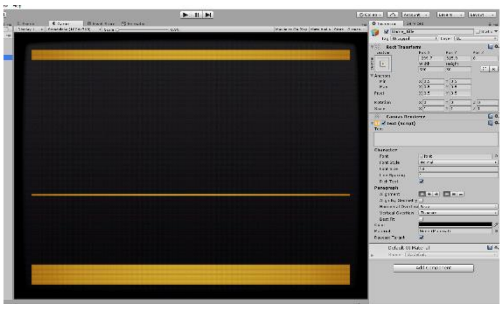
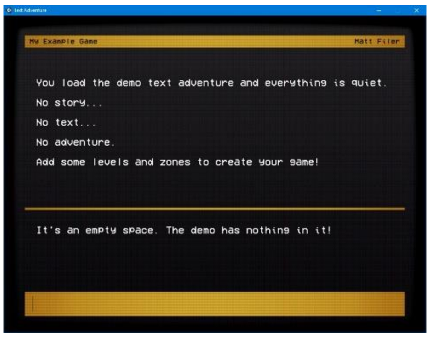


Figure 12 The Unity UI editor workspace.

For user input an Input Field element was used which was a convenient and easy way of capturing input and handling submission upon the user's request via the enter key or loss of focus. The Input Field was dropped onto the canvas and resized as needed, and the submission listener was then given my input handling function to call when triggered. This was far simpler than the *ASGE* implementation as no handling of character arrays needed to be carried out, the engine handles this all by default.

Adding sound within Unity was also straightforward. An AudioSource component was added to the game's empty script container that allowed for the ability to play the background audio with an adjustable volume and looping ability. The audio's output group is also able to be set here for further customisation if required. Sound priorities can be set to give priority to certain sounds if the engine's maximum number of concurrent sounds are surpassed. Additional settings such as panning and reverb are also available here as default, and able to be expanded upon via a number of third party plugins such as *Wwise* by AudioKinetic (2013). Keyboard sounds are scripted in C# by passing all required audio files through the previously mentioned drag-and-drop referencing system. These sounds are then triggered on a random switch statement and played through Unity's AudioSource class. This works well, however an AudioSource can only play one sound at a time, so this could get bulky for bigger projects – for the keyboard sound effects where only one is ever playing at a time however, this works acceptably.



```
You load the demo text adventure and everything is quiet.
No story...
No text...
No adventure.
Add some levels and zones to create your game!

It's an empty space. The demo has nothing in it!
```

Figure 13 The final Unity game.

Without the notion of classes or Blueprints as found in my other two implementations, it can be argued that Unity would be the hardest engine to utilise in a wider project. The implementation with scripting and UI takes up a couple of GameObjects within the Unity interface, and the 2D UI doesn't really seem to have much to expand upon.

That said, it is more than possible to expand this implementation as all scripting and audio is carried out on a single GameObject and all UI elements are contained within a Canvas which is more than possible to customise or expand upon. The Unity engine, like both others, being cross platform gives further flexibility as to how you could use the project.

6) Implementing in Unreal Engine

Initially Unreal Engine was ruled out for an implementation as it does not offer lightweight 2D solutions like Unity does, meaning that the project would have to be created in the same way that 3D games produce their menus - create a frontend blank level which the UI renders over. This seemed like an overly bulky solution for a text adventure, but soon it was realised that the solution is simple… utilise the 3D environment and have the text adventure placed within it, similar to *The House Abandon*. Not only does this justify the bulky nature of an Unreal project and build, but adds an extra reason to chose the engine for the implementation.

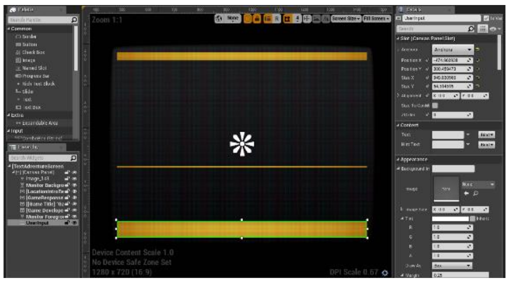


Figure 14 The UI editor interface.

