# Procedural Systems for Non-Combat RPGs

Alexander Baird and Squirrel Eiserloh

***Abstract*— The artifact created for this is a non-combat roleplaying game creation system. Quests and characters are procedurally generated through XML data. This article details the concepts behind the systems in the demo.**

***Index Terms*— Video Games, Roleplaying Games or RPGs, Procedural Generation, No Combat System**

## I. Introduction

THE author has explored various ways in which to use procedural generation systems for a RPG with no combat, chiefly through character and quest procedural generation. The author created the artifact within seven months.

Roleplaying Games (RPGs) traditionally rely heavily upon combat, but often include other elements such as quests, stats, skills, minigames, and procedural generation. Very few developers have tried to completely remove combat from a RPG as most RPG systems directly or indirectly serve combat as an ends. If there is no combat system, then a game creator must provide interest and replayability through alternative means.

## II. Research Review

A plethora of games have incorporated procedural generation in order to make them more interesting; this could be through dungeon or world generation [1] [2] [3] [4], character generation [5] [6] [7], or other methods such as event and story generation [8] [9] [10] [11]. Procedural generation is where the game generates some or all of its own content on the fly, through algorithms that may or may not accept human-designer input [9]. The most common way to use procedural generation in games has been to generate the world or dungeon that the player explores [3]. Some methods for dungeon or world generation include cellular automata, generative grammars, generative algorithms, and more [3]. One of the more advanced and interesting methods for generating content is through a genetic algorithm (an artificial intelligence), in which the computer generates several variants, judges which ones have the best results, then mutates its formula to try to achieve better results in the next iteration [3].

One popular method for procedural map generation is to take human-authored modular content and have the computer generate a floor using those pieces [1] [2]. *Mabinogi* takes a much simpler approach to the dungeon generation in that it does eschews the introduction of many complex rooms into its dungeon generation [1]. Rooms with locked doors are always preceded with monster rooms which contain the correct keys, insuring that those keys match the doors after them [1]. These dungeons still make room to produce dead ends for the dungeon, which abstracts how linear these dungeons are [1]. *Mabinogi* always produces results that enable the player to fully explore the dungeon [1]. Although *Mabinogi* has a minimalistic layout for many rooms and hallways, the pathing is always interesting in that each dungeon has different scenery; most dungeons use regular stone walls, but the outdoor dungeons use lots of trees and plants [1].

*Runescape* on the other hand takes a similar but more complex approach to dungeon generation, having not just doors that require keys, but also skill doors and puzzle doors [2]. Due to this complexity, some of chunks of the dungeon floor tend to be always locked off to the player; this is as the dungeon generates skill doors above the player's skill level [2]. Players which possess a certain skill (herblore, aka potion making) at a high enough level can overcome these the requirements they do not meet for these special doors by crafting potions that give them temporary buffs [2]. However, that option is not always available, and the dungeon fails to account for when the player does not have the appropriate herblore level to make said potion [2]. One could argue that this is broken design, but at the same time it is incentive for the player to train the skills to which the player may previously have turned a blind eye [2]. *FTL: Faster Than Light* also uses procedural generation to create its Beacon and Galaxy maps, each beacon treated as a dungeon room with a single event contained within, while the Galaxy the player is in defines the overarching theme of the events the player might encounter [8]. *Audiosurf* is a good example of procedural map generation in a game with no combat; *Audiosurf* generates the map and obstacles based on the sound waves from a song [4].

**Alexander Baird is a student at Southern Methodist University Guildhall (SMU Guildhall) (e-mail: atbaird@smu.edu)**, he received his bachelor's degree in May 2015 (B.S. Computer Science) from SMU and is presently working towards his Master's degree at SMU Guildhall**.**

**Squirrel Eiserloh is a professor at SMU Guildhall. Squirrel Eiserloh** is a game programming faculty Lecturer at SMU Guildhall, Southern Methodist University's game development graduate program. Since he graduated from Taylor University in 1996 (B.A. Physics) he has been working as a professional game developer in the Dallas area, contributing to over a dozen commercial game titles. He co-chairs the Dallas chapter of the IGDA, and coordinates the Math for Game Programmers sessions at the annual Game Developers Conference in San Francisco.

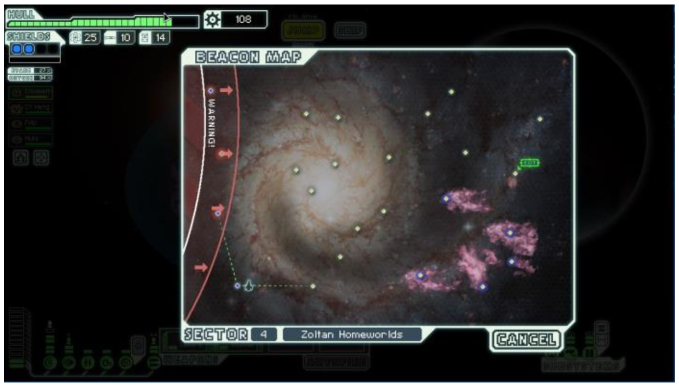


***Figure 1**: FTL: Faster than Light Beacon Map view.*

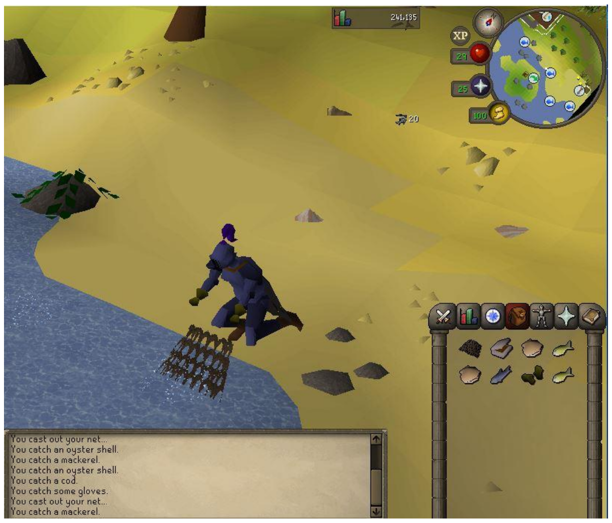
Some games have also incorporated abilities and skills that are non-combat focused, usually to break up the core gameplay flow [1] [2]. These games typically use abilities to enable the player to craft equipment or consumables to sell or use [1] [2]. *Runescape* chooses to make non-combat skills all generic, and good for casual play [2]. Each skill has a unique animation, but that is where the interest ends, as the tasks are generally just click-and-watch as the character performs the action [2]; *Mabinogi* takes the opportunity to transform these non-combat skills into its own mini-games [1]. For example, in order to smith weaponry, a clickable mini-screen opens for the player with specific dots on it for the player to click; the closer to the center of each dot the player clicks, the better quality the equipment [1]. Skills have also been used in *Runescape*'s procedurally generated dungeons, doors lock off the player unless she has the correct skill level [2]. *Mabinogi* also makes skills interesting to collect by starting the player with a handful that they know about, and she must perform quests and talk to Non-Player Characters (NPCs) in order to obtain the rest [1]. This approach could be interesting if combined with procedural world generation [1].

