Role Playing Game AI System

CS4624
Multimedia, Hypertext, and Information Access

Written By

Kurt Grzybowski
Jason Lao
Devin Toms
Daniel Sabanov

Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Instructor: Dr. Edward A. Fox
Client: Prof. Zach Duer
May 10, 2023

Contents

List of Figures

List of Tables

Chapter 1

Abstract

For this project our team was tasked to create an AI for an immersive role playing game that contains a modular ability system. The AI must be capable of interacting with the ability system effectively. The AI must be fair, and as such will be constrained by the same constraints as the players. It must have the same abilities, follow the same rules, and only operate on information that is also available to the players. Additionally, the AI must be capable of being an adversary or an ally to the player. We have implemented a basic routine-based approach to solve the problem. This approach uses a "Black Box" function to deduce a general course of action to take, such as an attack, defense, or heal. Then, the routine decides on the best approach to achieve its goal and adds specific actions to an action queue. Additionally, we have added ability tags that the AI uses in order to understand what purpose each ability serves. With the use of such an approach, we hope to achieve an AI that is both challenging and is immersive to play with or against. We performed manual testing by inserting our AI into various pre-defined scenarios. We have also documented our code with comments that explain the functionality of our code. Our deliverable content is mainly contained within the files `BlackBox.cs` and `State.cs` with small additions in various other files. As the game is still in the process of development, much of its functionality is still incomplete. As such, we had to work around some missing functionality, such as incomplete character classes and abilities, by making some assumptions about the final vision of the client. Therefore, the Black Box and the routines will need to be modified as new functionality is introduced into the game in order to better make use of that new functionality.

Chapter 2

Introduction

2.1 Client

Our client, Professor Zach Duer, is an Assistant Professor at the Virginia Tech School of Visual Arts [2]. Professor Duer works with sound, intuitive music spontaneity, careful composition, improvised performance, visualization and digital systems [1]. He is also very well versed in Unity and programming in general.

2.2 Background

Professor Duer has been working on developing a role playing game for over a year and has built a very diverse, modular ability system within in. The game is part of a research project into how players form and mirror social roles based on their virtual avatars. As such, the game needs to be as immersive as possible in order to draw the player in more effectively. This also raises a few constraints on the AI implementation, as the AI needs to be engaging and fair. As such, the AI cannot have access to any information that the player cannot access, and must operate by the same rules as the player, while at the same time still remaining challenging.

2.3 Objective

Our project aims to create an artificial intelligence (AI) to control the in-game entities that are not controlled by a human player (known as “non-player characters” or NPCs). To be considered successful our AI needs to accomplish the following goals:

- It must be operational.
- It must be capable of using all of the currently implemented abilities.

- It must be difficult to fight against.
- It must be capable of distinguishing friend and foe.

2.4 Deliverables

Our deliverables are the scripts that power the non-player, in essence the AI, characters, and documentation on how to interact with said scripts.

Chapter 3

Requirements

Prof. Duer's game has been in development for over a year as of the writing of this document, and as such there are clearly-defined game design elements that must be followed to ensure that the eventual finished production is cohesive. To this end, Prof. Duer has created a game design document (located in Appendix C) which details the specifics of how developed systems should interact at a high level [3]. We obtained from him this document so that we may more effectively carry out his visions for the design of the AI, as there are a number of game design elements that must be taken into account by the AI in order to ensure that the playing experience is consistent. Listed below are excerpts from this document which define each of these elements:

1. "Abilities are non-abstract. Wherever possible, all interactions with the world should be 'What You See Is What You Get'."
2. "Character abilities and properties should be directly diegetically represented wherever possible."
3. "Abilities should focus on enabling both the character and player to fulfill a social role in relationship to other players, stimulating meaningful interaction."
4. "Social roles should be well-defined and immediately apparent."
5. "Player power should mirror character power."
6. "The interaction system (including the combat system) should be structured to necessitate the improvisational use of abilities with medium-duration or long-duration player decision time."
7. "Fulfillment of social role is more important than balance."

Chapter 4

Design

4.1 System Overview

Based on Prof. Duer's requirements as laid out in Chapter 3, we identified that the AI should:

1. have the same access to information as the player,
2. be able to adapt to new abilities being added with minimal modifications,
3. make optimal decisions in a dynamic context, and
4. give off a feeling of true understanding of how abilities interact.

To this end, we created an outline for our AI which we felt would allow for a system in accordance to these guidelines. Figure 4.1 shows this outline of our design for the overall flow of information through our system. The circles represent information that we want to take into account which will be passed into the Black Box, while the triangles represent the different routines the Black Box returns to denote the objective that has been decided for the AI. These routines control how abilities are selected by the AI to be used. They are denoted by their high level function.