To achieve this 3D solution, Unreal Engine's UI Widget component was utilised, which allows the placement of a UI element within the 3D space. First, the assets required to recreate the look of the other two implementations were imported – the background image, foreground overlay and font. Luckily as Unreal has a wide range of support for common file formats, no conversions were needed and both images as PNGs or the font as TTF, all of which imported within seconds, the same as Unity. Creating the UI itself was quite straight forward as the GUI drag-and-drop editor in Unreal is very user friendly and simple to use. Both images were added in the correct z-order, anchoring them to the centre of the canvas - this immediately gave the same look of the other two implementations. Adding text was also simple, a series of text blocks with the correct font assigned were dragged in to the canvas and resized, along with an editable textbox for the user input. Within minutes the full UI was complete. Creating the UI through this editor was by far the easiest of all three implementations – anchoring was simple which was a bonus over Unity's editor, plus being able to drag and drop text elements rather than be forced to use position sliders like Unity was another ease-of-use factor. Clearly both Unity and Unreal win in this aspect over OpenGL as the "what you see is what you get" style of the editors make it far easier than manually setting sizes and placing elements through code.

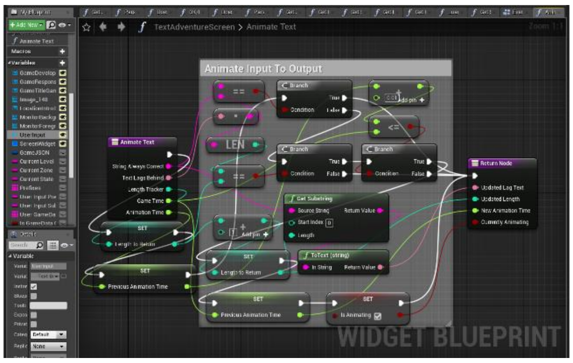


Figure 15 The Unreal Blueprint interface, with a script shown for animating the output text.

In implementing the logic of the text adventure, Unreal's Blueprint system was chosen rather than its C++ editor to separate this implementation from the others, and access Unreal's advanced Blueprint functionality, which is effectively its unique selling point. As default, Unreal doesn't have built in support for parsing JSON files which was an issue for reading the logic from the agnostic tool's JSON output. Luckily however, Unreal has a marketplace for plugins and through a search *'JSONParser'* was found, which provides the functionality required to read a JSON file and parse its logic into maps and arrays which can then be acted upon. All logic for the text adventure was created within the UI Blueprint to keep the project as easy to navigate as possible, also improving on its future reusability, since the entire text adventure is contained within this one Blueprint class. Blueprint provides a nice drag-and-drop GUI for editing scripts which makes it simple to create logic without touching any code at all. Although a nice system to work with on the whole through intuitive controls and a responsive interface, scripts get overly large quickly. Just creating a basic script that would be a small number of lines in code can soon become something you have to zoom and pan around to understand to its full extent in Blueprint. Functions and collapsible groups do help to combat this issue, but often this requires jumping in and out of different functions when debugging the Blueprint which quickly becomes a nuisance. As well as this, some Blueprint node names can be confusing which makes finding them a little tricky at times. This confusing node naming is a result of trying to abstract itself from any coding aspect to provide a scripting interface that designers can understand, which slowed progress when first starting out for this project. That this confusion is something that should not occur within the text adventure logic editor tool, as to combat this, the Brainiac plugin includes explanatory node names alongside written documentation to further explain the use of each node.

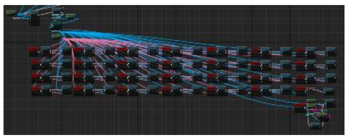


Figure 16 The final event-driven key input solution.

Aside from these UX issues, an issue with Unreal's input handling was encountered with the 3D UI which resulted in the creation of an event-driven Blueprint to handle keyboard inputs and perform their expected actions out to the textbox. Although textbox focus worked fine in the editor viewport, compiling the game seemed to break this entirely which caused the lengthy workaround. This is an issue with higher-level engine tools, as if a similar issue had occurred in the OpenGL implementation, the problem could have been easily debugged. On a plus side, the event-driven key Blueprint allowed the ability to have per-key sound effects which was easy to achieve through Unreal's 2D sound node. Unreal prefers to use WAV files for audio due to their uncompressed nature and high dynamic range capabilities, although MP3s can be imported as well if required. For this project, all files were converted to WAV to make the process simpler.

Unreal's built in audio handler works well - for this simple project only the volume controls and loop settings needed to be adjusted, however if needed a variety of mixing settings are available similar to Unity as well as conversation blueprints for dialogue. Up to 32 audio files can be played at once with Unreal's audio mixer (DarkTonic, 2014), which is impressive considering older low-level implementations support just one concurrent sound, through Windows functions such as *PlaySound* (Microsoft, 2016).