Another potential way to use procedural generation is by generating the story, or events within the game [9]. In games like this, it is better to view the story less like a line and more like a collection of episodes or volumes, since a fair number of the events that occur will likely be a bit more disconnected from the rest [9] [8]. Coding for these chains of events is best done by planning them out as partial order plans; a partial order plan defines a list of requirements, a list of actions to be taken if those requirements are met, but does not explicitly define which event explicitly comes before another [12]. After which, it is important to enable the ability to define location requirements for the associated entities; e.g. for NPCs to listen to their normal behavioral scripts and give priority to the story script when it calls for their participation [12] [13]. Some of this can be seen through *Crusader Kings II* in which the NPC kingdoms, initially, follow a particular chain of events at the start of the game [6]. From there, they eventually run based on the regular behavior scripts to try and win the game, changing how they behave based off the traits of each family member of the ruling family [6].

*Façade* is a game prototype that uses an artificial intelligence (AI) to procedurally generate narrative based on the player's interactions with the world and the two NPCs (Grace and Trip) [10]. This keeps the game interesting by changing out how it starts each time played, as well as change the story based off the player's decisions; Façade also avoids centering around combat by instead being built around drama [10]. The player's primary way to interact with the NPCs is to type out a line of text and hit enter; from there the AI recognizes key words in the sentence and selects the closest matching event to trigger [10]. The slight changes are enough to make it replayable and guarantees each playthrough is unique [10].

Another example of procedural story generation in action can be viewed through *Versu* [11]. *Versu* is a system for creating procedurally generated stories, which focuses on agent-driven story generation [11]. Agent-driven story generation involves characters reacting based on other characters' actions, as well as their own personallity traits (likes, dislikes, beliefs, and so on); this allows characters to act independently and for their roles to be easily interchangeable within the story [11]. Versu's big difference from Façade is that Façade's story generation is based on specific events; this forces characters to have to follow the explicit script laid down by an event [10] [11]. That also makes it to where Façade's characters can take longer to change out or make new characters for events [10] [11].

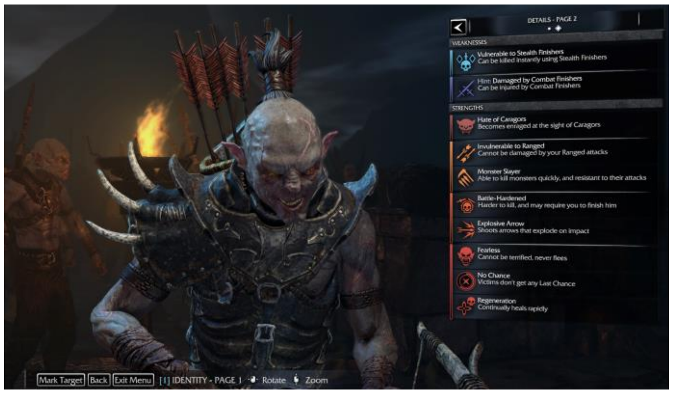
In *FTL: Faster Than Light*, the beginning and end points of the story are set in stone, but the events along the way are randomly generated and follow a particular theme based on which galaxy the player is presently in [8]. It is worth noting that if the depth, breadth or complexity of events is lacking, that the player may eventually catch on to the procedural generation occuring in the story, and grow bored; though it may take several playthroughs to notice [8]. Designers who intend is to combine procedural storytelling with procedural world generation, should consider the complexity of the world required by the game [13]. *FTL: Faster Than Light* kept their world generation very simple so as to avoid the complex issue of defining locations with more granularity than their galaxies and solar system waypoints [8]. If the events and world have complex location requirements, the world needs to be able to inform the event system important objects and locations [13].



```
You cast out your net...
You catch an oyster shell.
You catch a mackerel.
You catch an oyster shell.
You catch a cod.
You catch some gloves.
You cast out your net...
You catch a mackerel.
```

***Figure 2**: Runescape a player using a big net to fish.*

In the space of procedurally generating characters, it is important to focus on not just appearance a character; stats, personality, and other traits also play a role in procedurally generating a character [6] [5]. In *Crusader Kings II*, the changing traits of the ruling characters can cause the player to have to change her play style in order to continue to progress [6]. For exmaple the player could have been playing with diplomacy before, but then suddenly their current ruler passes, and the new ruler is a horrible diplomat, but a good military leader [6]. In *Middle Earth: Shadow of Mordor*, character procedural generation is used to generate unique orcs with different buffs and weaknesses [5]. These orcs might die only once and be forgotten, or might constantly come back to hunt the player [5]. This type of procedural generation can enable the player to get attached to particular characters due to their personallity and perserverance [5] [6]. This type of procedural generation also pairs well with procedurally generating events; *Middle Earth: Shadow of Mordor* used procedurally generated events as missions for hunting the lead orcs [5].



DETAILS - PAGE 2

WEAKNESSES

Vulnerable to Stealth Finishers
Can be killed instantly using Stealth Finishers

Him Damaged by Combat Finishers
Can be injured by Combat Finishers

STRENGTHS

Hate of Caragors
Becomes enraged at the sight of Caragors

Invulnerable to Ranged
Cannot be damaged by your Ranged attacks

Monster Slayer
Able to kill monsters quickly, and resistant to their attacks

Battle-Hardened
Harder to kill, and may require you to finish him

Explosive Arrow
Shoots arrows that explode on impact

Fearless
Cannot be terrified, never flees

No Chance
Victims don't get any Last Chance

Regeneration
Continually heals rapidly

***Figure 3**: Middle Earth: Shadow of Mordor *Orc Traits screen.*

Another example of Procedurally generated characters can be viewed through *Bad News*: a mixed reality game where the characters and map are procedurally generated before the performance [7]. In the game, the player's objective is to discover a deceased character's identity and notify their next of kin; the commands for the game are spoken allowed such that an operator (or "wizard" as they refer to the role) makes alterations to the simulation, characters are performed by live actors [7]. The characters being procedurally generated, as well as the town layout, makes it to where each experience is fun, and guarantees that past players can not spoil the fun; the actors have a card with what they are suppose to look like, how they act, and what topics they know about [7]. The game does provide a good example of how procedurally generating characters can provide unique interactions and a fun experience [7]. Simple changes in personallity change how to persuade a charcter to cooperate, or what they are able to say on certain matters [7].

As far as procedurally generating characters or items is concerned: its worth it to investigate changing colors of in game sprites on the fly; not only does it pair well with procedurally generating characters, but it would also leads to allowing player's to customize their character's appearance [6] [1]. *Crusader Kings II* procedurally generate its character appearances; however, it also gives the player the option to change the names and portraits of the characters she controls. In *Mabinogi* players receive equipment with procedurally generated colors from shops and monster drops; however there are dye items in the game that players spend absorbinent amounts of gold in order to change their equipments colors to what they like [6] [1].

