Grail to the Thief

An Interactive Audio Adventure

Interactive Media and Game Development

A Major Qualifying Project Report

submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

by Anthony Russo and DJ White

Advisors: Brian Moriarty, Keith Zizza

MQP MBJ 1300

Abstract

Grail to the Thief: An Interactive Audio Adventure is an Interactive Media and Game Development Major Qualifying Project at Worcester Polytechnic Institute. The purpose of the project was to create an immersive, engaging game playable by the visually impaired, together with tools to enable future developers to make such games. The completed project features polished dialogue and music, all originally recorded and mastered specifically for the project. A Psychological Sciences major was also brought on to the team to assure that our experience goals were being met.

Acknowledgments

We would like to graciously thank our advisors on the project, Professors Brian Moriarty and Keith Zizza, for helping us sort through the design, inspiration, audio development, and production of our game and tool. Without them, the product would not have been nearly as successful as it turned out.

We would also like to thank Greg Moore, WPI Psychological Sciences major, whose invaluable testing of the game resulted in a greatly improved project.

Contents

Table of Figures

Introduction

The idea for this project stemmed from Anthony’s IQP Project *Creating Video Games for the Visually Impaired (*coauthored by Steven Vandal and Neal Sacks). The paper was an analysis of the current games for the visually impaired that exist today, the technology that can be utilized to make games for the blind, and interviews with students at Perkins School for the Blind in Lancaster, Massachusetts, dealing with what they want in accessible games.

The paper made some key points, such as the advances in technology that could be used to make games for the visually impaired, as well as critical analysis of the mostly dated games that the blind had available to them. The paper closed with a quote the expresses the main problem with games for the visually impaired:

“While the technology exists today to make impressive games for the visually impaired, no one has taken it upon themselves to take this advanced technology and make a fun, accessible game.”

During the interviews at Perkins, students from their teens to their twenties were interviewed about what games they had played, what they enjoyed about those games, and what they would like to see in future accessible games. The group was varied and dynamic, having different opinions on what games were fun and which were not, what makes a good game, and how games should be developed. However, they all agreed on three main points about games for the visually impaired.

First, they wanted better designed and produced accessible games. The games currently available to the visually impaired were nowhere near the quality or polish of those for the sighted. They were also quite dated; most were over ten years old.

Second, they made a point of not wanting to hear any synthesized computer voices, such as screen readers, in their games. Synthesized voice is a staple in their everyday lives, being heard whenever they use a computer. They associated screen readers with monotonous tasks and schoolwork, not fun and play.

Finally, and perhaps most important, the students wanted more games. Titles built to be accessible to the blind are few and far between, leading them to adapt to playing games that do not even have the visually impaired in mind. They all enjoy video games, but do not have nearly enough to choose from and want a better, wider selection. [1]

Initial design

Inspiration

Initially, the Interactive Audio Adventure concentrated on audio, and treated its medium like a black box. It took in sound, and returned sound, requiring only a microphone and speakers. It was envisioned as working on iPhone, and would feel like having a conversation on speakerphone.

This concept was similar to early phone games such as *Hardbody Hospital,* essentially "radio dramas" where a player could make a handful of choices over the course of an episode. Our project aimed to keep that spirit, while opening up the experience. Instead of a branching script, the game was to act more like a parser-driven text adventure. It would feature a conversational AI agent, driven by commands that would be executed in the game world.

Key Experience

Our key experience goal was to completely immerse the player into the game world through audio. We would tell a coherent and engaging story, using only dialogue, sound effects, ambience and music.

This experience was designed to be completely accessible to the blind. By accessible, we mean that a person playing the game without any visuals can always be fully aware what is happening in the story, while maintaining a sense of personal control.

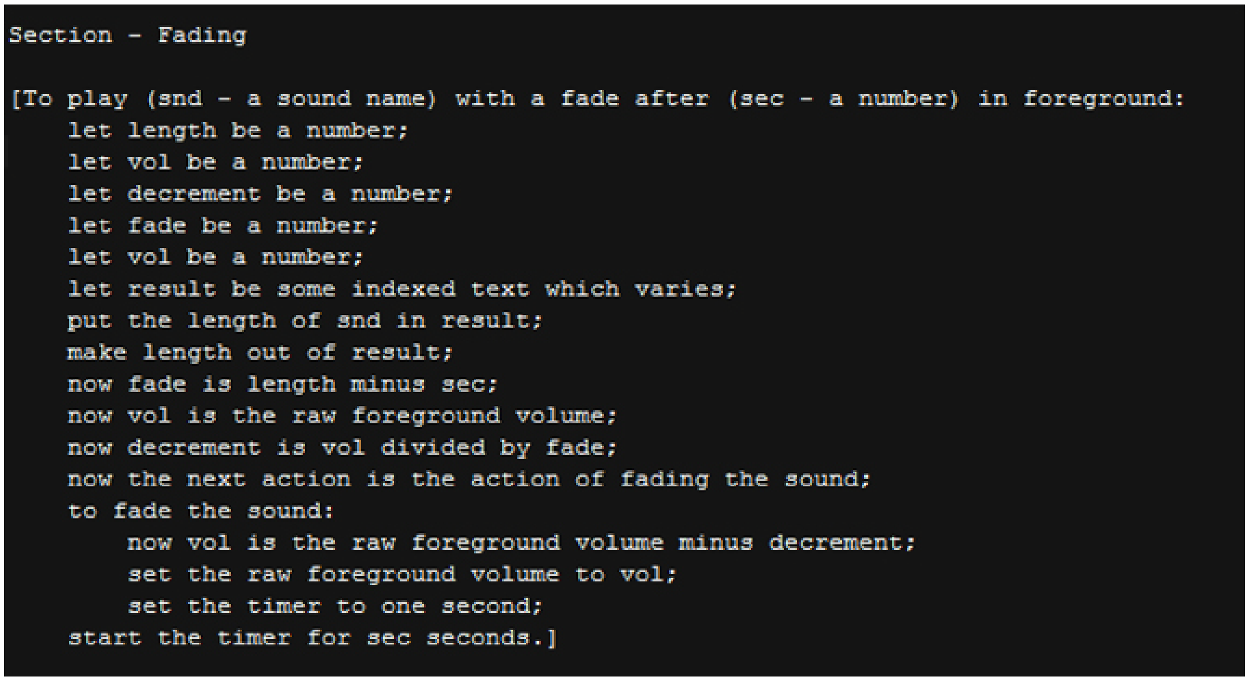
Initial Ideas

As the project came together, it was decided that iPhone was not a viable platform. The game required a framework in which to build the game first, and that limited the engines to Frotz, an open-source interactive fiction interpreter for iOS. However, we also required speech recognition. The only candidate technology was Dragon SDK from Nuance, which would have to be interfaced with Frotz. Unfortunately, there didn’t seem to be a simple way to play sounds on iOS from the interactive fiction story file.

Inform

The problem was solved by dropping the platform restriction. If the interactive fiction template remained, but stayed on Windows where it originated, it would be much easier to add audio input and playback. We settled on the Inform 7 interactive fiction engine because it would allow us to create an easy-to-distribute audio extension that others could use.

Creating the extension for Inform meant working in its unique language, IF7. This proved to be unexpectedly difficult. Many basic functions needed by our extension were only available in other extensions, and these were often not fully functional. It would also require users to install all of the required extensions along with ours.



```
Section - Fading

[To play (snd - a sound name) with a fade after (sec - a number) in foreground:
    let length be a number;
    let vol be a number;
    let decrement be a number;
    let fade be a number;
    let vol be a number;
    let result be some indexed text which varies;
    put the length of snd in result;
    make length out of result;
    now fade is length minus sec;
    now vol is the raw foreground volume;
    now decrement is vol divided by fade;
    now the next action is the action of fading the sound;
    to fade the sound:
        now vol is the raw foreground volume minus decrement;
        set the raw foreground volume to vol;
        set the timer to one second;
    start the timer for sec seconds.]
```

Figure 1: Sample Inform extension code

Timers, necessary for delaying and queuing sounds, were a particular hurdle. The available timers could only be set once, and could perform only a single action when it reached zero. Thus, setting multiple sounds to play with a delay required very detailed fine-tuning of timers, some that would have to start at the conclusion of others or wait for the length of a sound. Functions like playing a list of sounds or stopping upcoming sounds would likely not have been feasible at all.

Another major issue was the ability to play multiple sounds at the same time. The number of available tracks was limited to about a dozen. While not unworkable, this was far from ideal.

Writing the actual game code for Inform was a hassle. The program has its own scripting language, supposedly based on natural English syntax. It seemed quite foreign to us. We often encountered errors for not writing things in the proper “English” syntax that most of the time, because what we were doing was so complex, did not resemble English at all. To its credit, the language does handle story-related items, settings, variables, and characters very well, and this was useful in outlining the game. Characters were able to freely move throughout the world, revisiting areas that they had already seen. However, this free movement allowed players to backtrack and hear audio that they had probably already heard. Such repetition would break the immersive environment that we were trying to create.

Inform posed a particular problem with dialogue. It was extremely difficult to script conversations between the player and a non-player character, because Inform gives the player the ability to say anything at any time. This led to a lot of “I don’t understand what you said” responses. While it was possible to get through these conversations, it would require a lot of trial and error from the player.

The blind accessibility in this initial design seemed solid. Virtually all visually impaired people have learned the locations of keys on a keyboard through haptic feedback on the keys, Braille or some other touch-based indicator. Impaired players could readily input all commands by typing. We also considered the ambitious idea, time allowing, of making the game have input through voice commands, either through Google Voice or some other speech recognition software.

Writing

Brainstorming

Brainstorming began almost immediately for the plot of the game. The group was familiar with graphic adventure game design and tropes, all being fans of games such as *Maniac Mansion II: Day of the Tentacle*, *Grim Fandango*, and Telltale's *Walking Dead*. We also took it upon ourselves to play our way through a few classic text adventures, such as *Zork* and *The Hitchhiker's Guide to the Galaxy*.