4.2 The "Black Box"

Figure 4.2 labels what type of information we will specifically be looking for from each of the specified input sources. All of these inputs can be broken down into three different categories: self, other entities, and environment. The main focus in the Black Box is the other entities, which can either be allies or enemies – these input sources tend to be the most variable and relevant sources when deciding what actions we want to make.

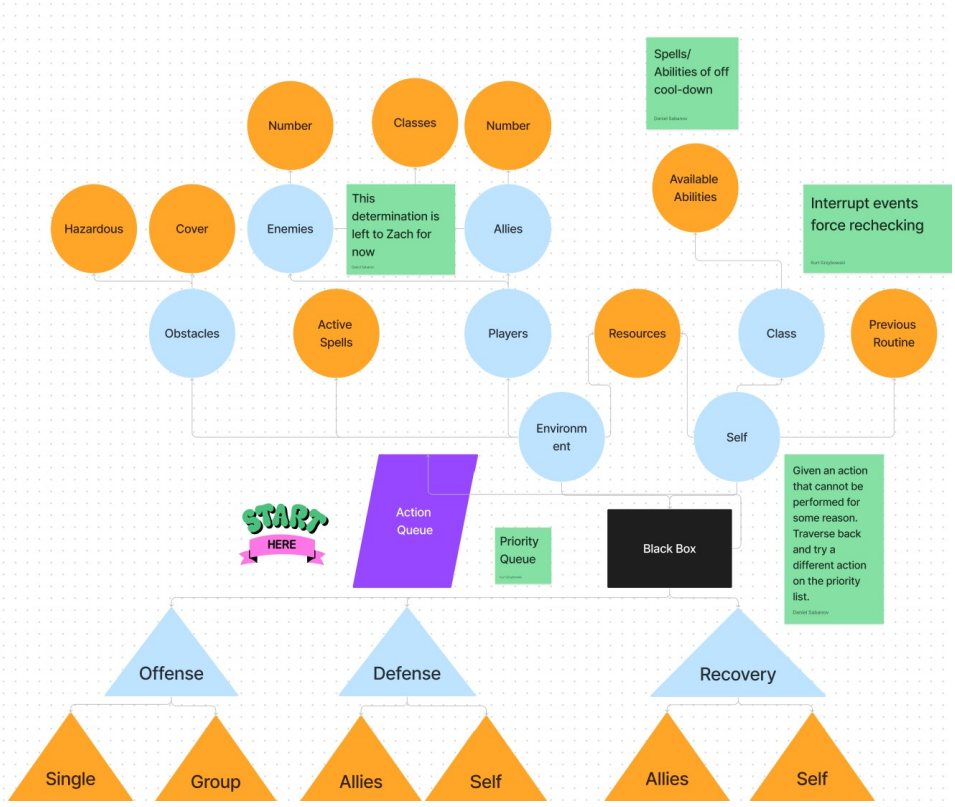

Figure 4.1: Full Outline of the AI System

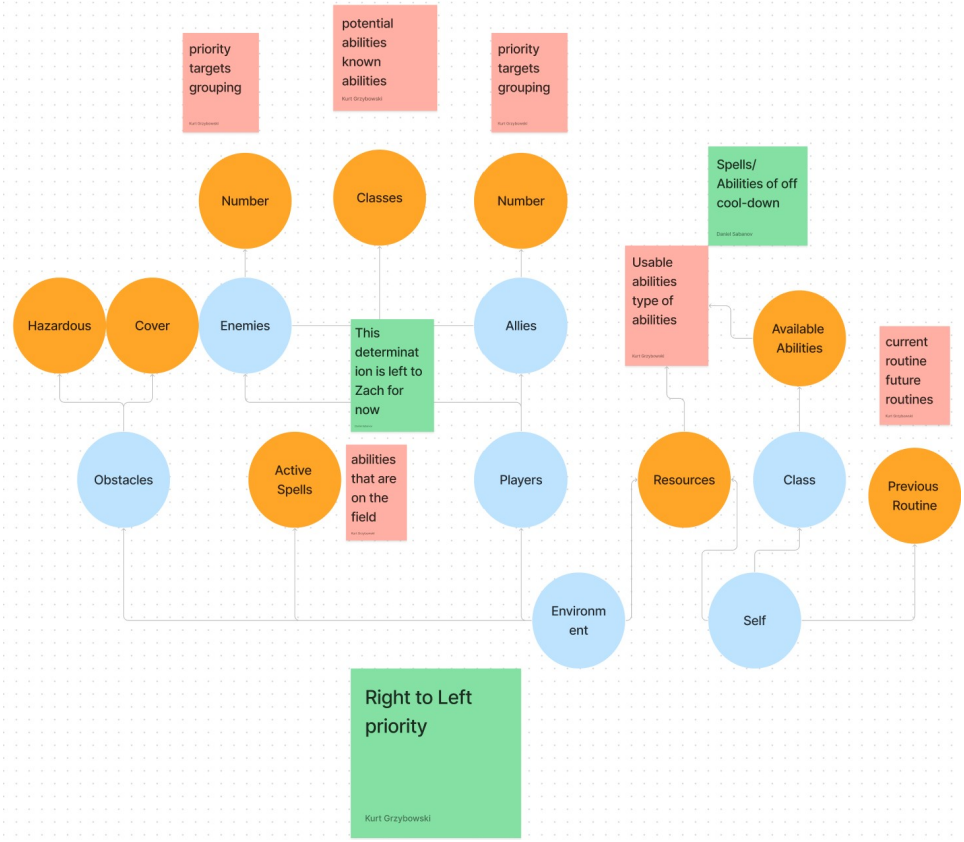

Figure 4.2: Input Outline

Figures 4.3 and 4.4 show a basic outline of how we will transverse the information that we gather. They also show the larger decision making ideas that we want our AI to be able to make. The gray boxes represent when the Black Box would output a decision. The first checks make sure that any current routines being run are still desirable. If not, then it continues down the decision tree. It next checks for active elements in the environment to see if any are addressable. If there are, then a routine to do so is set, otherwise it moves onto the main part of the decision process. Then, depending on if the character is more aggressive or passive (i.e., checking for offensive possibilities or defensive ones, respectively), it checks to find valuable groupings (such as those with low health) as possible high priority targets. If no obvious priority targets are found, then based on aggressiveness, a suitable target will be picked at near random.