In order to create an RPG without a combat system, it is important to pick out the stats and skills that are used carefully, as these will enable the player's ability to move around the world as well as what can be used to effect the NPC personallities [2] [5] [1] [6]. After which, it is a good idea to look at procedurally generating a series of events, and associated characters into the world [8] [9] [13]. The time needed for a game to be develop increases rapidly the more content is needed to be procedurally generated [3] [13] [12]. This is as the algorithms used for procedurally generating content needs to be made smarter in order to avoid the game stagnating [3] [13] [12].

## III. Methodology

### *The Game: Design*

The artifact created for this thesis is a role playing game creation system developed within a custom C++ game engine. The system allows for procedural generation of characters and quests, all of which are defined through data. This allows the designer to easily alter gameplay and piece together whatever events and content a designer could want and are read in at the start of the program. The map uses a tile based grid, while characters move freely without regard to the grid. The player can interact with various objects in the world, talk to NPCs and perform quests for those NPCs.

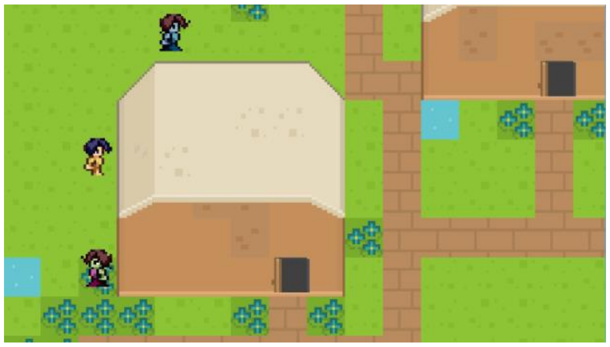


***Figure 4**: A sample screenshot from the artifact, showing a map, plus three characters.*

### *The Game: Implementation*

The artifact uses a custom C++ engine which renders graphics via OpenGL, and plays sound through FMOD. The majority of the content in the game is data driven, as it makes it easier for a designer to author the user experience, and allows for easy customization of the game. Game aspects that are data-driven include: *Agents*, *Agent Stats*, *Features*, *Global Variables*, *Hair*, *Clothing*, *Map Data*, *Map Files*, *Name Generators*, *NPC Attributes*, *NPC Jobs*, *Quests*, *Sounds*, *SpritAnimations*, *Sprite Resources*, and the dialogue system. *Sprite Resources* contain a file location for a texture, some texture coordinates, and a resource name to refernce it within the game (for example the image for a single grass tile). *Sprite*

*Animations* use Sprite Resources for individual frames, as well as an associated duration per frame.

```
<SpriteResources>
  <SpriteResource resourceName='human_baseForward0'
          textureLocation='DataSet1/Images/Characters/hu
man_base.png' spriteBounds='0.0, 54.0, 15.0, 36.0'/>
  <SpriteResource resourceName='human_baseForward1'
          textureLocation='DataSet1/Images/Characters/hu
man_base.png' spriteBounds='16.0, 54.0, 31.0, 36.0'/>
  <SpriteResource resourceName='human_baseForward2'
          textureLocation='DataSet1/Images/Characters/hu
man_base.png' spriteBounds='32.0, 54.0, 47.0, 36.0'/>
</SpriteResources>

<SpriteAnimation
          spriteAnimationName='human_base0_walkForward'
          defaultIntervalBetweenFrames='0.125'
          animationMode='loop' reverseDirection='false'>
    <Frame resourceName='human_baseForward1'/>
    <Frame resourceName='human_baseForward2'/>
    <Frame resourceName='human_baseForward1'/>
    <Frame resourceName='human_baseForward0'/>
 </SpriteAnimation>
```

***Figure 5****: An example of several sprite resources and an associated sprite animation XML.*

An *Agent* contains the information for every single character in the game, and has two variants (*Player* and *NPC*). Agents also have Hair and Clothing which are a series of animations. *Tile Definitions* define whether a tile is solid or not, as well as what sprite the tile has. *Name Generators* are used to generate a name for each Agent depending on gender (male, female) and race (orc, elf, human, etc.). *Map Files* contain file paths for a map's image and data, such as tile events, agents to spawn, and quests to use. *Quests* are a bunch of events and state data. *Quest Events* are a list of requirements that must be met before it can run, and two lists of triggers to perform when ran. For example, when one event triggers on a quest, the dialogue displayed for an Agent could have changed next time the player speaks to it. *Features* are any cosmetic or interactable objects in the game; for example, the doors in the game are Features that allow the player to move through the tile on which they are placed. Agents, Clothing, Hair, and Features are all *Entities*. The dialogue system handles displaying text and speech bubbles when the player interacts with an NPC.

```
<AgentGenerators>
  <AgentGenerator Name='Human'>
       <!-- ... -->
  </AgentGenerator>
</AgentGenerators>
```

***Figure 6****: An example Agent Generator declaration.*

The quests and the different Entity types each use a factory paradigm; this paradigm is where data is read in from XML, and parsed into a template, and when called spawns an entity as per the specifications of the data (procedurally generating as necessary). For example, the data for an Agent is read in and is named Villager, thus when a Villager Agent is needed, it will use the Villager Agent generate to spawn one. A resource database paradigm is used each of the different parts of the dialogue system, Sprite Resources, Sprite Animations, Map Files, and Tile Definitions; this paradigm loads in the data from XML, and maintains only one copy of the entity for the code to reference. This variant is necessary as there is no need for multiple versions of this data to exist; e.g. why would the designer want there to be multiple copies of the dialogue "what's up?" in as many different places?

```
Entity* playerEntity =
  AgentGenerator::SpawnPlayerInAgentGeneratorByAgentName(
      "human", vec2_pos, map_ptr);
Entity* npcEntity =
  AgentGenerator::SpawnNPCInAgentGeneratorByAgentName(
      "human", vec2_pos, map_ptr);
```

***Figure 7****: Examples of how to spawn different Agents from the Agent Generators.*

### *Map and Tile Definitions*

A *Tile Definition* is a set parameters for a tile to follow; it contains a color code, four layers of sprite resources, as well as whether the tile is solid or not. It is also possible to define events to trigger when the player enters, touches, or exits the tile. Tile variantions can be added by naming alternative sprite resources and a percentage chance for them to occur. This allows for a grass tile to have variations.