When brainstorming, we all agreed that a short, funny game would be the best idea for the project. Comedy has a history of lending itself to adventure games, appearing in many of the above titles. It was also very familiar to the group, and practical to write in the small amount of time we had to build the game (keeping in mind that for every line of dialogue we wrote, we needed to later record, master, and implement).

DJ is a fan of time travel fiction, which quickly led to the idea of a thief traveling through time, stealing things. We tied this into previous real-world knowledge by having the thief steal precious artifacts that have disappeared throughout history because this character traveled back in time and stole it. For our game, we settled on the Holy Grail.

Characters

Characters would be the most difficult part of our story. Because the game was completely audio based, most of the writing would be in characters interacting in conversations. Not only did this mean we had to write a lot of dialogue, but we had to create strong, interesting characters that were entertaining and believable.

We decided to make our main character, time-traveling thief Hank Krang, rather crass and unintelligent. This would give the player the unusual experience of controlling a personality they are probably not used to expressing: immature and extremely rude, with little common sense. We hoped this would lessen the anxiety of making immoral decisions, such as lying and cheating. For character reference, we looked to 1980s film villains, such as Johnny Lawrence from *The Karate Kid* and Biff Tannen from the *Back to the Future* franchise.

Hank's sidekick, TEDI (short for Time Excursion Digital Interface), is a hyper-intelligent supercomputer time machine. We envisioned TEDI as a watch and earpiece worn by Hank, so that only he can hear him. TEDI has no way to escape from Hank or resist his will, and is not happy about accompanying him through time. He is an excellent foil for Hank, being both intelligent and highly moral, qualities Hank obviously lacks. The back and forth, love-hate relationship between TEDI and Hank is the basis for much of the comedy. The inspiration for TEDI comes from several characters in other fiction, including Marvin from *The Hitchhikers Guide to the Galaxy*, KIT from the television series *Knight Rider*, and especially the personal computer from the cartoon *Courage the Cowardly Dog.*

We also had to create several secondary characters for Hank and TEDI to encounter. Unfortunately, we only had access to a small number of voice actors, so we kept this in mind when writing their dialogue.

Setting

We decided to set the game in medieval England, giving the game a fantasy feeling. This would place the characters in a environment that the entire group was familiar with, so simulating it in sound would be less difficult. Anthony had sound libraries for fantasy environments that he built for previous IMGD classes, making production easier.

Plot and Puzzle Design

The plot of the game revolves around Hank trying to steal the Holy Grail from Camelot. However, in order for the King to even allow him to view it, he must prove himself worthy through valiant actions, such as saving a princess, rescuing a child, and slaying a monster. Unfortunately, there have been very few problems in Camelot, so Hank must create his own problems through lying and trickery. In the end, Hank steals the Grail and returns home to sell it.

We named the game *Episode 1: Grail to the Thief* to suggest that the story had room for expansion. If the game proved popular, we could send Hank to more adventures in other time periods.

Because the game was planned as a text adventure, we wrote the game in a branching narrative, meaning that players were given choices that they could make that affected their story when comparing to other players experiences. We also wanted the player to be able to complete the three tasks in any order, which required us to take into account freedom of movement while writing.

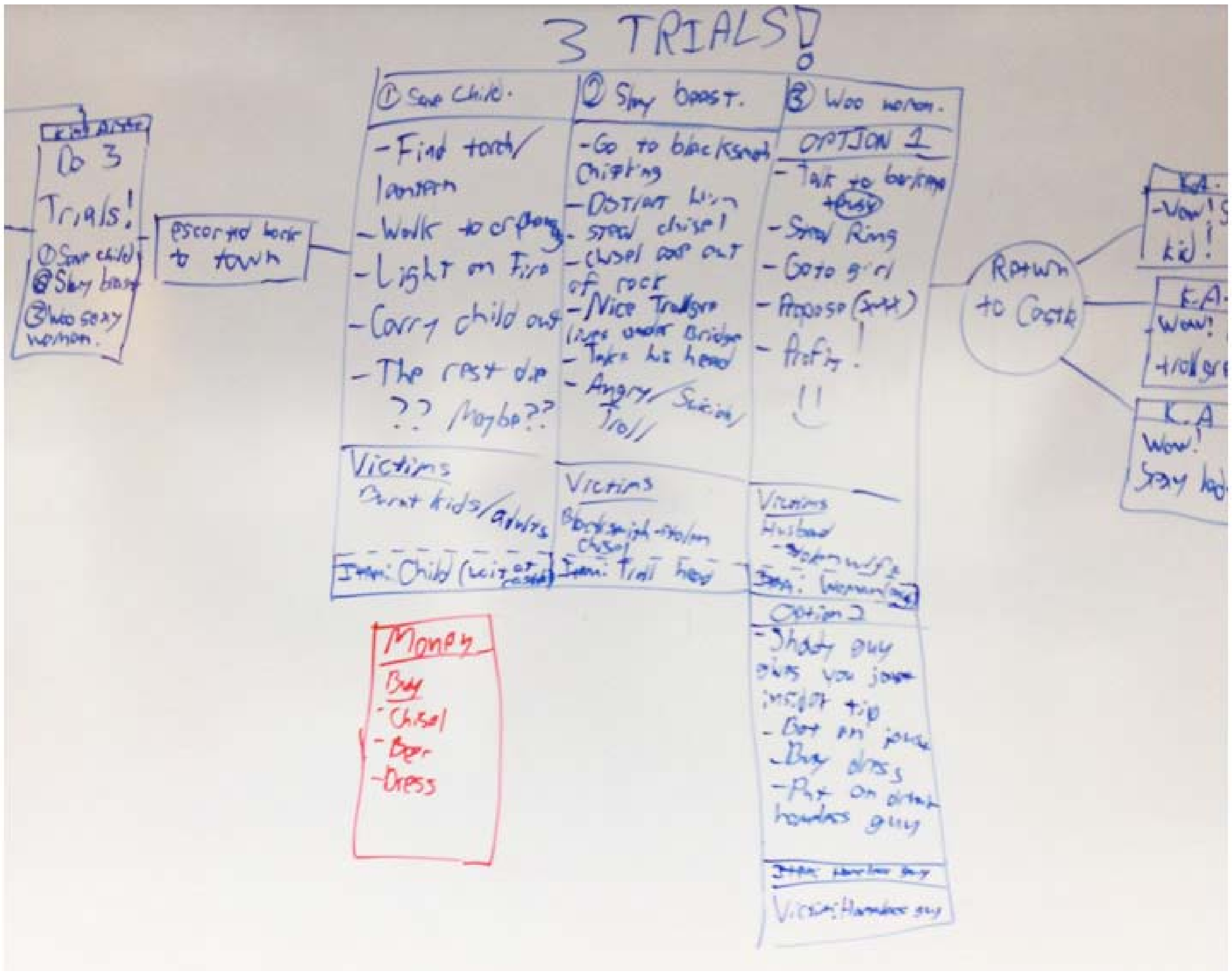


Figure 2: Brainstorming ideas on a whiteboard

To write the story, we first outlined everything on a whiteboard in section based nodes. This gave us the ability to sit back and look at how our game flowed, in what order tasks could be completed, see how many assets we had as far as characters and environments, and determine if the project was scoped well. After writing it all down, we took pictures of our outline, and cleaned it up in a mind mapping software called *XMind*, which gave us insight into the length and tone of dialogue and events.