This Unreal implementation is the easiest of the three to use within a wider project. Although the OpenGL game is class-based and could easily be imported to another project, some code work would have to be carried out to call functions of the text adventure in a new project. The ease of a single Blueprint in Unreal for the entire text adventure UI and logic makes it the simplest to expand upon of the three implementations. An entire game could be created and then the UI Blueprint dropped in without any further work and the text adventure would run without issue. Having a solution this simple requiring no modification would be perfect for utilising in a full project.

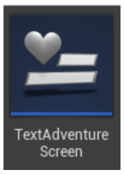


Figure 17 UI Blueprint.

7) A critical review

Overall, the final project is a success in that it meets its original aims through a robust agnostic toolset and three functioning implementations. The tool's supporting documentation provides a great helping hand to anyone coming into it for the first time and the interface is easy to use, separating totally from any programming requirement which makes it perfect for designers. Localisation support allows for extended accessibility and being able to output to three alternate implementations in 2D and 3D with no script editing needed gives great flexibility to the types of projects than can be created with the tool in a very efficient pipeline. Both Unity and Unreal implementations are created in a way that allows for easy integration to further projects with those engines, and the ASGE implementation is developed primarily in a single class that could easily be reused with little code editing in a multitude of C++ engine solutions.

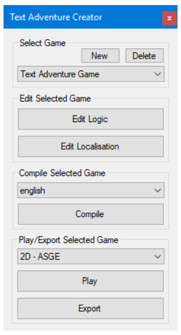
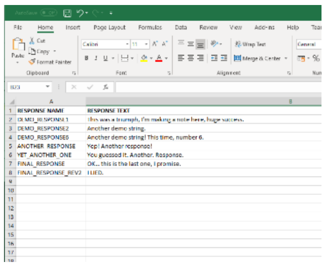


Figure 18 The final main tool window.

Successes of the implementations include the keyboard sound effects across the Unity and Unreal implementations, clean and responsive input handling in the Unity implementation, and a uniform design and play experience across all three. The loading screen in the *ASGE* implementation is a nice feature and given more time it would be nice to have it across all three, perhaps paying to remove the Unity watermark title screen – which can be seen as a disadvantage of that engine implementation in the version being used. Another success of the implementation is the support of 3D with Unreal Engine, something that is unexpected of a typical text adventure experience.

That said, there are some drawbacks and things which should have been carried out differently, if given more time. The major issue is arguably not being able to import custom assets (such as images or sounds) into the tools. The option to add a personalised logo to the loading screen or rearrange the layout of the game viewport through additional JSON parameters would be great to extend the tool's functionality. Being able to import your own soundtrack would also add an extra layer of customisation to help make each project created with the tools more unique, without much further alteration. These additions would really set the tools apart from anything else currently available. Integration with programs such as *Microsoft Excel* that designers and writers are already familiar with could also go a way to making the tools more useable for designers – for example, allow an *Excel* document to be imported into the localisation tool.



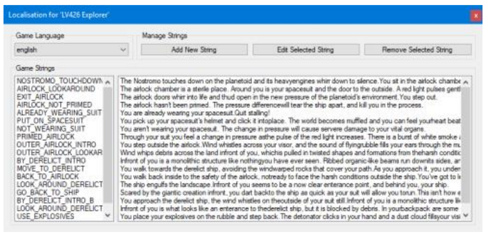
	A	B
1	RESPONSE NAME	RESPONSE TEXT
2	DEMO_RESPONSE1	This was a triumph, I'm making a note here, huge success.
3	DEMO_RESPONSE2	Another demo string.
4	DEMO_RESPONSE6	Another demo string! This time, number 6.
5	ANOTHER_RESPONSE	Yep! Another response!
6	YET_ANOTHER_ONE	You guessed it. Another. Response.
7	FINAL_RESPONSE	OK... this is the last one, I promise.
8	FINAL_RESPONSE_REV2	I LIED.
9		
10		
11		
12		
13		
14		
15		
16		
17		

Figure 19 Existing writer tools could be utilised like *Microsoft Excel* as demonstrated here.

Additionally, the option for extended logic per node branch would be a welcome feature. The ability to have AND/OR logic operators on the conditional "game data" nodes would be a great improvement, as in the current implementation they only act as AND logic which can feel restrictive when creating certain projects. Game over logic could also be improved, as currently it only ends the game. Having the option to add win/loss text on-screen would be a more complete solution to this. Win/loss logic is implemented in the *Brainiac* plugin and JSON converter, so it could potentially be added in an extended solution with very little modification.