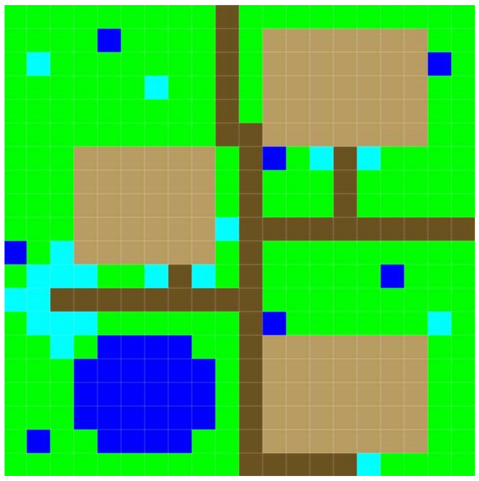


***Figure 8****: 20x20 source map image; 1 pixel is 1 map tile.*

```
<TileDefinitions>
  <TileDefinition name='grass1_1' MapCode='0,255,0'
    resourceName='grassTile1_1'
    altResourceName='grassTile1_1_alt'
    altResourceChance='0.5f' solid='false'/>
</TileDefinitions>
```

***Figure 9****: Example XML TileDefinition.*

However, most designers are not going to want to define a unique tile definition for each potential part of a house, then painstakingly change the colors used per tile of the house. In order to handle this, special case XML nodes are required within the Tile Definition XML. Based on a set of user defined rules, a special case overwrites the originally set parameters with a new set. A special case uses requirement tests against a named Tile Definition or the same one the designer is adding the special case to; these requirement tests can be performed against the same or different tile definitions to the north, east, south, and west. Each tile definition also contains a list of definition names to treat the same as it, for when the special cases are testing for it. This allows for a stove tile to be given a unique tile definition and placed within the house without the entire roof shifting down several tiles (if the roof placed through special cases).

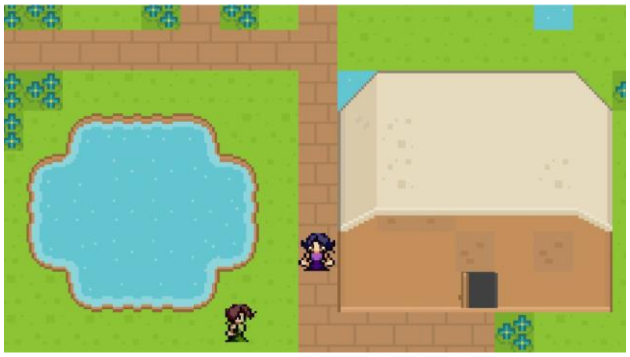
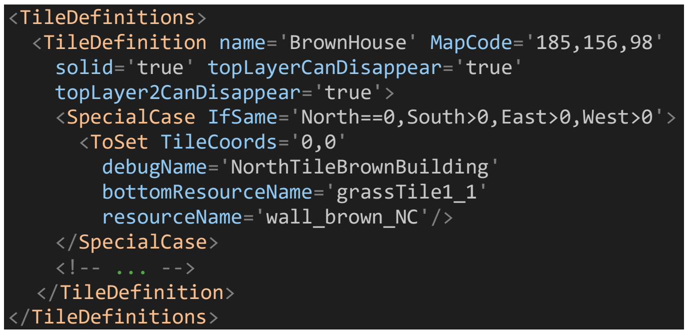


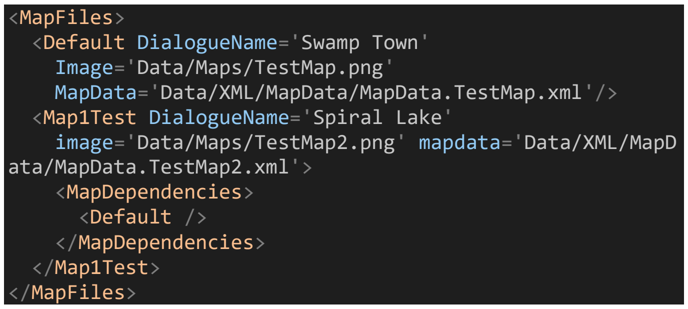
*Figure 10: A screen shot of the brown house and lake tile definitions in action.*



```
<TileDefinitions>
  <TileDefinition name='BrownHouse' MapCode='185,156,98'
    solid='true' topLayerCanDisappear='true'
    topLayer2CanDisappear='true'>
    <SpecialCase IfSame='North==0,South>0,East>0,West>0'>
      <ToSet TileCoords='0,0'
        debugName='NorthTileBrownBuilding'
        bottomResourceName='grassTile1_1'
        resourceName='wall_brown_NC'/>
    </SpecialCase>
    <!-- ... -->
  </TileDefinition>
</TileDefinitions>
```

*Figure 11: Example XML for TileDefinitions with special cases.*

The next important ability for the map is to be able to define tile events, as well as how to spawn in quests and entities. For this, there is an XML file associated with each map. This data is read after the image data is fully read. The *Map Data* XML node defines what music on the map, and how the camera is handled for that map. By default, no music plays on the map, and the camera is set to clamp to the edges of the map (clamp or unconstrained). Within the Map Data node, a designer defines what Tile Events, Features (interactable objects), NPCs, and Quests entities are placed in the map, as well as where. Positioning of the entities can be totally random or picked from a list of positions defined by the designer. *Tile Events* are functions that run based on when an Agent enters, leaves, or touches a Tile, e.g. they are used for switching Maps when the Player enters a tile. For an example of Map Data in XML, see appendix 1.



```
<MapFiles>
  <Default DialogueName='Swamp Town'
    Image='Data/Maps/TestMap.png'
    MapData='Data/XML/MapData/MapData.TestMap.xml'/>
  <Map1Test DialogueName='Spiral Lake'
    image='Data/Maps/TestMap2.png' mapdata='Data/XML/MapD
ata/MapData.TestMap2.xml'>
    <MapDependencies>
      <Default />
    </MapDependencies>
  </Map1Test>
</MapFiles>
```

*Figure 12: An example Map Files in XML.*

*Procedural Character Generation and Features*

The *NPC* and *Player* classes both inherit from the Agent class, where an *Agent* is any character or mobile entity in the game. *Features* are different from agents in that they define non-moving entities that can be either cosmetic or interactible. Features can also change how the tiles they are on interact with the player.