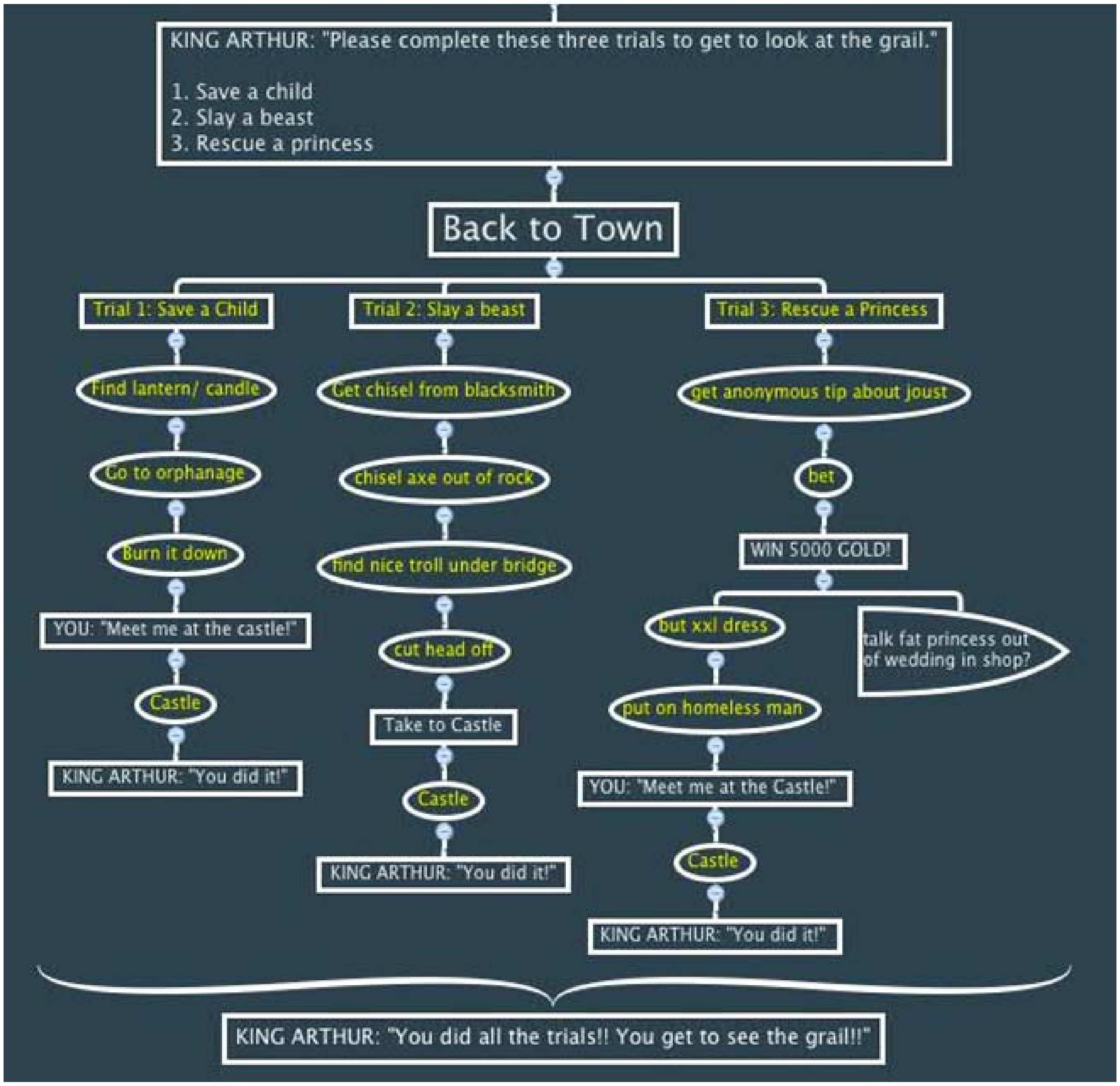
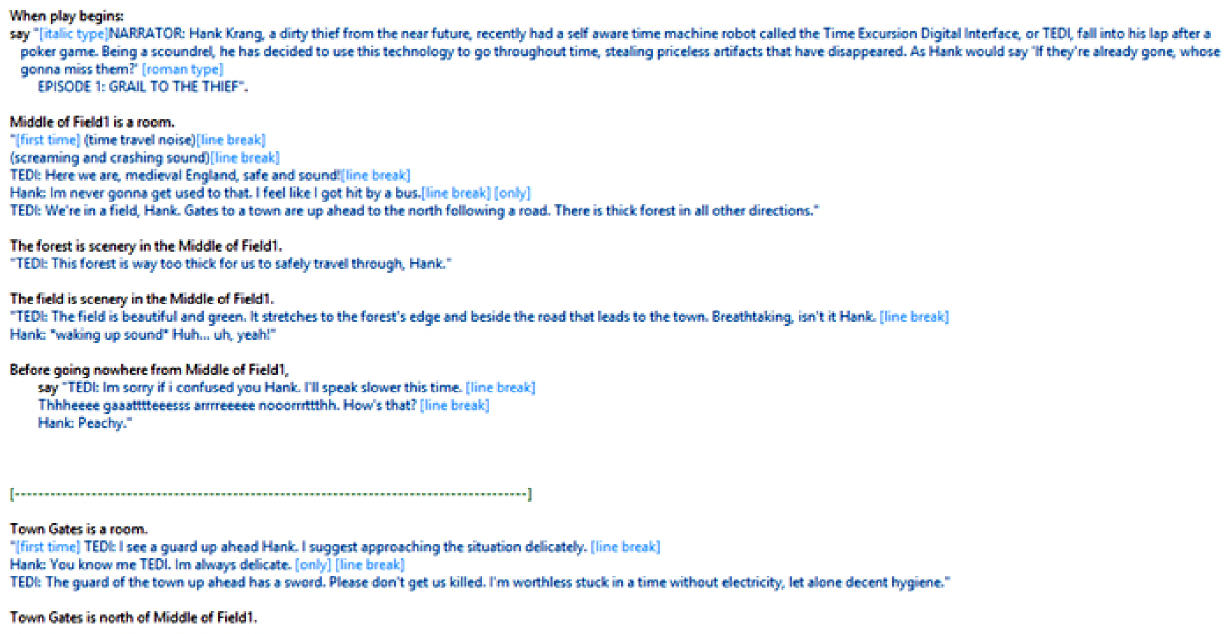

KING ARTHUR: "Please complete these three trials to get to look at the grail."
1. Save a child
2. Slay a beast
3. Rescue a princess
Back to Town
Trial 1: Save a Child
Find lantern/ candle
Go to orphanage
Burn it down
YOU: "Meet me at the castle!"
Castle
KING ARTHUR: "You did it!"
Trial 2: Slay a beast
Get chisel from blacksmith
chisel axe out of rock
find nice troll under bridge
cut head off
Take to Castle
Castle
KING ARTHUR: "You did it!"
Trial 3: Rescue a Princess
get anonymous tip about joust
bet
WIN 5000 GOLD!
but xxl dress
talk fat princess out of wedding in shop?
put on homeless man
YOU: "Meet me at the Castle!"
Castle
KING ARTHUR: "You did it!"
KING ARTHUR: "You did all the trials!! You get to see the grail!!"

Figure 3: Sample XMind outline

After *XMind*, Anthony began implementing the story directly in the Inform engine. While writing, he took into account how the player would react to certain environments, coming up with as many responses as he could for all of the inputs he thought a player would be likely to try in a given situation. Dialogue was written prefaced by the speaker's name, and sound effects and ambient sounds were surrounded by parenthesis. This not only established environments for nonlisteners, but also served as a reminder for which sounds would be required later. He also edited all predetermined states to be prefaced with the robotic character's name, TEDI, so that generic responses were spoken to the player by the computer, and not out of thin air by an impersonal narrator.



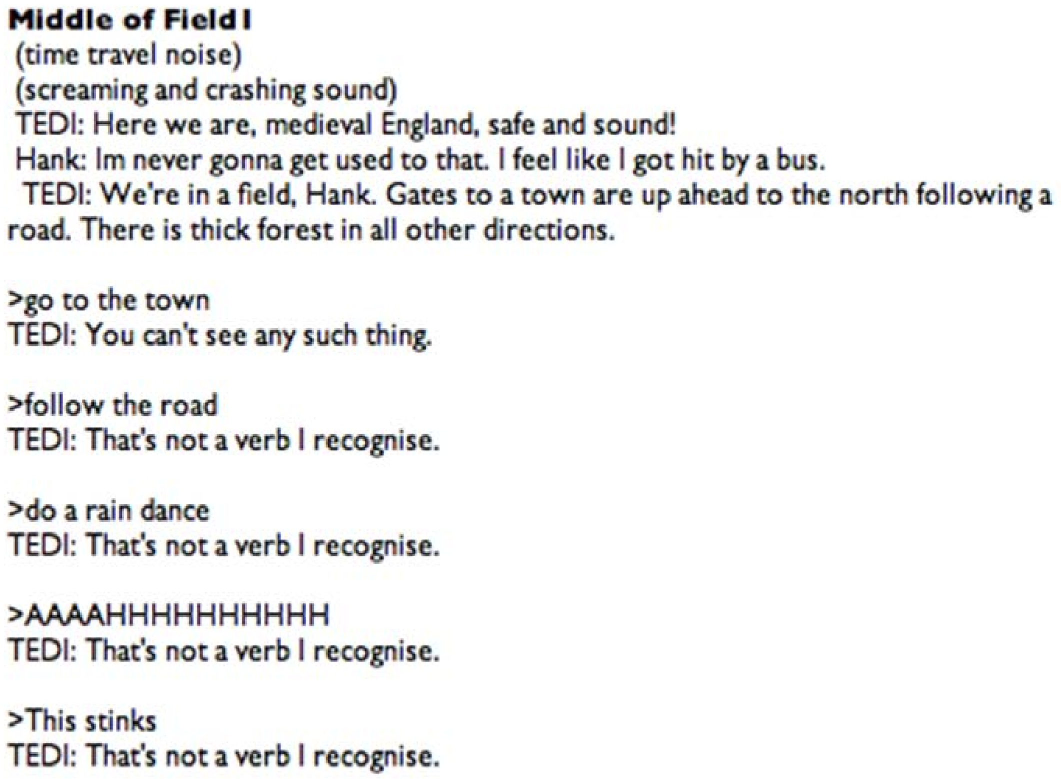
```
When play begins:
say "[italic type]NARRATOR: Hank Krang, a dirty thief from the near future, recently had a self aware time machine robot called the Time Excursion Digital Interface, or TEDI, fall into his lap after a
  poker game. Being a scoundrel, he has decided to use this technology to go throughout time, stealing priceless artifacts that have disappeared. As Hank would say 'If they're already gone, whose
  gonna miss them?' [roman type]
     EPISODE 1: GRAIL TO THE THIEF".

Middle of Field1 is a room.
"[first time] (time travel noise)[line break]
(screaming and crashing sound)[line break]
TEDI: Here we are, medieval England, safe and sound![line break]
Hank: Im never gonna get used to that. I feel like I got hit by a bus.[line break] [only]
TEDI: We're in a field, Hank. Gates to a town are up ahead to the north following a road. There is thick forest in all other directions."

The forest is scenery in the Middle of Field1.
"TEDI: This forest is way too thick for us to safely travel through, Hank."

The field is scenery in the Middle of Field1.
"TEDI: The field is beautiful and green. It stretches to the forest's edge and beside the road that leads to the town. Breathtaking, isn't it Hank. [line break]
Hank: "waking up sound" Huh... uh, yeah!"

Before going nowhere from Middle of Field1,
     say "TEDI: Im sorry if i confused you Hank. I'll speak slower this time. [line break]
     Thhheeee gaaattttteeesss arrrreeeee nooorrrttthh. How's that? [line break]
     Hank: Peachy."


[--------------------------------------------------------------------------------------]

Town Gates is a room.
"[first time] TEDI: I see a guard up ahead Hank. I suggest approaching the situation delicately. [line break]
Hank: You know me TEDI. Im always delicate. [only] [line break]
TEDI: The guard of the town up ahead has a sword. Please don't get us killed. I'm worthless stuck in a time without electricity, let alone decent hygiene."

Town Gates is north of Middle of Field1.
```

Figure 4: Programming in Inform

Initial Testing (Inform)

Greg began testing the game as soon as we had a working demo in Inform without any sounds, to see if players would enjoy the narrative and the dialogue. This testing proved essential to our project, because it showed us a number of important mistakes we were making.