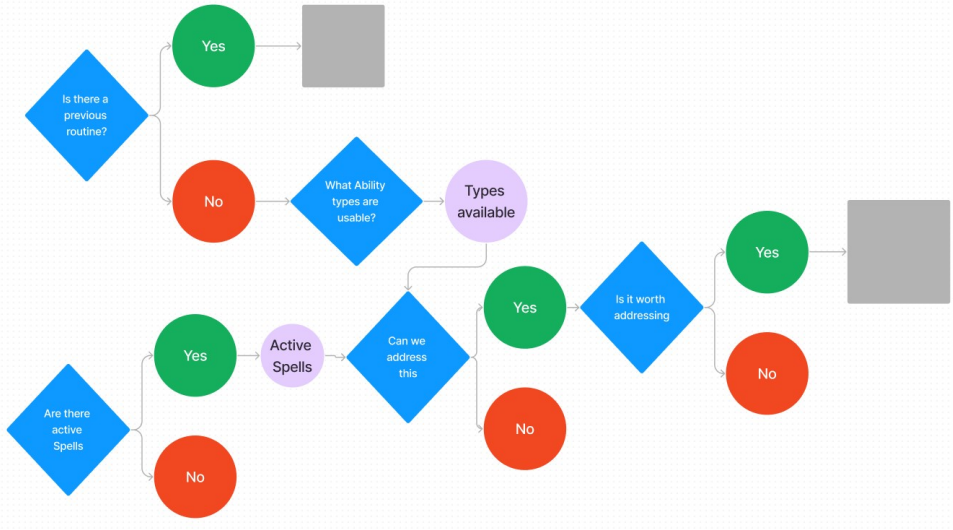

Figure 4.3: Black Box Flow Chart Part 1

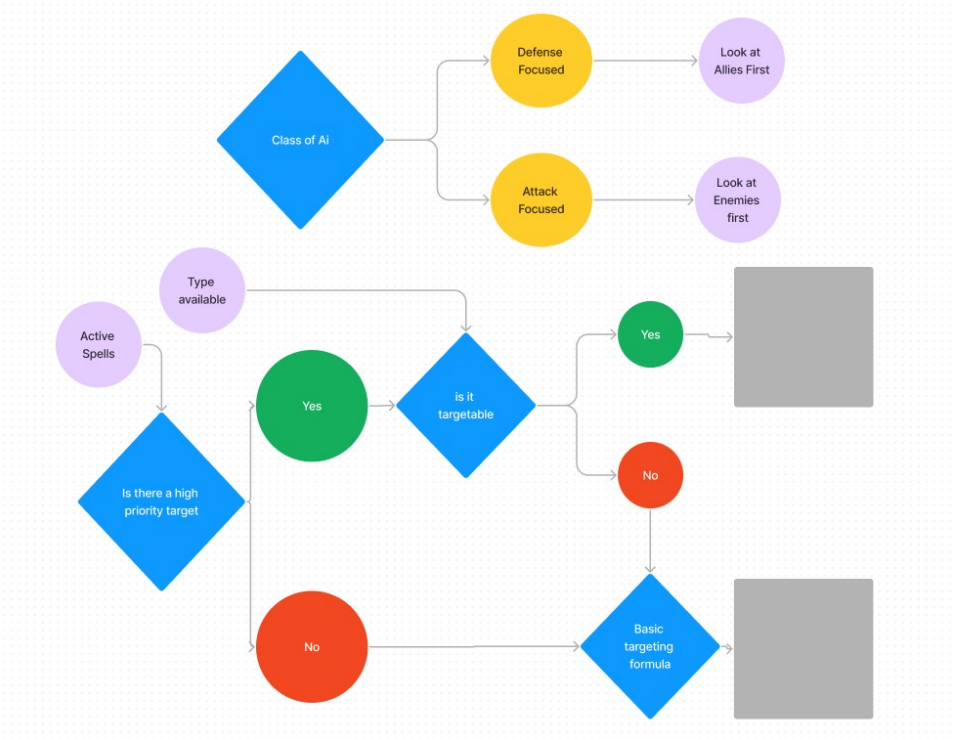

Figure 4.4: Black Box Flow Chart Part 2

4.3 The "Routines System"

During the process of drafting our initial designs, we came to the understanding that simply collecting data about the surrounding environment would not be enough for the AI to make any proper decisions. That is, while the AI would possess an understanding of what is happening around it, it would have no knowledge of how to act on that information with any level of optimality. Based on the system requirements, as well as our understanding of the current system, we identified three key problems which we set out to solve:

1. The AI has no understanding of the effects of its abilities.
2. The AI has no way of understanding which abilities are best for a given situation given their effects.
3. The AI has no way of "queuing" multiple abilities, thereby preventing it from being able to use complimentary abilities in sequence.

4.3.1 Ability Tags

The first of these issues has a rather straightforward solution – if the AI is unable to understand the effects of abilities, we must devise a method of associating effects with abilities in a computer-readable manner. After a relatively short period of deliberation, this manifested itself as the concept of "ability tags". Each tag represents a "signifier" – a representation of a specific aspect of an ability's deliverables.

There are two types of ability tags – "effect type" tags, which signify the specific attributes/effects of an ability's deliverables, and "range type" tags, which signify the range of an ability's deliverables. Abilities can be associated with up to N unique effect type tags (where N is the number of unique effect type tags) and a single range type tag. An example of how this works can be seen in Figure 7.1 – the "Sunburn" ability (whose human-readable description can be found in the green "Tooltip" section) can be cast from a range of 20 meters, which would associate it with the range type tag `RANGED`. In addition, it is associated with the effect type tag `DAMAGE TARGET` because its primary deliverable applies damage to its target. A full list of each ability tag signifier, as well as their corresponding descriptions, can be found in Appendix B.

An unfortunate side effect of this solution is an additional burden placed upon designers when creating a new ability, as they will now be required to define ability tags in addition to all previously required work. We feel that this is a worthwhile trade-off, however, as the additional work required of designers is rather low given the envisioned specificity of ability tags.