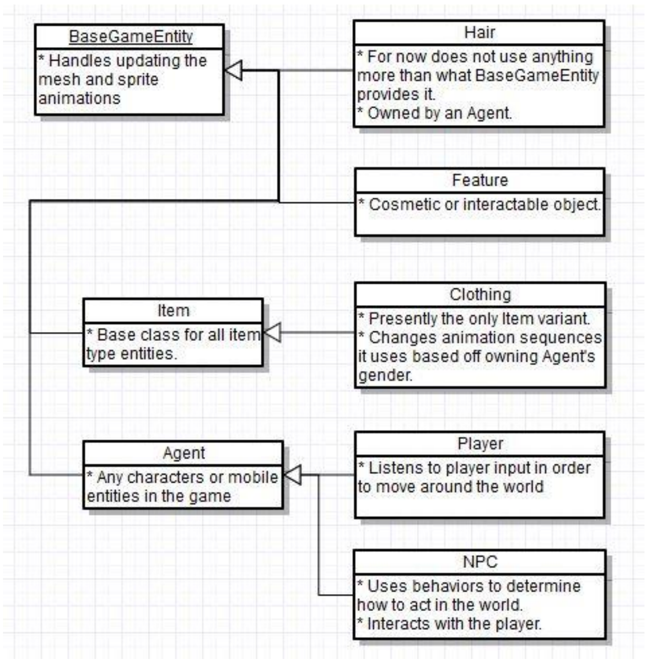

BaseGameEntity
* Handles updating the mesh and sprite animations
Hair
* For now does not use anything more than what BaseGameEntity provides it.
* Owned by an Agent.
Feature
* Cosmetic or interactable object.
Item
* Base class for all item type entities.
Clothing
* Presently the only Item variant.
* Changes animation sequences it uses based off owning Agent's gender.
Agent
* Any characters or mobile entities in the game
Player
* Listens to player input in order to move around the world
NPC
* Uses behaviors to determine how to act in the world.
* Interacts with the player.

*Figure 13: Class diagram for Entities.*

Before creating an *Agent Generators*, several other data types must be defined, including: Agent Stats, Sprite Resources, Sprite Animations, Hair Generators, Item Generators for Clothing, NPC Attributes, and NPC Jobs. NPCs with dialogue reference a corresponding *Dialogue Segment* by name, thus the that will also need to be defined.

*Agent Stats* are an integer range associated with a name. Agent Stats first are defined in their common XML folder; the stats read here will be applied to every agent that is ever created. At the start of the game, the stats will be procedurally generated based off the min and max set at the start; if the stat is changed later, its new value is clamped to be within the min and max. The Agent Generators can explicitly set its own ranges for each Agent Stat. When this happens, the Agent will procedurally generate its stats based on the Agent Generator's stat ranges in place of the original's. After the stat ranges are procedurally generated, the min and max on the Agent is set back to the default's; this allows for the stat ranges to be increased or decreased within the normal stat range accordingly. For example a stat of strength is defined with a min and max of 0 to 100, then the Agent Generator defines a new stat range for it of 10 to 10; this leads to the Agent Generator procedurally generating a strength stat of 10, and then that agent has a strength min and max of 0 to 100. This is useful as it allows for an Agent Stats to be increased or decreased that were originally defined (such as if the Player's strength was increased by 5 via a Quest's reward).

```xml
<AgentStats minimum='0' maximum='100'>
  <Dexterity abbreviation='dex'/>
  <Intelligence abbreviation='int'
    minimum='0' maximum='100'/>
  <!--
   Default will be 0 to 100, just putting
   this as an example. -->
</AgentStats>
```

***Figure 14**: Example XML of Agent Stats.*

```xml
<AgentGenerators>
  <AgentGenerator Name='Player'
   DefaultImage='human_base0'
   MovementSpeed='2.f'>
<!-- ... -->
    <StatRanges>
      <Strength min='10' max='10'/>
      <Dexterity min='10' max='10'/>
    </StatRanges>
<!-- ... -->
  </AgentGenerator>
```

***Figure 15**: An example Agent Generator defining custom agent stat ranges in its XML data.*

*NPC Jobs* are procedurally generated from a list of jobs the corresponding NPC's stats meet. These jobs have a name, and an associated set of Agent Stat requirements. The NPC Jobs an NPC can have can be limited per Agent Generator by defining a list of Jobs for the generator. If the NPC does not qualify for a job, then it is labeled as jobless.

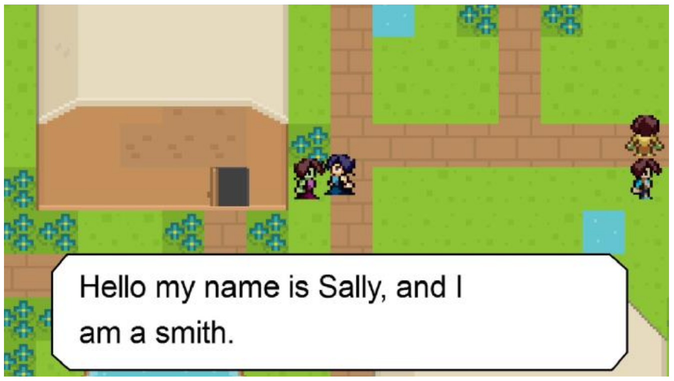

Hello my name is Sally, and I am a smith.

***Figure 16**: An NPC saying what their name and job is.*

NPCs also need to define what they do on update; this is achieved through *NPC Behaviors*. NPC Behaviors in this artifact include: look around, talk, and wander. NPC Behaviors are a series of classes that inherit from the NPC Behavior class, and each have custom code (two virtual functions must be defined for each of them). One virtual function defines how its calculates utility, and the other is the function that performs the chosen behavior's action. The NPC on update takes its full list of behaviors and finds the one with the highest utility score and runs it for that frame. *Features* also use a system very similar to NPC behaviors, but explicitly for *Feature Behaviors*. These *Feature Behaviors* define what to do on interact as well as what to do to a particular tile it is spawned onto.

The NPC Behaviors also use a self registration paradigm in order to make themselves easy to spawn (this also has the added benefit of limiting the number of includes a given file needs to have in order to spawn an NPC Behavior by a particular name). The self registration paradigm involves an external class (*Registration Helper*) being defined with an associated name and two functions that spawn the desired class (one for use with an XML node, one without); the Registration Helper loads itself onto a list upon being constructed. Then for each variant of the desired class, a static Registration Helper is defined; this paradigm takes advantage of game side code, in that all of it is compiled before run time, allowing all of the Registration Helpers to be added to the list before any XML data is read. The classes that use the self registration paradigm include: NPC Behaviors, Feature Behaviors, Tile Events, Dialogue Requirements, Dialogue Actions, Quest Requirements, Quest Triggers, and variations of the Item class.