Greg reported high scores in things such as frustration, how confused the testers were, and how difficult the game was, and extremely low scores in enjoyment, humor and engagement. Through analysis of the play through scripts, which Greg collected from every player, we were able to conclude as a group that players were not enjoying the game because they could type in literally anything as a command, leaving the affordances too ambiguous.



```
Middle of Field I
 (time travel noise)
 (screaming and crashing sound)
 TEDI: Here we are, medieval England, safe and sound!
 Hank: Im never gonna get used to that. I feel like I got hit by a bus.
  TEDI: We're in a field, Hank. Gates to a town are up ahead to the north following a
road. There is thick forest in all other directions.

>go to the town
TEDI: You can't see any such thing.

>follow the road
TEDI: That's not a verb I recognise.

>do a rain dance
TEDI: That's not a verb I recognise.

>AAAAHHHHHHHHHH
TEDI: That's not a verb I recognise.

>This stinks
TEDI: That's not a verb I recognise.
```

Figure 5: An example of what the collected data from playing the game in Inform looked like.

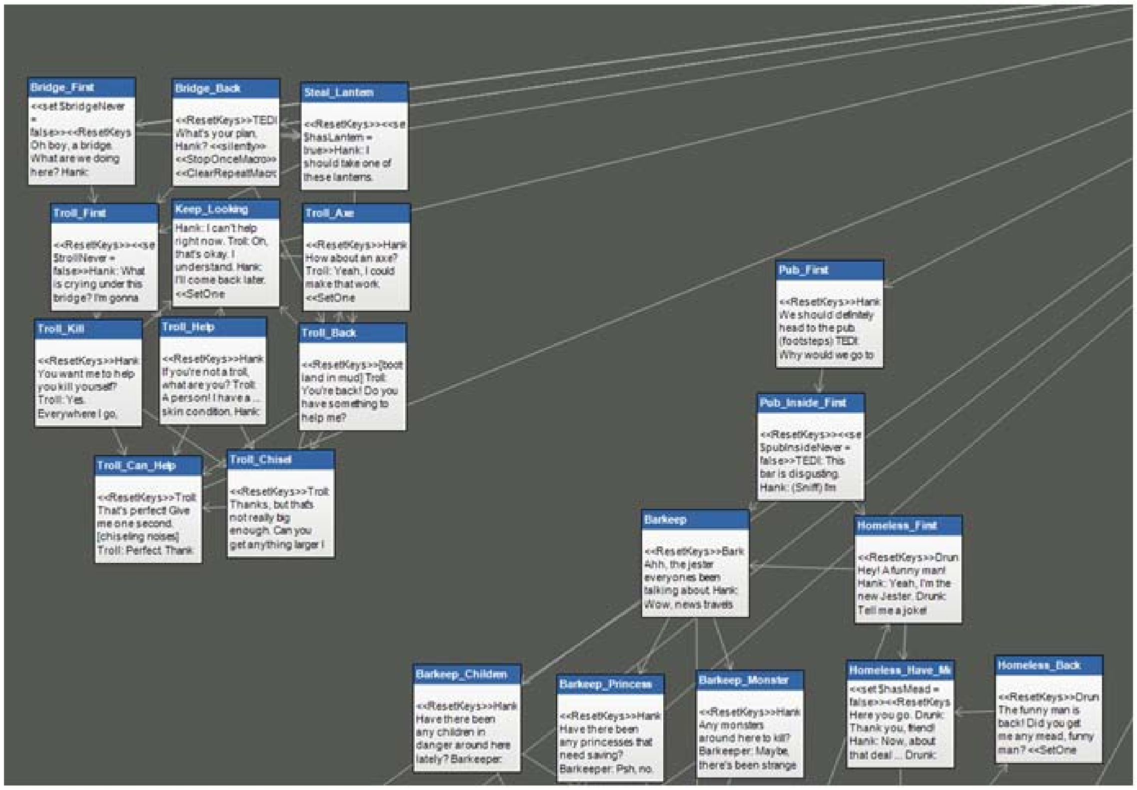
There is a certain skill set that goes along with playing a text adventure, such as knowing which words and syntaxes are likely to successfully communicate the action you want to perform. For example, players who typed commands such as “Have Hank walk to the town” were being too verbose for the Inform parser to understand what they meant. An experienced player would know that the simple imperative “Go to town” is sufficient. However, our target audience is not generally familiar with text adventure conventions, so this form of gameplay would not work for the project we were trying to make. It became obvious that drastic changes needed to be made to the design, and quickly (it was already the end of B-Term when this came up).

Switching to Twine and Web Audio

Decision to Switch

After deciding that our basic design had to change, we started looking for options. One possibility was the hyperlink-based adventure game engine Twine. At the time, Twine was rapidly growing in popularity, and games were popping up everywhere. Twine runs in a browser, allowing players to run it without having to download anything.

Branching narratives written in Twine are node-based, with only a few choices available at each decision point. We determined that adapting our story options to a few key choices would be fairly simple.



Bridge_First
<<set $bridgeNever = false>><<ResetKeys Oh boy, a bridge. What are we doing here? Hank:

Bridge_Back
<<ResetKeys>>TEDI What's your plan, Hank? <<silently>> <<StopOnceMacro>> <<ClearRepeatMacr

Steal_Lantern
<<ResetKeys>><<se $hasLantern = true>>Hank: I should take one of these lanterns.

Troll_First
<<ResetKeys>><<se $trollNever = false>>Hank: What is crying under this bridge? I'm gonna

Keep_Looking
Hank: I can't help right now. Troll: Oh, that's okay. I understand. Hank: I'll come back later. <<SetOne

Troll_Axe
<<ResetKeys>>Hank How about an axe? Troll: Yeah, I could make that work. <<SetOne

Troll_Kill
<<ResetKeys>>Hank You want me to help you kill yourself? Troll: Yes. Everywhere I go.

Troll_Help
<<ResetKeys>>Hank If you're not a troll, what are you? Troll: A person! I have a ... skin condition. Hank:

Troll_Back
<<ResetKeys>>[boot land in mud] Troll: You're back! Do you have something to help me?

Troll_Can_Help
<<ResetKeys>>Troll That's perfect! Give me one second. [chiseling noises] Troll: Perfect. Thank

Troll_Chisel
<<ResetKeys>>Troll Thanks, but that's not really big enough. Can you get anything larger I

Pub_First
<<ResetKeys>>Hank We should definitely head to the pub. (footsteps) TEDI: Why would we go to

Pub_Inside_First
<<ResetKeys>><<se $pubInsideNever = false>>TEDI: This bar is disgusting. Hank: (Sniff) I'm

Barkeep
<<ResetKeys>>Bark Ahh, the jester everyones been talking about. Hank: Wow, news travels

Homeless_First
<<ResetKeys>>Drun Hey! A funny man! Hank: Yeah, I'm the new Jester. Drunk: Tell me a joke!

Barkeep_Children
<<ResetKeys>>Hank Have there been any children in danger around here lately? Barkeeper:

Barkeep_Princess
<<ResetKeys>>Hank Have there been any princesses that need saving? Barkeeper: Psh, no.

Barkeep_Monster
<<ResetKeys>>Hank Any monsters around here to kill? Barkeeper: Maybe, there's been strange

Homeless_Have_M
<<set $hasMead = false>><<ResetKeys Here you go. Drunk: Thank you, friend! Hank: Now, about that deal ... Drunk:

Homeless_Back
<<ResetKeys>>Drun The funny man is back! Did you get me any mead, funny man? <<SetOne

Figure 6: A section of the game in the Twine layout.

As for audio, the Web Audio API, a JavaScript-based audio API for browsers, seemed like the best choice. Since Twine is also JavaScript-based, the two engines could be readily merged, resulting in an easy-to-use system to control sounds through the use of Twine macros.

The switch to Twine also greatly simplified the problem of producing audio assets. With Inform, we would need to create literally thousands of responses to various actions that might be tried by the player, whether or not they had any relevance to the game. With Twine, we would only have to produce the audio required by a limited set of choices and responses.

Accessibility and Simplicity

We brainstormed various ways for a sight-impaired player to input their choices. We considering using Google Speech, Google's speech recognition software, but the primitive state of the technology discouraged us. Then we realized that, because all available choices were explicitly listed after each response, they could be sequentially numbered, and the numbers read aloud by the narrator, allowing sight-impaired players to specify a choice by touching a single key.

We also decided that we had to give the player the ability to listen to a scene more than once, in case they missed a crucial line of dialogue. This was implemented by using the space bar to repeat the most recently played passage.

The driving factors in the decision to switch to Twine were that it made the game more accessible and easier to build. The time lost due to changing engines was minimal compared to the time gained by abandoning the challenges of Inform. There was little chance that we could have achieved the depth in Inform that we could find in Twine.

Tech

With the Web Audio API, the functions that Inform had made difficult, or not available at all, became almost trivial. The change to HTML and JavaScript meant that we also gained a host of new capabilities. The most important of these was timers, which were necessary for a variety of functions.

Timers, as many as needed, could be set to repeat or fire once, and they could all be saved to be edited later. A queue of sounds could be played by creating a series of timers, each one adding to the length of the previous. The saving of timers also allowed us to deal with the possibility of players rushing the screen and moving on before a scene was finished. By simply telling each scene to first wipe any playing or queued sound, we were able to solve a problem that likely could not have been fixed in the Inform version.

Here is a list of all audio capabilities added to Twine with our extension:

- Playing sounds with the ability to
    - Set the volume
    - Loop
    - Start after a set delay in milliseconds
- Play a sound and fade it out to a variable volume at a variable time
- Crossfade between two sounds with the same parameters

- Play a queue of sounds in order with a delay
- Play a list of sounds in random order with a delay
- Create a collection of any of the above actions, and associate them with a button press

In addition, any number of sounds may be played concurrently (there are memory issues with large numbers, but they become a cacophony well before that), and sounds may be tagged as ambient or single-time sounds. Single-time sounds, whether they are playing or not, can be wiped out with the previously mentioned stop function which would not have been possible in Inform.