One feature that would be interesting to experiment with would be similar to that of *The House Abandon* in being able to modify the 3D environment around the player in the Unreal implementation. This functionality was not included as the tools were intended to be agnostic across all three implementations, however allowing for additional engine-specific nodes in the logic editor could open up this functionality. For example, assigning a tag to a 3D entity in Unreal that could be referenced in a *Brainiac* node to show/hide depending on logic conditions would be a good next step to advance the functionality of the tools, allowing for some interesting projects to be created.

The tools are more than suitable for use within a wider project, and each implementation provides a good starting point for any developer to take the project and expand it further. All logic scripting in the implementations provided is consolidated as much as possible for simplicity in expansion, and all scripting is commented suitably to explain the processes it takes to parse the JSON and act on the user's input. Using the tools in a wider project is totally possible "as-is" without any alterations.



NOSTROMO_TOUCHDOWN	The Nostromo touches down on the planetoid and its heavyengines whirr down to silence. You sit in the airlock chambe
AIRLOCK_LOOKAROUND	The airlock chamber is a sterile place. Around you is your spacesuit and the door to the outside. A red light pulses gent
EXIT_AIRLOCK	The airlock doors whirr into life and thud open in the new pressure of the planetoid's environment. You step out.
AIRLOCK_NOT_PRIMED	The airlock hasn't been primed. The pressure differencewill tear the ship apart, and kill you in the process.
ALREADY_WEARING_SUIT	You are already wearing your spacesuit. Quit stalling!
PUT_ON_SPACESUIT	You pick up your spacesuit's helmet and click it intoplace. The world becomes muffled and you can feel yourheart beat
NOT_WEARING_SUIT	You aren't wearing your spacesuit. The change in pressure will cause servere damage to your vital organs.
PRIMED_AIRLOCK	Through your suit you feel a change in pressure asthe pulse of the red light increases. There is a burst of white smoke
OUTER_AIRLOCK_INTRO	You step outside the airlock. Wind whistles across your visor, and the sound of flyingrubble fills your ears through the m
OUTER_AIRLOCK_LOOKAR	Wind whips debris across the land infront of you, whichis pulled in twisted shapes and formations from theharsh conditi
BY_DERELICT_INTRO	Infront of you is a monolithic structure like nothingyou have ever seen. Ribbed organic-like beams run downits sides, ar
MOVE_TO_DERELICT	You walk towards the derelict ship, avoiding the windwarped rocks that cover your path. As you approach it, you under
BACK_TO_AIRLOCK	You walk back inside to the safety of the airlock, notready to face the harsh conditions outside the ship. You've got to
LOOK_AROUND_DERELICT	The ship engulfs the landscape. Infront of you seems to be a now clear enterance point, and behind you, your ship.
GO_BACK_TO_SHIP	Scared by the giantic creation infront, you dart backto the ship as quick as your suit will allow you torun. This isn't how
BY_DERELICT_INTRO_B	You approach the derelict ship, the wind whistles on theoutside of your suit still. Infront of you is a monolithic structure li
LOOK_AROUND_DERELICT	Infront of you is what looks like an enterance to thederelict ship, but it is blocked by debris. In yourbackpack are some
USE_EXPLOSIVES	You place your explosives on the rubble and step back. The detonator clicks in your hand and a dust cloud fillsyour visi

Figure 20 The final localisation string explorer in the toolset. Strings are created through the additional editor window.

Unity's implementation is perhaps the strongest of the three as it is lightweight, robust and responsive. Memory management is effective from the advantages provided by C#, and performance is superb. Further expansion is simple through the Unity GUI and although limited to 2D, could easily be dropped into another game. The text adventure script could always be ported into a 3D environment, with the UI recreated in that space. In second place, due mainly to its unique look is the Unreal implementation. Its bulky nature and UI artefacts in the 3D space knock it down from first place, however this project is visually very appealing, and the environment adds an extra depth to the game which the 2D implementations cannot match. The Blueprint class system is superb for further expansion, meaning that a developer could simply drop the text adventure Blueprint into a project and immediately access all of its features – including the UI, in 2D or 3D. Let alone its easy reuseability, the Blueprint system is easy to navigate even for novice developers to extend the core system's functionality without touching any code. In last place is the *ASGE* OpenGL implementation. Although the most lightweight in terms of storage space, and least "branded" of the three (Unity forcing a splash-screen and Unreal providing its badge as the app's icon), this implementation just doesn't feel as solid as the 2D implementation in Unity or as unique as the 3D implementation in Unreal. The reality is that this solution is the best on memory management due to good C++ coding practices, and further work to the engine implementation could add features seen within the Unity build, however as it stands the input field in the Unity implementation just makes the experience feel more natural and less "hacky". I also feel that the *ASGE* implementation would be the hardest of the three to expand upon, however it is still more than possible due to it being primarily coded into one class. That project would be reliant on using C++ however, and having the ability to import third party libraries.