*Hair* and *Clothing* data has a similar yet simpler structure than characters: a name, default image, and sprite animations for each direction. Hair can define specific colors from which to use, and clothing is able to define an alternative set of animations. For example, Clothing just needs alternative animations, but a potion would need a declaration of what to do when used. Hair and Clothing can then be assigned per agent generator for procedural generation. Hair and Clothing both have gender specific options.

```xml
<HairGenerators>
 <HairGenerator name='male_Townsfolk_hair1'
  isDefault='true' dialogueName='Simple Hair'
  defaultimage='male_townsfolk_hair1_back0'>
   <Colors>
      <Red value='0' />
      <Brown value='1' />
      <Black value='2'/>
      <!-- ... -->
   </Colors>
   <Animations>
    <Animation direction='south' animIdx='walk'
      animationName='male_townsfolk_hair1_forward' />
     <Animation direction='north' animIdx='walk'
      animationName='male_townsfolk_hair1_back' />
     <Animation direction='west' animIdx='walk'
      animationName='male_townsfolk_hair1_left' />
     <Animation direction='east' animIdx='walk'
      animationName='male_townsfolk_hair1_right' />
  </Animations>
 </HairGenerator>
</HairGenerators>
```

***Figure 17**: XML example Hair Generator.*

Names are procedurally generated for each character. In order to accomplish this, name generators and name sets are defined through XML. A name set is key with a bunch of names. *Name generators* contain several name sets, with a defined order to use the name sets, per gender. The name sets, when called by the name generator, then randomly generate the name to use, avoiding using the same name as last time when possible.

```
<NameGenerators>
  <VillagerNameSet
    male='first_male,last'
    female='first_female,last'>
    <first_male>
      <Bill/>
      <Buck/>
      <Phil/>
      <Ralph/>
      <Will/>
    </first_male>
    <first_female>
      <Claire/>
      <Clem/>
      <Jennifer/>
      <Sally/>
      <Sarah/>
      <Sally/>
    </first_female>
    <last>
      <Smith/>
      <Farmer/>
      <Foreman/>
    </last>
  </VillagerNameSet>
</NameGenerators>
```

**Figure 18**: *An example Name Generator in XML.*

Maps can then define what agents and features they wish to spawn from their XML data file. They can also specify a list of interesting positions for the entities when spawning them, as well as the percentage chance to actually use the data, and how many to spawn.

```
<MapData music='Village1'
    constrainCameraBounds='true'>
  <!-- ... -->
  <FeaturesToGenerate>
    <Door numberToSpawn='3'>
      <Position position='7,9'/>
      <Position position='14,1'/>
      <Position position='14,14'/>
    </Door>
    <Firepit>
      <Position position='1,3'/>
    </Firepit>
  </FeaturesToGenerate>
  <AgentsToGenerate>
    <Villager
      number='1' Job='Smith' chance='0.5'/>
    <Villager number='3-5'/>
  </AgentsToGenerate>
  <!-- ... -->
</MapData>
```

**Figure 19**: *XML Map Data for spawning features and agents.*

*Dialogue Segments, Groups, and Choices*

NPCs open *Dialogue Segments* by name when the player attempts to talk to them. A *Dialogue Segment* at its base form is a name with a string of text and an associated Speech Bubble. The text can be indefinetly long and can be defined through an attribute on the dialogue segment node, or through child text XML nodes; all of the text encountered is combined from attribute to top child down to bottom child. Once the full string is read in, the dialogue segment parses each individual word into text pieces. The text pieces will determine if the word is a variable name or just more text; variable names are denoted by the dollar signs on either side of some enclosed text. When the dialogue segment is called by an NPC, it restrings the text pieces together, and replaces the variable calls with any valid values it finds; as an error indication, it will spit the variable name back out if the game is unable to find any text under that variable name. The dialogue segment. while stringing the text pieces together, will also places the strings on separate pages as needed (there is a limit of two lines of text per page).

```
<DialogueSegments
    SpeechBubble='SpeechBubble'>
  <Greeting1
   uniqueGrouping="greeting"
   text="Hello my name is $MyFirstName$.">
    <Triggers>
      <GlobalSetVariableFloat
        Variable="Greeting1Encountered"
        value="1.0"/>
    </Triggers>
  </Greeting1>
  <Greeting2 uniqueGrouping="greeting"
   text="Hi, I'm $MyFirstName$. I am a $Job$."/>
  <LongWindedVillagerGreeting>
    <Text text="Hello, I am $MyFirstName$."/>
    <Text text=" I love my job as a $Job$."/>
    <Text text=" I still wish I could spend more time"/>
    <Text text=" with my family and friends though."/>
  </LongWindedVillagerGreeting>
</DialogueSegments>
```

**Figure 20**: *Several example dialogue segments in XML.*

*Dialogue Groups* contain the names of multiple Dialogue Segments; a dialogue segment can add itself to a dialogue group. The dialogue group handles procedurally generating which segment to call, and will attempt to guarantee that the same dialogue segment as last time does not open unless there is no other option. Additionally, Dialogue Segments can define a set of requirements before a Dialogue Group can open them; e.g. before the dam breaks, the NPCs use joyful greetings when the player speaks to them, but after the dam breaks, the NPCs constantly mention and complain about the dam. Non-existant dialogue groups will be created on demand. However, dialogue groups can also be created through their own XML node, which allows for declarations of group wide dialogue event triggers (these are triggered whenever a dialogue segment in the group is closed).

```
<DialogueGroupings>
  <Greeting>
    <Triggers>
      <GlobalSetVariableFloat
        Variable="GreetingEncountered"
        value="1.0"/>
    </Triggers>
  </Greeting>
</DialogueGroupings>
```

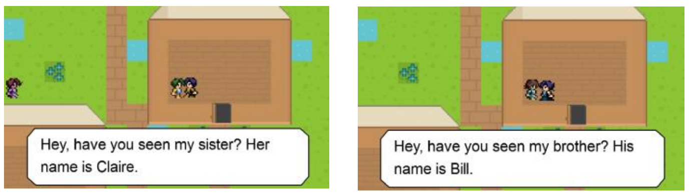
*Figure 21: An example of a Dialogue Group with a trigger.*

Dialogue Segments also contain *Dialogue Choices*. A *Dialogue Choice* is an associated text label along with a series of triggers and settings for if that choice is picked by the Player. The Dialogue Choice menu displays once the player has read all of the text within the regular dialogue segment.

```
<LongDialogueExampleWithChoice>
  <Text
     text="Hey, did you know monsters are coming back?"/>
   <Decision>
      <Yes1
        DialogueToOpen=
         "LongDialogueExampleWithChoiceSelectionYes">
        <Requirements>
          <GlobalCheckVariableFloat
             Variable="Choice1" value="==0.0"/>
        </Requirements>
        <Triggers>
          <GlobalSetVariableFloat
             Variable="Choice1" value="1.0"/>
        </Triggers>
      </Yes1>
      <Yes2 DialogueToOpen=
          "LongDialogueExampleWithChoiceSelectionYes">
        <Requirements>
          <GlobalCheckVariableFloat
             Variable="Choice1" value="==1.0"/>
        </Requirements>
        <Triggers>
          <GlobalSetVariableFloat
             Variable="Choice1" value="0.0"/>
        </Triggers>
      </Yes2>
      <No DialogueToOpen=
           "LongDialogueExampleWithChoiceSelectionNo"/>
      <Maybe DialogueToOpen=
          "LongDialogueExampleWithChoiceSelectionMaybe"/>
    </Decision>
</LongDialogueExampleWithChoice>
```

*Figure 22: An example dialogue segment with dialogue choices in XML.*

Variable names can also be used to access information off of specific quests. The variables can access agent and text variables off a quest. For example, it can get the name of a character's brother or sister, and still change based off the character's gender. For a variable name to know it should get information out of a quest, every unique name needs to be separated by periods. For text quest variables, it needs: the quest name, period, the text variable name. For pulling information off of an agent in a quest, such as the character's name, it needs: the quest name, period, the character's quest variable name, period, "FirstName".