With all of that information, it becomes fairly clear that much more could be done. The option present in simply playing a sound could be applied to any other function, or crossfading could be applied to a queue. Again, this is a return of the issue of readability. At some point macros could, even optionally, use too many parameters to be inviting to novice developers. By keeping things simpler, these developers are not scared off by the number of options they have, and those that notice the capabilities and have a desire to mix and match functions are free to do so. They might even look into Web Audio itself, and see that it still holds potential. There are a number of other functions it provides that are unnecessary for most applications, but would be relatively simple to implement, even to pass from Twine.

Post-Switch: On the Right Track

Story

Now that we were on the right track as far as initial game design, we still had to adapt our narrative from Inform to Twine. Anthony continued writing the game with freedom of movement in mind. DJ realized that the node-based Twine engine largely eliminated the need to backtrack to areas already visited, and that players could always be nudged to complete puzzles in those areas. This meant that the game now needed no repeating audio, which was the number one immersion-breaker we were trying to avoid. This redesign also led to the player being unable to make useless moves, streamlining the narrative in a way that communicated our story in a much more natural way, giving it the flow of a film, television program, or novel. This is something that has been lacking in the "open narrative" games that we have seen in recent years.

Audio

Though recording had been happening for some time before the switch to Twine, the workload of audio production facing us became extremely apparent as the game started to take shape through sound.

As far as recording dialogue, we were lucky to have both the equipment and the actors available to us for free. Two WPI students, Sasha Abdurazak and Peter Lepper, were extremely generous in the amount of time they were willing to commit to the project. Several hundred lines of dialogue were recorded, all in the audio suite on campus. We captured vocal performances using the Zoom H2 microphone, attached atop a microphone stand, and using several soundproofing items like wall mounted foams and stand shields. Also, we invested in a pop filter to insure the highest quality audio, avoiding plosives and distortion wherever possible.

Figure 7: Peter Lepper recording in the sound lab

While recording, we really let the actors make the characters their own, developing interesting, fun personalities. Anthony directed the recording sessions, coaching the actors through the tone and volume of their dialogue, allowing several takes and letting them improvise whatever they wanted, within reason.

As far as organizing sounds, Anthony created Excel sheets that outlined the take number, which character was reading the line, what line in what scene it was, the quality of the take, and whether it was a candidate to go in the final game. All sounds were organized in folders that corresponded to their particular Twine node, and split up to later be put together in Audacity audio program, after being exported through compressors, equalizers, pitch shifters, and other filters in Apple Logic Pro 9, a professional audio program.

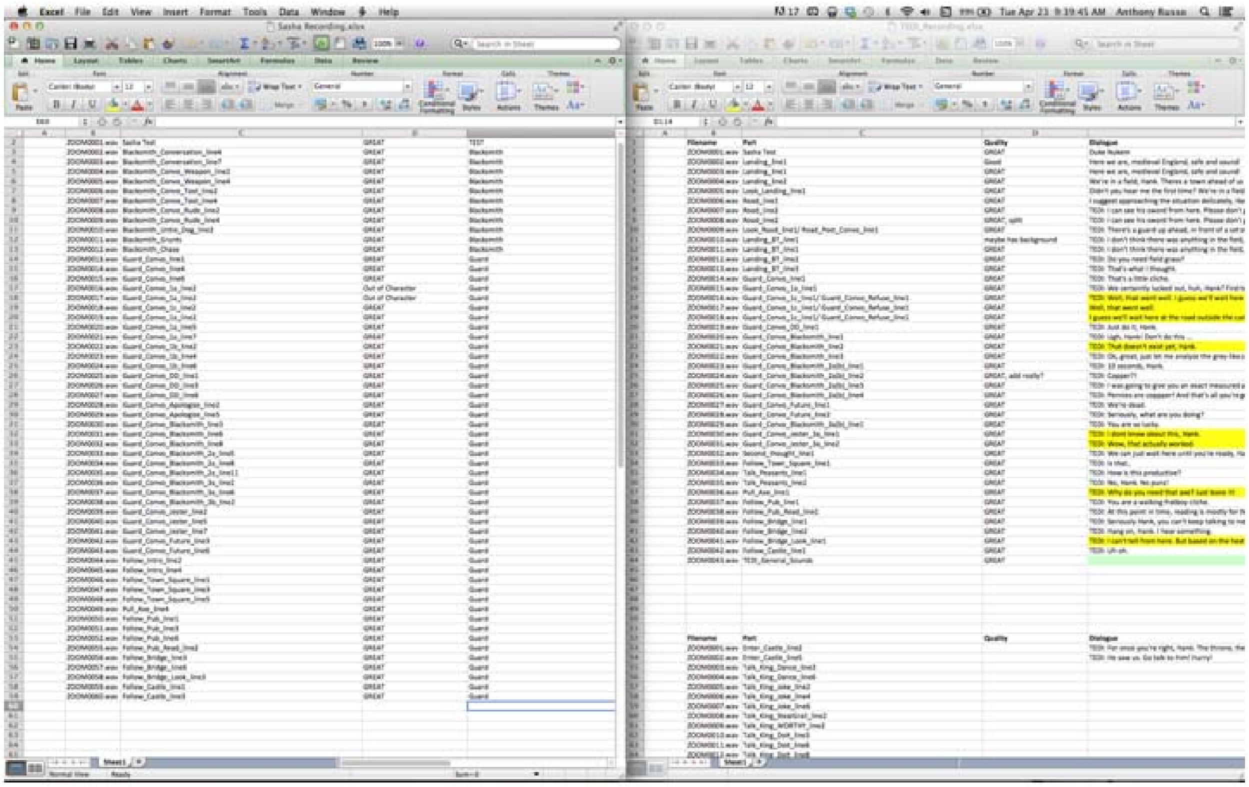


Figure 8: Organization of Sounds

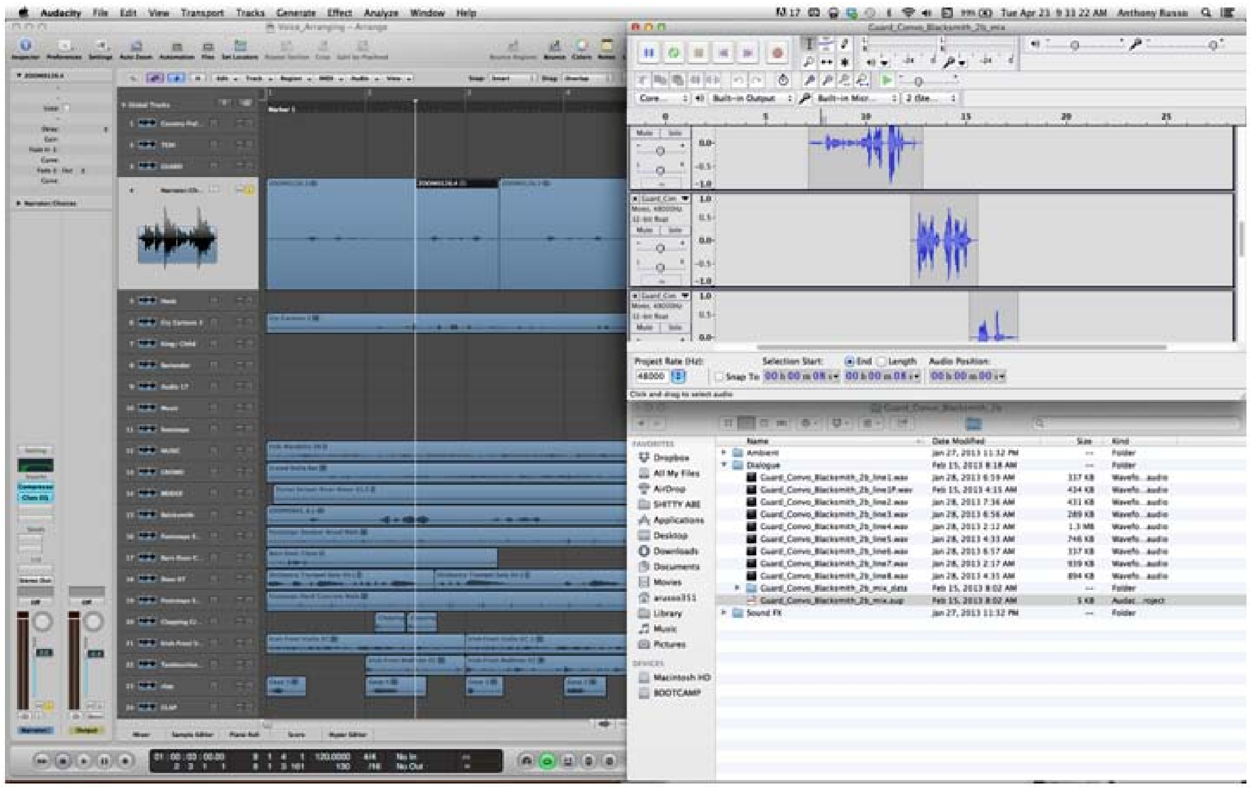

Audacity File Edit View Transport Tracks Generate Effect Analyze Window Help

Figure 9: Common desktop layout for sound editing

Anthony had several libraries of sound effects left over from previous audio classes, which proved to be invaluable to the sound design of the project. Foley recording (the capture of practical sound effects, such as footsteps) were recorded in the audio suite with the Zoom H2. Music was created in Logic, using both sound loops and virtual instruments, resulting in a completely original score for the game.

Audio was prepared in two sections: a pre-mixed or “baked” version of what would play whenever the player entered a Twine node, and an ambient track that could run through multiple nodes. While the baked single-play tracks were relatively straightforward to make, ambient tracks had to be smoothly looped to avoid distracting the player from the story experience.

To minimize loading times, all audio was exported in the .ogg file format, shrinking the original .wav files to a tenth of their original size without noticeable quality loss.

Tech

In the end, one of the greatest technical boons to the project was the Web Audio API. With Inform, sounds had to be played using system calls made in IF7, which meant using rudimentary timers and cumbersome library extensions so that more than a single sound could be played at once. While we eventually got it all working, it wasn't nearly as flexible or extensible as Web Audio proved to be.