Figure 21 The un-removable logo splash screen in the free version of Unity, shown at the start of the game.

In terms of the creation of the project, Unity again wins. The interface of Unity is simplistic and easy to navigate, even down to the coding aspect which can be carried out in *Microsoft Visual Studio*, a familiar IDE with code hinting. Scripting in the engine feels intuitive with easy to understand function names and variables, with all the advantages of C#. The interface of Unity itself allows for the useful drag-and-drop referencing as mentioned previously, which again is a useful feature for visually referencing GameObjects. The interface of Unreal is equally intuitive and easy to navigate, however arguably slightly more complicated to script within when starting out due to somewhat bizarre names to some Blueprint nodes. The Blueprint system is overall a useful tool and powerful for developers who are just starting out, however in retrospect, using C++ seems like a more appealing option. The UI editor of Unreal does top Unity's solution with an easy to use drag and drop interface, although the mysterious UI bug that was encountered throws questions about the reliability of some of its functionality. With no GUI, ASGE of course comes last for a 2D UI-driven game. That said, the interface of *CLion* is very easy to understand and it is not at all cluttered with pointless options that are rarely used. Error hinting can sometimes be erroneous, however breakpoint debugging is overly useful and helps to find the issues most of the time. Code can be enlarged for readability and smart code navigation allows for easy tweaks and fixes when revisiting a project.

In summary, this project has shown a lot from creating user friendly tools to parsing data reliably in different engine environments. Using and understanding JSON is a valuable skill to have as its lightweight data structure is great for further implementation in games for uses such as character attributes. Not having to recompile a game to modify parameters is useful for a fast-iterative development process. A similar system to this was used in the development of video game *Alien: Isolation*. *Brainiac Designer* controlled the game's behaviour trees which could be edited while the game was running live, so effects were immediately visible in the game (Bray, 2016 & 2018). A similar live system also applied to lighting and character configurations (Gratton, 2017). Further take-aways from this project are how easily logic systems can be recreated across multiple engines – showing that no matter how different an engine may seem, the real distinction between them are their toolsets.

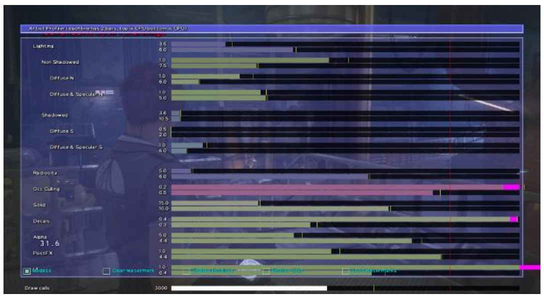


Figure 22 In-game debug options are shown within an early build of *Alien: Isolation* (Gratton, 2017).