Hey, have you seen my sister? Her name is Claire.

Hey, have you seen my brother? His name is Bill.

*Figure 23: Screenshots of an NPC with a procedurally generated sibling.*

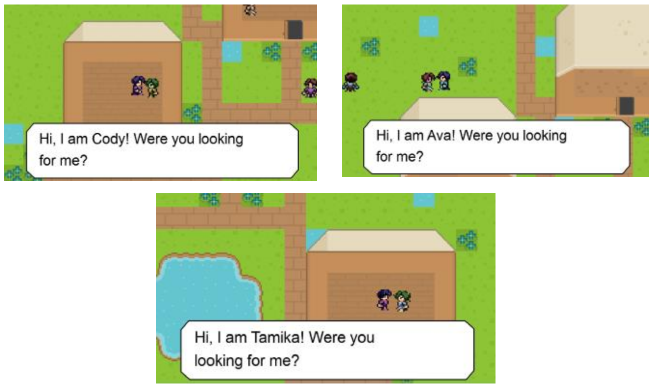
*Procedural Quest Generation*

A procedurally generated *Quest* is complex for the author of said quest to write, but is easy in concept. Variables for the quest need to be defined, especially if any way to access the player is needed. The supported variable types are characters (NPC and player), text for inserting into dialogues, floats, and booleans. Characters can be explicitly defined as an NPC or a player variant. These variables are used later on throughout the life of the quest. After defining the variables, the instructions for spawning Agents and Features are needed. If a pre-existing agent exists that meets the requirements, and is not already claimed by a quest, then the quest will claim it and move it to the intended position. Otherwise the quest will spawn the a new entity with the provided instructions. The variable name for these entities must be assigned. By default, features will always be spawned as new.

The most intricate part of the quest system are the quest events. *Quest Events* require that a set of requirements and triggers be defined for them. An event does not run unless all of its requirements are met. If the events requirements are met, then the event calls of all of its triggers. Both the requirements and triggers use custom code for each variant, and use a self registration system. Quests Events have two lists for triggers, the regular triggers which occur before performing any interaction, and the post triggers which occur afterwards.

Quest Triggers and Requirements can do a fair number of things, such as: changing npc dialogue segments, or setting both local variables to the quest and global variables. As such Quest triggers and requirements both use their own versions of the self registration paradigm.

After a quest is fully written out, the quest can be spawned through a map's data file. On the map data file, an array of positions can be defined for each quest NPC variable; the quest will procedurally generate which one to use and move the NPC to it. This allows for more designer control over the quest. For an example of a quest in XML, see Appendix 2.


Hi, I am Cody! Were you looking for me?

Hi, I am Ava! Were you looking for me?

Hi, I am Tamika! Were you looking for me?

***Figure 24**: The younger sibling from the "Find my Sibling" quest, but at the three different locations.*

A quest agent or feature can be exposed as a global variable; this is so as to give other quests the ability to have access to it and also avoid spawning another NPC for the same intended use. This is done by declaring an attribute of GlobalVariableName on the Agents or Features to generate within the quest; if the attribute is set, it will attempt to pull from the global pool of variables for this entity instead of spawning or claiming an unused entity. This was done so as to allow for basic quest chains, but does not allow for procedurally generating quest chains by itself.

## IV. POST MORTEM

Exploring procedural generation systems was at the core of this thesis' goal, however the intention with the artifact originally was to create a full fledged game instead of a demo. Various existing elements of the artifact proved to take far more time than originally expected; this lead to various planned features (e.g. player skills, and minigames) being cut, as well as the chance to create even more maps and quest examples. However, the systems in this artifact are the essential pieces of the original purpose: to explore procedurally generating content for characters and quests. Thus their funtionallity and level of polish have taken precident over any additional features.

The procedural generation systems for characters and quests are flexible; it is easy to add new code onto the paradigms as needed. Whenever a designer needs a new behavior, requirement or trigger, they need only ask a programmer, and they can create the content within a reasonable short time span. Additionally the use of XML with these systems allows designers to rapidly develop upon quests, character generators or other content.

Most of the development time was spent on setting up the various paradigms as well as the ability to read in XML data. This was critical to have as, although it is slow create the initial code, it makes future work on these systems incredibly easy. Supporting XML also allows designers to focus on the important information, instead of having to surf and alter game code. The second time sink was in parsing the dialogue into individual pieces; the author underestimated the complexity of the system before starting work on it. As such there are parts of it that could be made nicer, such as how it does not use a paradigm to find the variable values. Procedurally generating content was the third time sink; this was mostly due to trying to balance the system so as to avoid the case where the same content is repeatedly generated. E.g. first person spawned is named Sally Moore, second person spawned is named Sally Moore; due to the nature of procedural generation, its possible to have this case occur, especially without additional work to try and prevent this. The author has used a few concepts that should mitigate this case.

What could have mitigated some of the time sink problems would have been to pre-create a bunch of XML files at the start; also, it would have been helpful to draw out the expected class diagrams. The systems are highly polished at this point, but they could have been at that point much sooner in development had there been better planning. One way that could of sped up creating the paradigms would have been to make them all run through a generalized system that handles them all by type; but the author felt that it would loose some level of readability if that step were taken.

## V. CONCLUSION

RPGs can be fun without combat, but they do require additional content to enable them to be fun. Some examples would be making the stories more dramatic, or having minigames to break up the flow. The key appears to be that for a game to be fun, monotony needs to be avoided (i.e. drama increases suspense, minigames break up the flow, etc.). Procedurally generating content aids in how fun a game can be by preventing monotony, aka it guarantees a unique playthrough occurs each time. However, procedural generation can not stand on its own unless the game itself is relatively small; bigger projects will need to have the procedurally generated content affect more elements in the game as well as add in more types of interactions with the world.

The big thing to watch out for when procedurally generating content is feature creep. Expect procedurally generation features to take between 1.5 to 2 times as long as the regular version, based on how complex the generation formula needs to be. The increase in development time required is due to the additional time spent balancing the generation formula. The more elements that can be limited to a smaller range of values, the easier it is to balance the procedural generation for one piece of data. For example, the artifact limits character genders to male and female, and character skin color to what the shader supports (integer values 0 through 6).

## VI. FUTURE WORK

It would be good to add a weighted scale for procedural generation onto what clothes, hair, names, and colors the entities pick; this will make it less likely for the content to use the same set up over and over. The quest chains need to be fully procedurally generatable as well. This would be accomplished by adding in quest existance dependencies as well as the ability for a map to procedurally pick one quest from a handful.

