|  |  |
| --- | --- |
| 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 |

This report outlines the process of creating designer-focussed tools that can be used to develop modern narrative-driven text adventures. The aim of the project was to create a toolset that could output an easily parse-able resource, able to be read by a simple logic implementation in a number of modern engines.

Starting out

Back in a time where processing power was at an all-time low, what better way to take users to another place than through the medium of an interactive fiction game? Through creativity of the developer, this simple medium provides a far richer experience than reading a book through interaction and exploration of the unknown. 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.

In modern gaming culture, text adventures are seen as a dying breed. 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. Years prior the cost of an engine and extra development time required would 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. 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.

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 events of the text adventure affected the environment that the player was in. The unique interaction between the 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. Instantly it became a hit 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.

Taking a narrative driven approach

One thing that set *The House Abandon*’s mechanics apart from the rest is how story driven its approach was. Rather than a traditional text adventure that uses a grid system the player navigates through using cardinal directions picking up items, The House Abandon had effectively a 3D 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. This is a far more engaging experience than the cardinal design and there is proof that it works – the *Hitchhiker* text adventure sold over 300k copies, large for the time. The success of *The House Abandon* and its subsequent acquisition by *Devolver Digital* is also further evidence to this point.

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?

To answer these questions, I first dismantled the logic of *The House Abandon* to see how I could possibly utilise bespoke implementations over multiple engines. Playing through the game I tested every possible input for five starting areas and wrote down the responses to each. I analysed the basic player movement mechanics of the game and came up with informed guesses of how the systems were operating under the hood. This led me to answer my 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. 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 implementations – controlled by this agnostic tool.

An agnostic implementation

Initially, I had planned to implement this agnostic logic script in XML. I was aware of an open source tool for modifying behaviour trees known as *Brainiac Designer* that I had used on a previous project and figured it would be perfect as a linear node interface for creating zones and scripting logic per commands. Unfortunately, after attempting this solution I hit a brick wall; 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 ugly workarounds to parse.

That led me to JSON, effectively the modern replacement of XML that is lightweight and far easier to read – which as a bonus allows for easier debugging. 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*’ provides basic enough functionality for this project’s requirements.

To test these solutions out, I typed up a basic JSON script in *Notepad++* and wrote logic in each of these 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. I extended the JSON script 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. I was happy with the result.

This left the issue of the JSON editor GUI. Due to time restraints on the project, I didn’t want to waste effort in building a node editor from scratch. I tried a series of solutions available online such as ‘*JSONViewer*’ [1], ‘*JSONmate*’ [2], and ‘*JSON GUI*’ [3] – but none of these provided the designer-focused interface that I was after. An easy to read, neat and simple tool that is not an ugly programmer-focused design was key for the aim of my project. This tool is to be used by game designers and writers, not programmers.

“The intersection between good programmers and good writers, while non-zero, is pretty small.” – Elan Ruskin, GDC 2012

I kept coming back to *Brainiac Designer* for this solution, and eventually 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 great, 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. I wanted the experience to feel more consolidated, so I decided to create a launcher application that would handle the creation of *Brainiac Designer* projects, allow localisation of strings, and additionally allow for a nicer workflow for switching projects and converting the XML script to JSON for any implementation. Using C# I developed the launcher which handles the aforementioned tasks, and utilising the open source nature of *Brainiac Designer*, I changed it to load the active project from the launcher immediately – giving a nice unified experience across the tools.

With the working launcher, converter, and Brainiac interface; I set about extending the functionality of the game logic. 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, I initially had a node 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, I changed the parent node 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 me to update 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.

In testing my implementation, I re-created the opening of *The House Abandon* to prove that the toolset works effectively. I then eventually went on to create a short game based in the *Alien* universe.

Keeping it 2D with Unity and OpenGL

To implement the game logic in OpenGL, I used the barebones engine framework *ASGE* – created by a lecturer at UWE Bristol. This framework allows easy access to basic initialisation, update and render functions as well as event listeners for key presses and simple sprite handling.

C# unity – lists and dynamic stuff

2d implementation

Going into the third dimension with Unreal Engine

Initially I ruled out Unreal Engine 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 I soon 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.