4.3.2 Routines

Unlike the rather straightforward solution of ability tags, equipping the AI with the ability to determine which abilities are best for the given situation given their effects requires a more complex approach. Early on in the design phase, many different approaches were considered, such as more traditional logic-based algorithms and the use of machine learning (ML) models. We felt that each approach had pros and cons, with traditional logic-based algorithms being fast and low-cost but inflexible, and ML model-based approaches being much more adaptive but requiring a vast amount more time spent training and fine-tuning the model. Due to the complexities of such a system and the current state of games being such that many systems have not been implemented yet, we decided to focus our attention on creating a more modular system that would allow us (given the time) or a future team to implement a more sophisticated decision algorithm without having to worry about the rest of the AI implementation.

In creating this system, we took inspiration from the "Gambit System" in the video game *Final Fantasy XII* [4]. In this game, players are able to make use of a series of programmable "if-else" type statements to automatically direct AI-controlled characters in battle [5]. While the complexity of such a system would result in a burden of implementation far beyond the scope of this project, we felt that we could develop a similar system by using a pre-determined set of known "good" abilities based on the current situation. This led to the creation of "routines" – predetermined sets of "good" abilities and corresponding movement behavior based on the given situation. While not programmable or modular, our goal was to develop something which could still at least somewhat realistically replicate the human decision-making process.

During our design process, we identified three primary combat-oriented "categories" under which decisions can be made: attacking, defending, and healing. Attacking routines are primarily focused around trying to eliminate hostile entities using damage-dealing abilities and inflicting them with harmful status effects. Defending routines are focused on protecting both oneself and/or allied entities from damage and other harmful effects. Finally, healing routines focus on using abilities which "restore" oneself and/or allied entities from inflicted damage and harmful status effects. The determination of which routine should be followed is made by the Black Box, and the chosen routine works in tandem with ability tags to create a list of "good" abilities based on what the AI has access to. More information on how this works can be found in Chapter 5.