Also, dialogue segments need to be treated more like cutscenes. It would be good to add some additional features to dialogue segment chains. E.g. the developer could add a time interval before the segment opens or apply effects and play sounds. Adding emotion animations to the characters would be a good additional feature for the system; these could be single sprite images that spin, stretch, and move on a timer.

A faction system and a race system would also add quite a lot of diversity to characters in the game. These factions and races would also need some way to define unique words based on definable features about other characters. Having a faction and race system would also pair well with a likeability system; e.g. the player is of the orc race, all human npcs hate orcs, so they will refuse to give the player information or progress quests until the player becomes more likeable.

The artifact presently does not have a real inventory or item system. The author would create one such that the player could navigate through menus in order to equip clothing or other wise. The author would also add more item types, which could make way for a crafting system. The crafting system that the author prefers would use ingredients from the players inventory or the score the player achieves in minigames. These minigames could also add onto quest functionallity; e.g. because the player cooked a high quality fish for an NPC, the NPC will tell the player everything they want to know.

The author can forsee a need for a development tool when creating the XML data. This is as the amount of data expands, it can quickly become harder to keep track of the names of various bits of data. E.g. the number of dialogue segments could grow incredibly rapidly, thus it would be easy to lose track of what names have already been used per dialogue segment; a development tool would mitigate the issue. The development tool would merily verify that a given data type has not already used a particular name, and spit out some sort of error message when it happens. It would also need to be able to load and save out to XML.

## VII. References

[1] *Mabinogi (PC),* Nexon Korea Corp., 2016.

[2] *Runescape (PC),* Science Park: Jagex Ltd., 1999-2016.

[3] R. v. d. Linden, R. Lopes and R. Bidarra, "Procedural Generation of Dungeons," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 6, no. 1, pp. 78-89, 2014.

[4] D. Fitter, *Audiosurf,* Valve Corporation, Ascaron, 2008.

[5] *Middle Earth: Shadow of Mordor (PC),* Warner Bros Interactive Entertainment, 2014.

[6] *Crusader Kings II (PC),* Paradox Interactive, 2012.

[7] J. Ryan, B. Samuel, A. Summerville, M. Mateas, N. Wardrip-Fruin and T. Brothers, "Bad News," Studio, Expressive Intelligence, 2017. [Online]. Available: https://www.badnewsgame.com/. [Accessed 26th April 2017].

[8] *FTL: Faster Than Light (PC),* Subset Games, 2012.

[9] S. Eiserloh, J. Forbes, J. Hamel, T. Roberson, M. Sellers, L. Law, W. Emigh, J. Grinblat, R. Holmes, I. Schreiber and S. Swink, "Group Report: Generative Systems, Meaningful Cores," Project Horseshoe, 2015.

[10] M. Mateas and A. Stern, *Facade,* Procedural Arts, 2005.

[11] R. Evans and E. Short, "Versu - A Simulationist Storytelling System," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 6, no. 2, pp. 113-130, June 2014.

[12] Y.-G. Cheong, M. Riedl, B.-C. Bae and M. Nelson, "Chapter 7: Planning with Applications to Quests and Story," in *Procedural Content Generation in Games*, Springer, 2016, pp. 119-137.

[13] K. Hartsook, A. Zook, S. Das and M. O. Riedl, "Toward Supporting Stories with Procedurally Generated Game Worlds," in *2011 IEEE Conference on Computational Intelligence and Games*, 2011.

## VIII. Appendix

### Appendix 1: "Drylands Village" Map Data XML

```xml
<MapData Music='Village1' ConstrainCameraBounds='true'>
  <TileEvents>
    <TileEvent TileCoords='0,39'>
      <OnEnter Function='ChangeMaps' MapFile='Drylands'
          PlayerPosition='43,39'/>
    </TileEvent>
    <TileEvent TileCoords='0,38'>
      <OnEnter Function='ChangeMaps' MapFile='Drylands'
          PlayerPosition='43,38'/>
    </TileEvent>
  </TileEvents>
  <AgentsToGenerate>
    <Human number='7-9'>
      <Position positionRange='2,2,41,37' />
    </Human>
  </AgentsToGenerate>
  <FeaturesToGenerate>
    <Door number='5'>
      <Position position='34,25' />
      <Position position='28,15' />
      <Position position='28,9' />
      <!-- ... -->
    </Door>
    <RightDoor number='7'>
      <Position position='16,14' />
      <Position position='39,15' />
      <Position position='39,9' />
      <!-- ... -->
    </RightDoor>
  </FeaturesToGenerate>
  <Quests>
    <Quest Title='RepeatableDeliveryQuest'
         PercentageChanceToOccur='1.0'>
      <DrylandsVillage>
        <Positions forEntity='Foreman'>
          <Position position='20,16' />
          <Position position='9,26' />
          <Position position='17,27' />
          <Position position='14,22' />
          <Position position='15,25' />
        </Positions>
        <Positions forEntity='Baker'>
          <Position positionRange='30,26,36,28' />
        </Positions>
      </DrylandsVillage>
    </Quest>
  </Quests>
</MapData>
```

### Appendix 2: "Find My Sibling" XML

```xml
<Quests>
 <FindMySibling title='Find My Sibling'>
  <Variables>
   <NPC VariableName='OlderSibling'/>
   <NPC VariableName='YoungerSibling'/>
   <Player VariableName='player'/>
   <Text VariableName='QuestItem1' value='Staff'/>
   <Float VariableName='QuestState' defaultValue='0.0' />
  </Variables>
  <AgentsToGenerate>
   <Human number='1' Job='Smith' AlwaysSpawn='true'
     VariableName='OlderSibling'
     GlobalVariableName='OlderSibling'
     Dialogue='FindMySiblingStart'>
   <NPCAttributes>
     <Age>
       <WhiteList>
         <Young_Adult />
       </WhiteList>
     </Age>
    </NPCAttributes>
   </Human>
   <Human number='1' Job='Smith' AlwaysSpawn='true'
    VariableName='YoungerSibling'
    GlobalVariableName='YoungerSibling'
    Dialogue='SiblingSpeechSpawnText'>
    <NPCAttributes>
     <Age>
      <WhiteList>
       <Teenager />
      </WhiteList>
     </Age>
    </NPCAttributes>
  </Human>
 </AgentsToGenerate>
 <Events>
  <Event EventName='Start'>
   <Requirements>
    <Requirement RequirementType='CheckVariableFloat'
       Variable='QuestState' value='==0.0'/>
    <Requirement RequirementType='CheckPlayerInteraction'
       Interact='OlderSibling'/>
   </Requirements>
   <Trigger>
    <SetQuestVariableFloat Variable='QuestState'
       value='1.0'/>
    <ChangeDialogue Entity='YoungerSibling'
       DialogueSegment='SiblingSpeech'/>
    </Trigger>
   </Event>
  <!-- ... -->
  </Events>
 </FindMySibling>
</Quests>
```