To achieve this 3D solution, I utilised Unreal Engine’s UI Widget component, which allows you to place a UI element within the 3D space. First, I imported the assets I would need recreate the look of the other two implementations – 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. 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. I added both images in the correct z-order, anchoring them to the centre of the canvas. This immediately gave me the same look of the other two implementations. Adding text too was simple, I dragged in and resized a series of text blocks with the correct font assigned, along with an editable textbox for the user input and 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 using position sliders like Unity was another ease-of-use factor. Clearly both Unity and Unreal win in this aspect over OpenGL as the WYSIWYG style of the editors make it far easier than manually setting sizes and placing elements through code.

In implementing the logic of the text adventure, I opted to use Unreal’s blueprint system rather than its C++ editor. As default, Unreal doesn’t have built in support for parsing JSON files which was an issue for reading the logic from my tool’s JSON output. Luckily however, Unreal has a marketplace for plugins and through a search I managed to find *‘JSONParser’* which provides the functionality required to read a JSON file and parse its logic into maps and arrays which can then be acted upon. I created all logic for the text adventure 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 script 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 I then found myself often jumping in and out of different functions when debugging the Blueprint which quickly became 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 I found slowed my progress when first starting out. I wanted to make sure that this confusion didn’t happen in my text adventure logic editor tool, since it follows a similar designer-focused approach. To combat this, I went back to my Brainiac plugin and simplified node names alongside writing documentation for the tools to explain the use of each node.

Aside from these UX issues, I encountered an issue with Unreal’s input handling with the 3D UI which ended up with me having to create an event-driven Blueprint to handle keyboard inputs and perform their expected actions 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 my OpenGL implementation, I could have easily debugged the problem.

This Unreal implementation is also 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.

A critical review of the result

Overall, I am very happy with the final project. I think that the toolset is quite robust and its supporting documentation provides a great helping hand to anyone coming into it for the first time. The interface is easy to use and separates totally from any programming requirement, making it perfect for designers. Localisation support allows for extended accessibility and being able to output to three alternate implementations with 2D and 3D versions and no script editing needed gives great flexibility to the types of projects than can be created with the tool.

That said, there are some drawbacks and things I would have done differently given more time. The major issue I have is not being able to import custom graphical assets into the tools. I would like there to be the option to add your own logo to the loading screen, or rearrange the layout of the game viewport entirely through additional JSON parameters for text placement coordinates. I think this would add an extra layer of customisation to the final result that someone creates with the tools and really set it apart from anything else currently available.

Additionally, I would like the option for extended logic per node branch. I think being able 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 projects. Game over logic could also be improved, currently it only ends the game, so having the option to add win/loss text on-screen would be a nicer solution to this.

I think a major drawback to the project is one of its successes – how cut off from the programming aspect it is. Using the tools as a programmer you are quite cut off from changing game elements or extending the project. Obviously if released publicly, I could opt to go open source, but this utilises the engine-specific editors and is not a benefit of my tools themselves. Having an option to customise the 3D environment of the Unreal Engine game would be nice to combat this issue – perhaps a further tool within the launcher that allows for scripted effects in the environment, creating systems similar to that of *The House Abandon*’s late meta-game.

This project has taught me 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 things such as character attributes. Not having to recompile a game to modify parameters is great 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 [4]. A similar live system also applied to lighting and character configurations [5].

Having previously never used Unreal Engine for a full project, this was also a good introduction into the Blueprint scripting system. It allowed me to see its drawbacks, especially with node naming, and use those to benefit my own project.

COMPARE ALL 3

References

[1] <http://jsonviewer.arianv.com/>

[2] <http://jsonmate.com/>

[3] <https://json-gui.site/>

|  |  |  |
| --- | --- | --- |
| 1 |  |  |

[5] <https://www.youtube.com/watch?v=FXKEiFUXBIo> clive gratton

[4] <https://archives.nucl.ai/recording/its-in-the-vents-the-ai-of-alien-isolation/> andy bray

<https://www.theguardian.com/technology/2014/oct/22/interactive-fiction-awards-games>

<https://www.gdcvault.com/play/1015528/AI-driven-Dynamic-Dialog-through>

<http://inform7.com/>

<https://ahopeful.wordpress.com/2015/04/22/wander-1974-a-lost-mainframe-game-is-found/>

<https://ifcomp.org/>

<http://www.bbc.co.uk/programmes/articles/1g84m0sXpnNCv84GpN2PLZG/the-game-30th-anniversary-edition>