4.3.3 Ability Queue

Finally, in order to create an even semi-realistic AI, the system must be able to "queue" multiple abilities for the purpose of using complimentary abilities in succession. For example, if the AI were to use an ability which supplies them with an additional resource, the system should queue abilities in such a way that an ability which consumes that newly-created resource can be queued ahead of time. We once again pulled from *Final Fantasy XII*'s Gambit System, though rather than trying to replicate a portion of the system we tried to replicate a portion of its behavior. That is, although *Final Fantasy XII* does not queue abilities, instead allowing for complimentary ability combinations through clever programming by the player, we hope to achieve a similar effect by programming our AI to recognize complimentary ability combinations (via a routine) and then queue those abilities in succession.

This intended effect led to the creation of the "ability queue" – a queue associated with the AI which stores a list of abilities to use when ready. The ability queue can only hold up to three abilities at a time, limited primarily to allow the AI to reconsider the situation frequently rather than only reconsidering when interrupted. While the ability queue is not required to be a priority queue, doing so adds a level of granularity and allows for further optimization of ability selection.

Chapter 5

Implementation

5.1 Ability Tags

Ability tags served as a rather simple solution within the larger system. With the modular-ability system already completed, and ability tags relevant to the scope of an ability's definition rather than any specific run-time instance, we edited the already extant `Ability` class' properties to include data structures to hold range type and effect type signifiers. This also provides designers with a convenient interface which they can use to modify AI functionality. Effect type and range type signifiers are defined as enumerated variable types `EffectType` and `RangeType`, respectively. Abilities can be assigned up to one `RangeType` tag and an array of any number of `EffectType` tags (though the user is warned of the consequences of adding more than one of the same tag to comply with design constraints). These are assigned to the ability as public variables which can be accessed and edited in the Unity Inspector, which can be seen in Figure 7.1. Work was done to add ability tags to abilities currently in the game with human-readable descriptions, regardless of their implementation, though as mentioned within Section 4.3.1, future abilities will require manual tag assignments by the designer.

5.2 Routines

Routines are an area that underwent some major deviations from our initial design. When first designing the system, we envisioned an `AIManager` class, which would function as the name describes, actively listening for Black Box signals and deciding on which routine would be best. Under this system, routines would have been self-contained and partially-modular algorithms, that ranked available abilities based on their likelihood to produce favorable results. This would have likely used some sort of machine-learning model. However, due to time constraints and complications with how the Unity engine handles data sharing, we decided to scale back and alter the framework

to fit within the pre-existing state-based AI model.

Within this new model, the AI enters a `PickRoutine` state, during which it calls a function of the Black Box to receive information on which routine should be taken next. Further specification of this functionality and of how the Black Box works under this model can be found in Section 5.4. Once a recommended routine is acquired, the AI then transitions to a corresponding movement-related state (e.g., if the recommended state `ATTACK` is given, then the AI transitions to the `Pursue` state, which causes the AI to chase after the primary target) to carry out the chosen abilities in the queue. Due to time limitations, the ability selection processes of routines do not perform any sort of optimization to determine which of the usable abilities provides the highest return. Despite this, the area in the code where this function would be accessed has been marked in both the code and in the discussion in Chapter 8. The addition of such an algorithm for each routine would significantly improve performance of the AI, and is also the point at which a ML model could be utilized if desired.

5.3 Ability Queue

Similarly to ability tags, the ability queue was relatively untouched in the system design changes and is thus a rather straightforward implementation of the initial design. One small consolation had to be made in regards to the actual data structure used – due to the structure of how routines were altered to enable the creation of a prioritization function, the ability queue is represented by a `Queue` rather than a `PriorityQueue`, as using the latter would be redundant. This is still permitted within the original design specifications, however, and so does not pose an issue at this time.

5.4 The "Black Box"

All the work that the Black Box needs to do is handled in a single file called `BlackBox.cs`. When it is deemed that a routine needs to be picked, the Black Box function `NextRoutine` is called. This function takes in no parameters, and determines what routine would be best from the input sources it takes from. The sources we called from to make the decisions are: all resources of the player, all other players' health within a certain range, all abilities that the player can currently cast, and the abilities currently in the environment. These input sources are all used to create scores that help decide what routine to pick. After the decision is made, the function returns a `RoutineInformation` struct. This struct contains enums for State, Range, Urgency, and Quantity targeted, as well as the target entity. The State enum defines the category that the desired action falls under: defend, attack, or heal. The Range enum defines a rough idea of how far away the target is.