With Web Audio, we were free to use any resource the browser could. The basic functions that Inform required extensions to even attempt were solved by using few JavaScript functions. We were also able to tackle issues that we never even attempted to solve in Inform. Twine cut down on our development time and left our tool room to grow.

Usability was an important consideration when creating new functions for the Twine audio engine. Since Twine is intended to be used by very inexperienced developers, we intentionally kept the functions and their options to a minimum.

The reason that the Web Audio engine was designed separate from the Twine extension is to ensure its readability. By simply adding to Twine's built-in macros, we make sure that the extension only slightly increases the difficulty level of using it versus Twine alone. Since the source of the engine is kept separate, these developers can look at the options given to them by the extension without worrying about the details of how the actions are performed. At the same time, any experienced developer that wishes to add functionality to the engine will find it easy to do so.

Despite the separation, the engine is not intended for use beyond Twine. This is due to two issues related to the design of Twine. First, all parameters sent from Twine to the Web Audio engine are passed as strings. This means that for any non-utility function called from Twine, all parameters must first be checked for data type and converted as necessary.

The second limitation in Twine is the number of parameters that can be passed. JavaScript is capable of handling variable numbers of parameters, and therefore could conceivably create a list of infinite sounds to play in a queue. Twine, on the other hand, must manually pass each parameter from the macro call to the engine. For this reason, the limit on queuing is fifty sounds, more than enough for most applications. This sacrifices some of the readability of the engine, but we were not as concerned with readability there.

These issues, when compared with the roadblocks faced in Inform, were relatively trivial. Twine led to the desire to make the tool useable for users with a wide variety of skill levels, but it also ensured that this was simple, without preventing ambitious users from adding to the engine’s functionality. In Inform, adding to the engine would have required skills even the most advanced users might not possess.

Release as a Tool

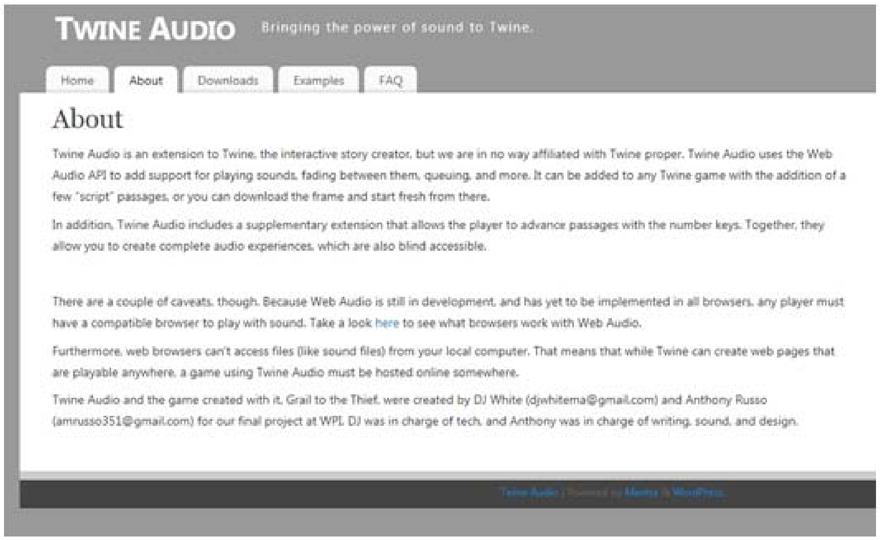
When releasing to the public, there were a few additional factors beyond the basic guidelines for the Interactive Audio Adventure. The first was ensuring accessibility to the visually impaired. Twine's use of hyperlinks means that the player must click to advance. This is technically accessible, since it's safe to assume that the player can use a screen reader if necessary. However, this is far from ideal, as mentioned in *Creating Video Games for the Visually Impaired.* Because of this, we created a set of functions to bind links to number key presses, allowing a visually impaired player to keep their hands in one place on the keyboard.

However, we recognized that this approach was not a universal desire. Many Twine developers who may want to create a story with sound may not be worried about accessibility. For instance, most Twine stories leave links in the middle of pages, rather than a list at the end like a choose-your-own-ending book. This might not work well with a numbering system. As a result, we kept this aspect of the extension separate, effectively creating two extensions. The releases are packaged both with and without the key-press extension.

Another important factor in releasing the tool was applying it to a wide range of skill levels. The main release is designed to work much like Twine itself, with a collection of added abilities. The actual audio controls are handled elsewhere, by the audio engine. For many developers, this should be enough. For others, though, we provided a direct link to the engine's source. We also made it clear where the engine is included in the extension, and how to substitute one's own version instead of the default location.

Finally, the method of packaging caters to a final subset of developers. Rather than taking the source of Twine and creating a new version that includes the audio controls, we decided to create the controls in a passage in a Twine story file. This way, the controls are visible for those who want to edit them, and for those that want to add sounds to an existing story, the process is as simple as copying the passages in the desired release into the existing story.

Altogether, the release comes in two forms, with the engine source code available separately. This ensures that Twine stories using it may be existing or new, and fully blind-accessible or not. Developers may choose to look at the sources for the extensions, or never consider them. They may decide to edit them, or change the engine itself and use a new version. Nearly every aspect of these tools is transparent and available to make sure that as many developers as possible can use them to create more audio adventures.



About

Twine Audio is an extension to Twine, the interactive story creator, but we are in no way affiliated with Twine proper. Twine Audio uses the Web Audio API to add support for playing sounds, fading between them, queuing, and more. It can be added to any Twine game with the addition of a few "script" passages, or you can download the frame and start fresh from there.

In addition, Twine Audio includes a supplementary extension that allows the player to advance passages with the number keys. Together, they allow you to create complete audio experiences, which are also blind accessible.

There are a couple of caveats, though. Because Web Audio is still in development, and has yet to be implemented in all browsers, any player must have a compatible browser to play with sound. Take a look here to see what browsers work with Web Audio.

Furthermore, web browsers can't access files (like sound files) from your local computer. That means that while Twine can create web pages that are playable anywhere, a game using Twine Audio must be hosted online somewhere.

Twine Audio and the game created with it, Grail to the Thief, were created by DJ White (djwhitema@gmail.com) and Anthony Russo (amrusso351@gmail.com) for our final project at WPI. DJ was in charge of tech, and Anthony was in charge of writing, sound, and design.

Figure 10: Release Web site at TwineAudio.com

Accessibility

Grail to the Thief is completely blind accessible. However, there were some developer limitations that we would like to see addressed in the future if people decide to expand on our engine. These are, notably, that audio input would be really helpful and interesting, as well as combining the numbered input we utilized and giving an option to speak those numbers.

Also, because it depends on the Web Audio API, the game currently only works in Webkit-based browsers such as Chrome and Safari. However, Web Audio has now been officially adopted as part of the HTML5 standard, and should soon be available on all major browsers. Mozilla has already announced its intention to support it with Firefox.

Final Testing

Greg continued to test the game throughout the production process, constantly trying new iterations of the design. A total of 111 volunteers (55 male, 56 female) played the game, and were randomly assigned one of three variables to test with. They either played the game only seeing text on the screen, only hearing audio, or both seeing the test and hearing the audio. Each participant played the game for five minutes, filled out a survey (included below), and gave their opinions on their experience. Greg also saved their transcripts of their play-throughs for further evaluation.

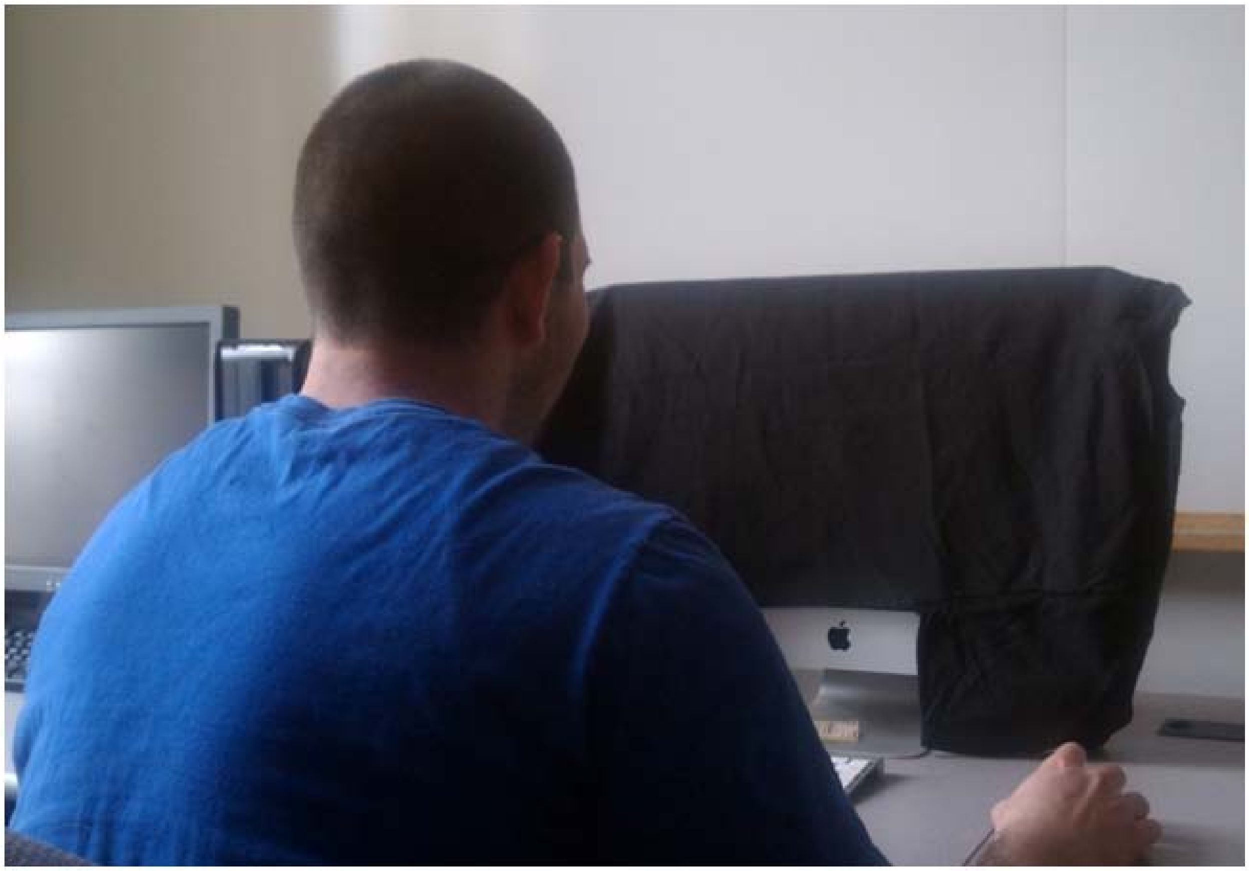


Figure 11: A volunteer playing the game without onscreen text

The results we saw with the Twine version of the game were the opposite of the Inform version. High scores were assigned to engagement, enjoyment, story and sound, and low scores given to frustration, confusion and game difficulty. Several participants wrote in the comment section of their survey that they would like to continue playing it because they were genuinely having fun, which was one of our primary goals.