References

- Alexander, L (2014) *The joy of text – the fall and rise of interactive fiction* (The Guardian). Available from: https://www.theguardian.com/technology/2014/oct/22/interactive-fiction-awards-games
- AudioKinetic (2013) Wwise Unity Integration (2017) [computer program]. Available from: https://www.audiokinetic.com/library/edge/?source=Unity&id=main.html
- Bray, A (2016) *It's in the Vents: The AI of Alien Isolation* [presentation at nucl.ai 2016]. Available from: https://archives.nucl.ai/recording/its-in-the-vents-the-ai-of-alien-isolation/
- Bray, A (2018) LinkedIn Messages to Matthew Filer, 28 December. Available from: https://docs.google.com/document/d/1voM6eiX6MV7rie3pVTN-WIuYfKTthaSszCyqy19Dv-A/edit?usp=sharing
- Bunny83 (2012) SimpleJSON (2017) [computer program]. Available from: http://wiki.unity3d.com/index.php/SimpleJSON#Download
- Ciraolo, M (1985) *Top Software: A List of Favourites* (II Computing). Available from: https://archive.org/stream/II_Computing_Vol_1_No_1_Oct_Nov_85_Premiere#page/n51/mode/2up
- DarkTonic (2014) *Unity 5 is great! Do I still need Master Audio? Absolutely!.* (FreeFourms). Available from: http://darktonic.freeforums.net/thread/130/unity-great-master-audio-absolutely
- Gabsdil, M (2001) Building a Text Adventure on Description Logic [online]. Saarland University. Available from: http://robnugen.com/downloads/textadventure/Gabsdil-et-al.pdf
- Gamasutra (2017) *Experimental Text Adventure 'Stories Untold' Out Today* [press release]. Available from: http://www.gamasutra.com/view/pressreleases/292728/EXPERIMENTAL_TEXT_ADVENTURE_lsquoSTORIES_UNTOLDrsquo_OUTTODAY.php
- Gratton, C (2017) *A transformative approach to game engine development* [presentation at Digital Dragons 2017]. Available from: https://www.youtube.com/watch?v=FXKEiFUXBIo
- Ho, D (2003) Notepad++ (2018) [computer program]. Available from: https://notepad-plus-plus.org/
- Huxtable, J (2017) AwesomeSauce Game Engine (2019) [computer program].
- Image Libraries (2018) OpenGL Wiki [online]. 25 October. Available from: https://www.khronos.org/opengl/wiki/Image_Libraries
- JetBrains (2016) CLion (2018) [computer program]. Available from: https://www.jetbrains.com/clion/download/previous.html
- Kalicinski, M (2006) RapidXML (2009) [computer program]. Available from: http://rapidxml.sourceforge.net/
- Kollmann, D (2008) Brainiac Designer (2009) [computer program]. Available from: https://archive.codeplex.com/?p=brainiac
- Lohmann, N (2014) JSON for Modern C++ (2018) [computer program]. Available from: https://github.com/nlohmann/json
- Ludum Dare (2016) *Ludum Dare 36 – Theme: Ancient Technology | The House Abandon*. Available from: http://ludumdare.com/compo/ludum-dare-36/?action=preview&uid=112536
- MacLeod, N (2017) *Stories Untold: The House Abandon [ep1]*. Available from: https://gamingmodreviews.wordpress.com/2017/05/12/stories-untold-ep1/
- McKellan, J (2019) E-mail to Matthew Filer, 8 January. Available from: https://docs.google.com/document/d/1tsVzs2-RHx5A38GNbxMfFgNMQcuX_FKL_5q0UXFpioU/edit#
- Meutia, A (2017) *Colossal Cave Adventure – The Most Famous Classic Text-Based Adventure Game* (Crackware). Available from: https://www.crackware.me/bs-mobile/bs-android/colossal-cave-adventure-the-most-famous-classic-text-based-adventure-game/
- Microsoft (2016) *PlaySound function* [documentation]. Available from: https://docs.microsoft.com/en-us/previous-versions/dd743680(v%3Dvs.85)
- Moriarty, Brian J. (2014) *A History of the Unity Game Engine* [online]. Worcester Polytechnic Institute. Available from: https://digitalcommons.wpi.edu/iqp-all/3207/
- My Abandonware (2011) *The Hitchhiker's Guide to the Galaxy*. Available from: https://www.myabandonware.com/game/the-hitchhikers-guide-to-the-galaxy-42
- Nurseitov, N (2009) *Comparison of JSON and XML Data Interchange Formats*. Montana State University. Available from: https://pdfs.semanticscholar.org/8432/1e662b24363e032d680901627aa1bfd6088f.pdf
- Ruskin, E (2012) *AI-driven Dynamic Dialogue through Fuzzy Pattern Matching. Empower Your Writers!* [presentation at GDC 2012]. Available from: https://www.gdcvault.com/play/1015528/AI-driven-Dynamic-Dialog-through
- Socke (2017) JSONParser (2017) [computer program]. Available from: https://www.unrealengine.com/marketplace/jsonparser
- Tanbusch, S (2015) *A Rediscovered Mainframe Game from 1974 Might be the First Text Adventure* (Killscreen). Available from: https://killscreen.com/articles/rediscovered-mainframe-game-from-1974-might-first-text-adventure/
- Webster, A (2014) *The Classics: "The Hitchhiker's Guide to the Galaxy" text adventure* (The Verge). Available from: https://www.theverge.com/2014/3/8/5484666/the-classics-the-hitchhikers-guide-to-the-galaxy-text-adventure
- Zurawel, K (2017) *Interactive fiction and the origins of the conversational interface* (TechCrunch). Available from: https://techcrunch.com/2017/03/14/interactive-fiction-and-the-origins-of-the-conversational-interface/