To help eliminate which abilities are possible, the Urgency enum defines how much priority should be given, and when, to the preservation of resources when choosing abilities. The Quantity enum defines whether the ability chosen should be an AOE (area-of-effect) or single target ability. The target is simply the entity controller that corresponds to the chosen target.

Chapter 6

Assessment

As our AI will be used within the context of a video game, we primarily tested our implementation through manual play testing. Professor Duer gave us access to scripts that we could use to build and create various scenarios to test our project. Using these tools, we could control the number of enemies and their positions, helping us to test Black Box factors such as ability range, radius, and damage. Some scenarios tested included a number of closely grouped enemy entities (to test AOE attack abilities), damaged entities (to test target prioritization), and a moving enemy entity (to test decision making over an extended period). Additional tools like the game's user interface (UI) elements and Unity's built-in console were also be used to see how AI instances are interacting at a more fine-grained level.

For our testing we found that our attacking logic works as intended. It correctly locates high priority targets first where possible, otherwise targeting the first available enemy with the lowest health.

The first problem we came across in testing is switching from a attack routine to a healing routine. The healing logic works. However, when the AI switches routines, some information is lost, which prevents the correct behavior from playing out. The point at which data is lost is unknown for now and will have to be resolved as future work.

Another issue that we found and were not able to address is an AI's distancing. It will continue to close in, until in range for the next attack, and never back off. This leads to the NPC continuously attempting to move towards its target.

Additionally, after all enemies have perished, the AIs go into the patrol state since there are no enemy targets in sight. This prevents the NPC's from performing healing abilities without the presence of a hostile entity, which is slightly unfavorable behavior. A possible solution would be to have the AI exit the patrol state when there are missing resources in addition to enemies, so that the NPCs will heal themselves after a fight is completed.

Chapter 7

User Manual

Designers should primarily interact with the AI behavior by changing tags assigned to abilities. Each ability has a separate list of tags, corresponding to various information, or "signifiers", about the ability that the AI takes into consideration when interacting with the environment. Signifiers define traits which affect how the ability is used, such as whether it is close or far-ranged (denoted by "range type" signifiers), and certain aspects of an ability's deliverables (denoted "effect type" signifiers). More information on these signifiers can be found in Appendix B. Only one range type tag may be associated with an ability. Conversely, up to N effect type tags can be applied to an ability (where N is the number of unique effect type tags), though these tags should not be listed more than once – doing so may cause the AI to over-prioritize the ability in its decision-making.

The simplest way of changing the tags assigned to an ability is through the Unity Inspector, as shown in Figure 7.1. Note the "Ability Tags" section. Within this box group is a list of the tags currently applied to an ability. Clicking the + button allows for the addition of a new tag, and clicking on any of the tags within the list will bring up a list of all programmed signifiers, of which selecting one will update that assigned tag. This is shown in Figure 7.2.