Interestingly, players who took the time to listen to the audio and not look at the text thought the story flowed more smoothly, suggesting that this game would be enjoyed by the blind. Unfortunately, due to logistic and legal issues, we were not able to perform formal tests with actual blind participants. However, Anthony sent the game over to the Perkins School, and anecdotal feedback has been highly positive.

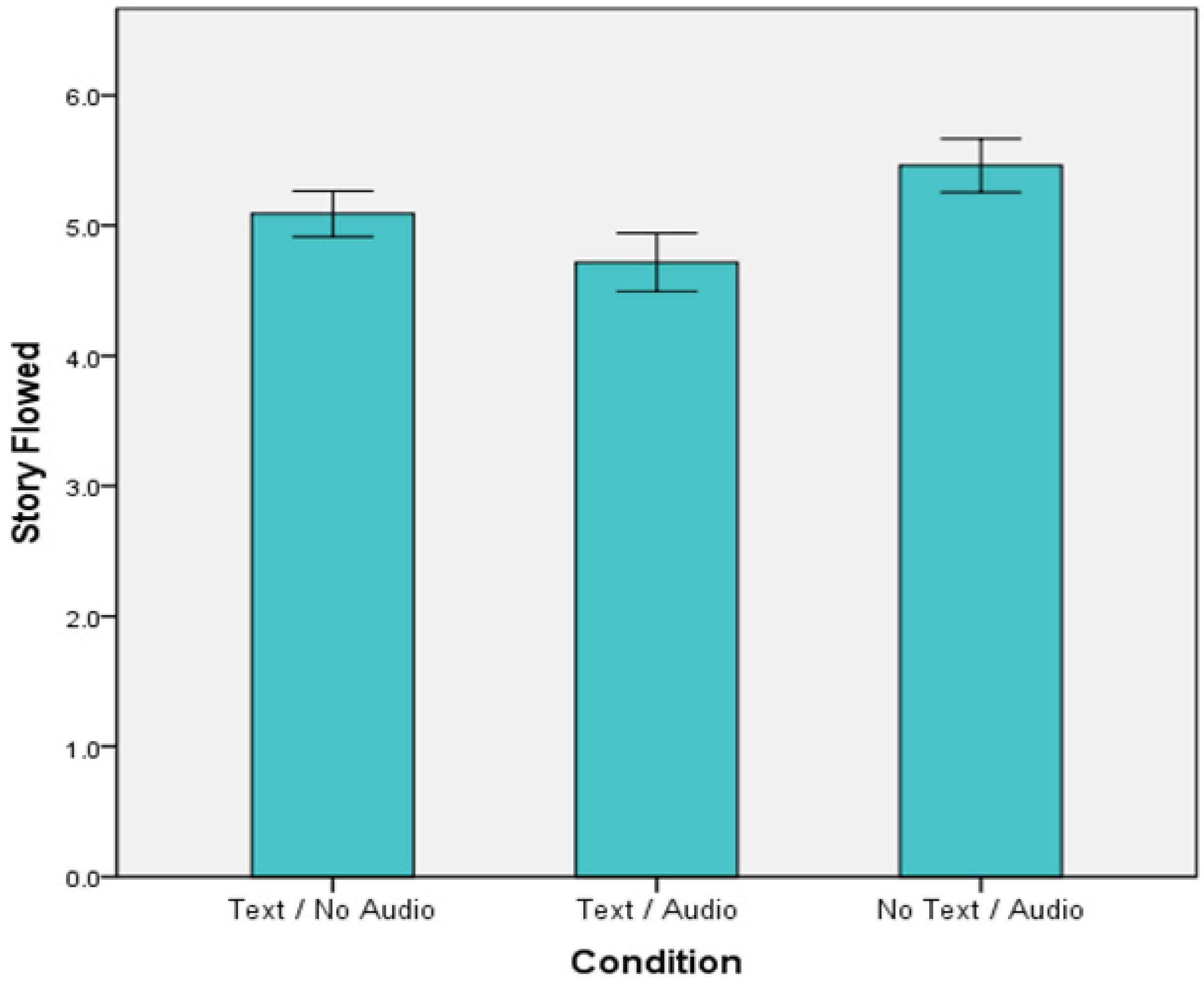


Figure 12: A graph demonstrating enjoyment of the variables on a seven-point scale

Conclusion

Not only did the group create a game for the visually impaired that is completely accessible, but also we rigorously tested and iterated on our design to ensure that the game was immersive, enjoyable, and what every game should be, fun. Also, on our journey, we created a tool to help other create games for the visually impaired and released it to the public for free, hoping to promote awareness and get other developers to create their own blind accessible games.

References

1. Russo, Anthony, Neal Sacks, and Steven Vandal. "Creating Video Games for the Visually Impaired." *WPI Publishing*. WPI, May 2012. Web. 22 Apr. 2013.

Appendices

A. Post-Play Survey

1. I would continue playing this game if I had the opportunity to.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

2. I had a lot of fun playing this game.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

3. I was engaged while I was playing this game.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

4. I was immersed in this game while I was playing it.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

5. I felt frustrated while I was playing this game.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

6. My mind wandered to topics unrelated to the game while I was playing it.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

7. I found the story in this game engaging.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

8. I found the story in this game humorous.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

9. The choices I made in the game affected the story in the way that I wanted it to.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

10. The events and challenges in the game flowed naturally together.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

11. The puzzles and challenges in the game were too difficult.

1	2	3	4	5	6	7
Strongly Disagree						Strongly Agree

12. The storyline of the game flowed naturally, was easy to follow, and was engaging.

1 2 3 4 5 6 7

Strongly Disagree Strongly Agree

13. Only answer if you heard sound while playing the game: The sounds matched the scenarios in the game.

1 2 3 4 5 6 7

Strongly Disagree Strongly Agree

14. Only answer if you heard sound while playing the game: The sounds in the game were realistic.

1 2 3 4 5 6 7

Strongly Disagree Strongly Agree

15. Only answer if you heard sound while playing the game: The audio of the game added a great deal to my enjoyment when playing the game.

1 2 3 4 5 6 7

Strongly Disagree Strongly Agree

16. How often do you play video games?

1 2 3 4 5 6 7

Never Very Often

16. Approximately how many hours per week do you play video games?

____

17. If you were to play a video game, what genres of video games would you play? (Circle all that apply and rank in order of preference)

-Action	-Adventure	-Facebook/Online	-Fighting
-Mobile/Phone	-Music	-Multiplayer	-Puzzle
-Racing	-Role-playing	-Simulation	-Strategy
-Shooter	-Sports	-Massively Multiplayer Online	

-Other: ____

18. What is your gender?

Male   Female

19. What is your class year?

Freshmen   Sophomore   Junior   Senior   Graduate

20. Do you have any other comments about this game?

B. Twine API

```
/*********************************

WAIT! Be sure that you set (NUMBER OF SOUNDS) to the correct number. This
tells the audio engine when it's okay to continue on and start the game. Take
note though: This won't make sure that the engine waits until every sound has
loaded, just that they have been attempted. The sound is counted regardless
of whether a sound is successfully loaded or not.

The game still starts on the passage titled "Start", but this is just a
landing pad: it can say whatever you want or nothing at all. After a moment,
the engine will move the game along to another page, this one titled
"Loading". Once all sounds have loaded (or failed to load), the game will
automatically move on to the new start, titled "Start_Game".

Okay, on to how to set up the engine in Twine passages:

Step 1: Call <<InitMacro>>before any other macro, on the passage titled
"Loading".

Step 2: Call all the <<AddMacro [Sound_Name]
['/File_Path/File_Name.ogg']>>without the brackets. [Sound_Name] is whatever
you want to refer to sound as. The filepath can be local or explicit,
JavaScript isn't picky. Another  note here: Twine won't understand if you
have a space in any of these names (or anywhere else), so stick to
underscores and the like.

Step 3: Call all the <<LoadMacro [Sound_Name]>>. You only need the name now,
not the path. This loads the sound into memory, so it can definitely take
some time for a lot of sounds, or a few long ones.

All of this should be done in the passage named "Loading". You can simply put
the <<InitMacro>>, followed by a block of <<AddMacro>>s, followed by a block
of <<LoadMacro>>s. However, anything else must come in another passage.

The real start passage is now called "Start_Game". (If you'd like, you can
change this just below, in InitMacro).

Now that our engine is started, and we have all our sounds, we can use them:

Step 1: At the start of any passage, call <<StopOnceMacro>>. This will stop
ANY sounds that is not ambient, so a player clicking rapidly will still only
hear the current passage.

Step 2: Also at the start of any passage, but unrelated to the previous, call
<<ClearRepeatMacro>>. This cleans the list of commands that play when a
player presses the repeat button (or the spacebar).

Step 3: Whenever you play a sound in any way, if you want it to be repeated,
call
<<AddRepeatMacro [Macro_Name] [Macro_Paramaters] ... >>. It's a good idea to
add the repeat to the list when the sound plays. That way, they will be in
the same order in the repeat list as the first time they play.
```

Here is a list of all macros with descriptions (type all parameters without the brackets):

<<InitMacro>>

Starts the audio engine (Creates the AudioContext). Call this before anything else.