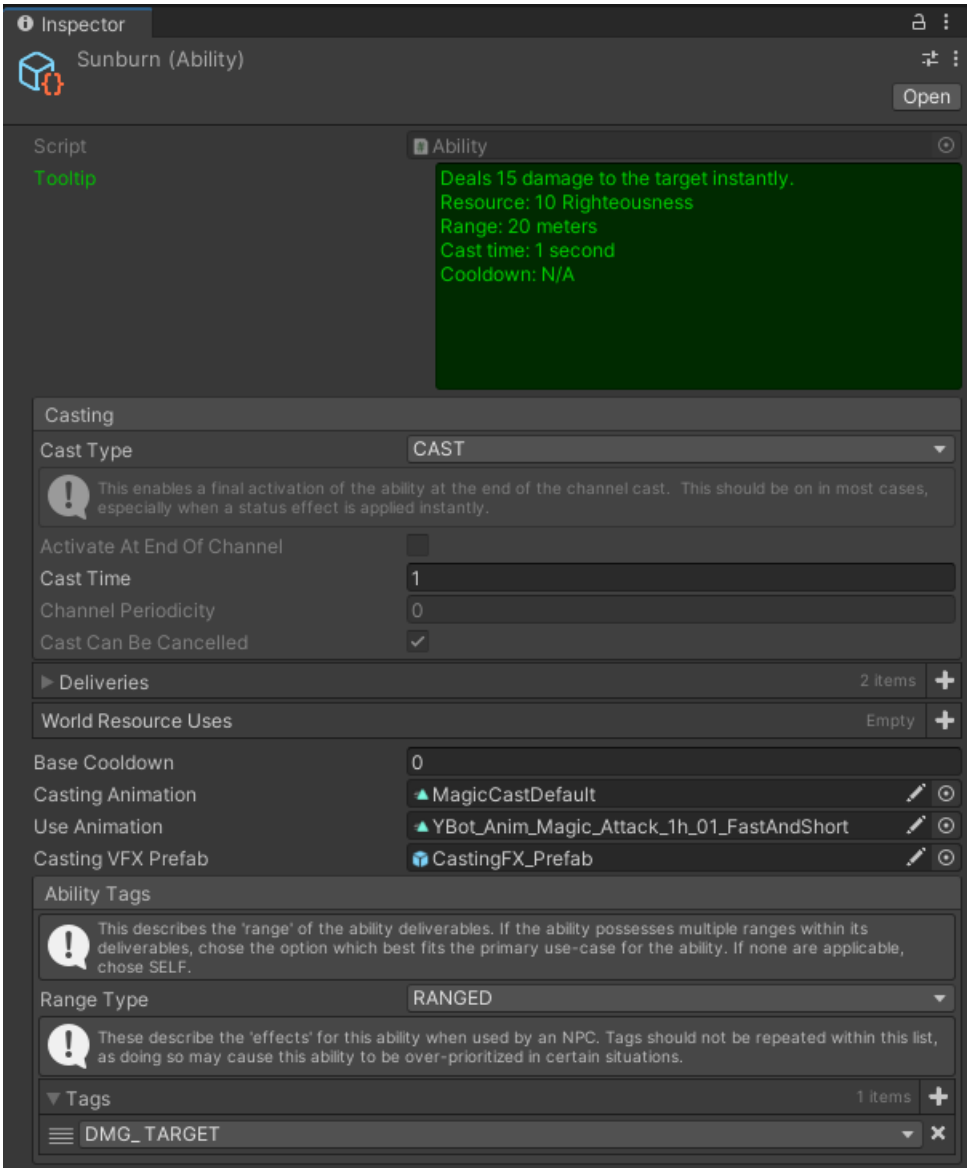

Figure 7.1: “Ability Tags” Section of Unity Inspector

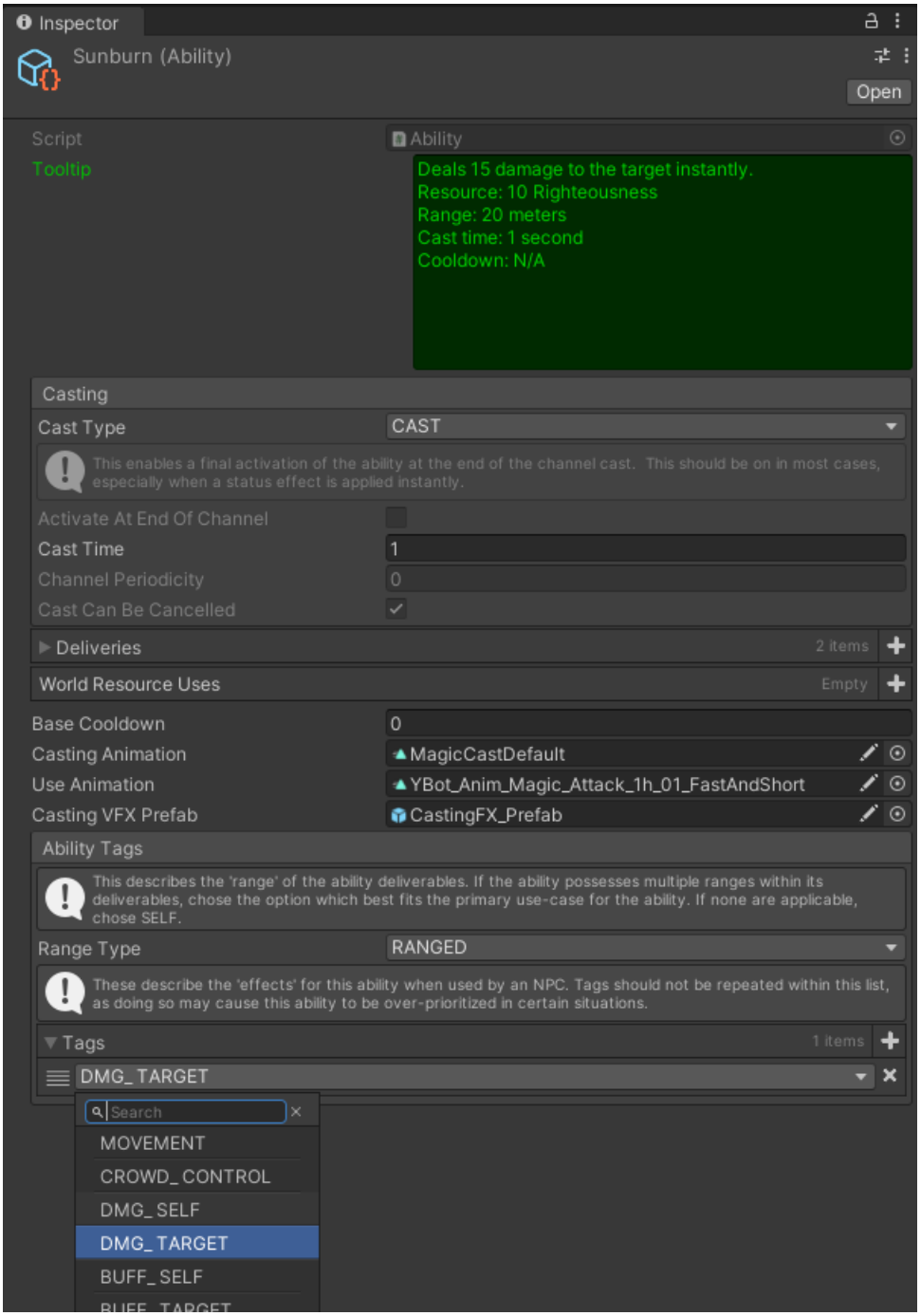

Figure 7.2: Adding an effect type ability tag to the “Sunburn” Ability

Chapter 8

Developer Manual

8.1 Project Structure/Installation

The project has been implemented as a series of scripts within the existing Unity project.

8.2 Ability Tags

Ability tags are a key component of the AI system, as they allow for the AI to be able to consider an ability's behavior when determining the appropriate routine to execute (see the "For Designers" section of the User Manual for more details on what attributes these tags should signify). They can be accessed within the `Ability.cs` file. This file contains an enumerated variable `UseCaseType`, of which each variable corresponds to a signifier. These tags are stored within the class as an array of `UseCaseTypes`, which is accessible within the Unity Inspector under the "Ability Tags" box group.

8.3 State.cs

`State.cs` is the point at which our systems intersect with the the main game logic. This class defines the AI logic, causing the in-game entity to act upon elements of the game world. The eponymous `State` class serves primarily as an abstract class, containing within it inner classes which implement a specific step in the decision-making/action process (and will be from here on referenced when the term "state" is used). NPCs are loaded into the game world in an `Idle` or `Patrol` state, which both look for a potential target within sight range. Upon spotting a target, they then enter into the `PickRoutine` state, which queries the Black Box to determine the recommended next routine and stores that information within the entity's `entityController`. It then transitions into the `Pursue` state to approach the target until in ability range. Meanwhile it should call the `BuildAbilityQueue()` function, which does the

work of building the ability queue. Upon arriving at the target, the AI enters the `UseAbility` state, which handles casting the ability and performing ability deliverables. In addition to `Pursue`, the `Hold` state (which causes the AI to stand still until the target is in range) and the `Run` state (which causes the AI to move away from the target) have been implemented, though they are not used. Determinations to transition into these states can be added to the current logic within `PickRoutine` if they become relevant in the future.

8.4 BlackBox.cs