<<AddMacro [Sound_Name] ['/Filepath/File_Name.ogg']>>

Adds [Sound_Name] to the list of all sounds. After being added, the sound can just be referred to as [Sound_Name].

Example: *<<AddMacro horse_gallop '/horse_galloping.wav'>>*

<<LoadMacro [Sound_Name]>>

Loads [Sound_Name] into memory to play it later.

Example: *<<LoadMacro horse_gallop>>*

<<StopOnceMacro>>

Stops all sounds intended to play once. Any sound not labeled as "ambient" is assumed to only play on its own passage. This should be placed at the start of every passage, to keep sounds from piling up.

<<StopMacro [Sound_Name]>>

Manually stops [Sound_Name], if it is playing. Since most sounds play once, this is best for ambient sounds.

Example: *<<StopMacro horse_gallop>>*

<<PlayMacro [Sound_Name] [Volume] [Loop] [Delay] [Ambient]>>

Plays [Sound_Name]. This gives you the most control over individual sounds.

[Volume] is a number from 0-1, default is 1.

[Loop] is a Boolean, default is false.

[Delay] is a number in seconds, default is 0.

[Ambient] is a Boolean, default is false.

You can play a sound with only the name, but if you want to change a later parameter, you must enter
everything before it. Anything not labeled ambient is added to a list of sounds to be played for a single passage, even if it is looping. These will be stopped if *<<StopOnceMacro>>*is called.

Example: *<<PlayMacro horse_gallop>>*assumes *<<PlayMacro horse_gallop 1 false 0 false>>*

<<FadeOutMacro [Sound_Name] [Time] [Volume] [Ambient]>>

Plays [Sound_Name], and fades it out. Time is the time from the end to start fading out. Volume is the target volume.

The sound starts at normal volume (1), so a target of 0.5 is half of the starting volume.

[Time] is a number in seconds, default is 3.

[Volume] is a number from 0-1, default is 0.5.

[Ambient] is a Boolean, default is false.

You can play a sound with only the name, but if you want to change a later parameter, you must enter
everything before it.
Example: *<<FadeOutMacro horse_gallop>>*assumes *<<FadeOutMacro horse_gallop 3 0.5 false>>*

<<FadeInMacro [Sound_Name] [Time] [Volume] [Ambient]>>

Plays [Sound_Name], and fades it in. Time is the amount of time to fade in. Volume is the starting volume. The sound starts at that volume and gradually reaches normal (1), so a start of 0.5 is half of the volume it will reach.

[Time] is a number in seconds, default is 3.

[Volume] is a number from 0-1, default is 0.5.

[Ambient] is a Boolean, default is false.

You can play a sound with only the name, but if you want to change a later parameter, you must enter
everything before it.

Example: *<<FadeInMacro horse_gallop>>*assumes *<<FadeInMacro horse_gallop 3 0.5 false>>*

<<CrossfadeMacro [Sound_Name_1] [Sound_Name_2] [Time] [Volume] [Ambient] >>

Combines the above two macros to crossfade two sounds. The [Sound_Name_1] will fade out for the length of [Time], and [Sound_Name_2] will fade in for that time.

[Time] is a number in seconds, default is 3.

[Volume] is a number from 0-1, default is 0.5.

[Ambient] is a Boolean, default is false.

You can play a sound with only the names, but if you want to change a later parameter, you must enter
everything before it.

Example: *<<CrossfadeMacro horse_gallop horse_whinny>>*assumes *<<FadeInMacro horse_gallop horse_whinny 3 0.5 false>>*

<<QueueMacro [Delay] [Ambient] [Sound_Names]>>

Plays a list of up to 50 sound names, listed individually. Each sound will wait for the given [Delay].

[Delay] is a number.

[Ambient] is a Boolean.

Example: *<<QueueMacro 0 false horse_gallop horse_whinny dust_cloud>>*

<<RandomMacro [Delay] [Ambient] [Sound_Names]>>

Plays a list of up to 50 sound names randomly. The sounds must be listed individually. Great for ambient sounds. Each sound will wait for the given [Delay].

[Delay] is a number.

[Ambient] is a Boolean.

Example: *<<RandomMacro 5 true birds_chirping cow_moo rooster_crow>>*

<<AddRepeatMacro [Macro_Name] [Parameters]>>

Adds any of the playing functions above to the Repeat List. The list preserves the order in which they are added.

Repeat List is the list of functions to call when the player clicks "Repeat Audio" or presses the space bar.

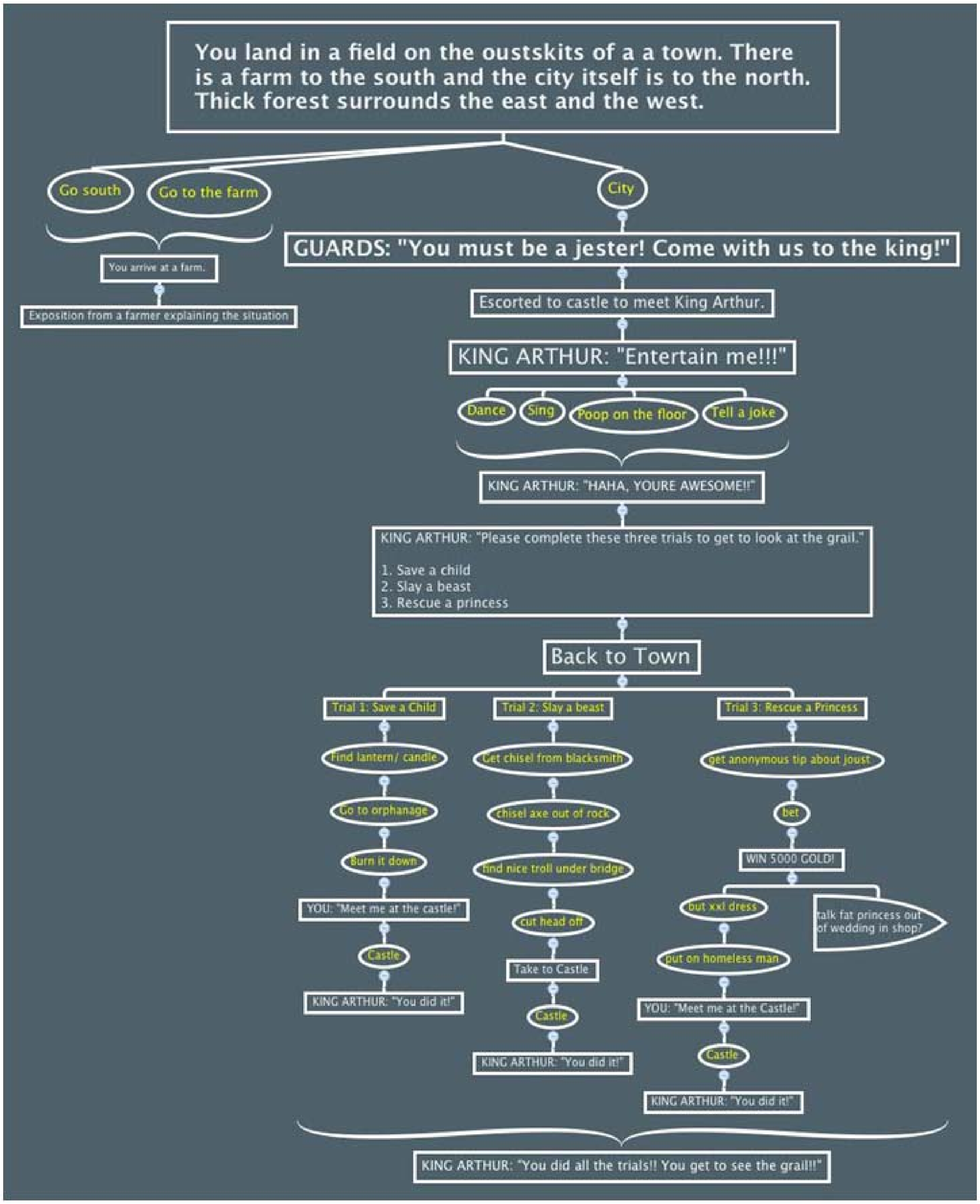
Example: *<<AddRepeatMacro QueueMacro 0 false horse_gallop horse_whinny dust_cloud>>*

<<ClearRepeatMacro>>

Clears the Repeat List. Like *<<ClearOnceMacro>>*, this is intended to be called at the start of every passage.

***/

C. XMind story layout


You land in a field on the oustskits of a a town. There is a farm to the south and the city itself is to the north. Thick forest surrounds the east and the west.
Go south
Go to the farm
City
You arrive at a farm.
Exposition from a farmer explaining the situation
GUARDS: "You must be a jester! Come with us to the king!"
Escorted to castle to meet King Arthur.
KING ARTHUR: "Entertain me!!!"
Dance
Sing
Poop on the floor
Tell a joke
KING ARTHUR: "HAHA, YOURE AWESOME!!"
KING ARTHUR: "Please complete these three trials to get to look at the grail." 1. Save a child 2. Slay a beast 3. Rescue a princess
Back to Town
Trial 1: Save a Child
Trial 2: Slay a beast
Trial 3: Rescue a Princess
Find lantern/ candle
Go to orphanage
Burn it down
YOU: "Meet me at the castle!"
Castle
KING ARTHUR: "You did it!"
Get chisel from blacksmith
chisel axe out of rock
find nice troll under bridge
cut head off
Take to Castle
Castle
KING ARTHUR: "You did it!"
get anonymous tip about joust
bet
WIN 5000 GOLD!
but xxl dress
talk fat princess out of wedding in shop?
put on homeless man
YOU: "Meet me at the Castle!"
Castle
KING ARTHUR: "You did it!"
KING ARTHUR: "You did all the trials!! You get to see the grail!!"