The `BlackBox.cs` file holds the Black Box structure of our AI implementation. The Black Box as a structure is responsible for evaluating the in-game situation and the environment surrounding the entity. Based on these observations, the Black Box decides on the general approach that the actor should take, with those currently implemented being attacking, defending, and healing. The Black Box summarizes the information that is crucial for routine selection in a structure named `RoutineInformation`. `RoutineInformation` contains the name of the next routine that the AI should take on, the target which the AI should prioritize, the distance to the target, and whether the AI should use area-of-effect spells on the target or single-target spells. The structure also contains a field named `URGENCY`. However this field has not been fully implemented yet, and is added as a possible avenue for further development.

Chapter 9

Lessons Learned

9.1 Timeline

A visual reference for our timeline can be found in Figure 9.1. At the start of the project, we held multiple planning sessions, familiarizing ourselves with the current state of the project, brushing up with the C# language, and brainstorming possible system designs. Many of these sessions included the assistance of Prof. Duer, who provided us with both documentation and live demos to speed the acclimation process.

Throughout March our main goal was to determine an approach that would best accomplish our given task. We worked on finalizing our design to make the implementation process go smoother.

In early April, shortly after the commencement of our implementation process, we begin to realize that the scope of our design had outpaced the time constraints we were placed under. As a result, we held a "Mid-Project Correction" meeting to reassess what we could reasonably complete with our remaining allotted time. A detailed description of the changes made as a result of this reassessment can be found within Chapter 5.

In May we cleaned up the issues that were revealed to us through our testing, and cleaned up the files we had worked on, including documenting all that we have done.

9.2 Problems and Solutions

The biggest problem we came across was finding where and how to get certain information from the code base. There really isn't a perfect solution to this problem. It was more about finding and using as many different strategies as possible to more efficiently find whatever we were looking for. The simplest solution was to ask our client. However, not only can this be slow due to the need to wait for a response, but it also is unprofessional to run to our client at every single roadblock. The most intuitive solution is to just learn the code

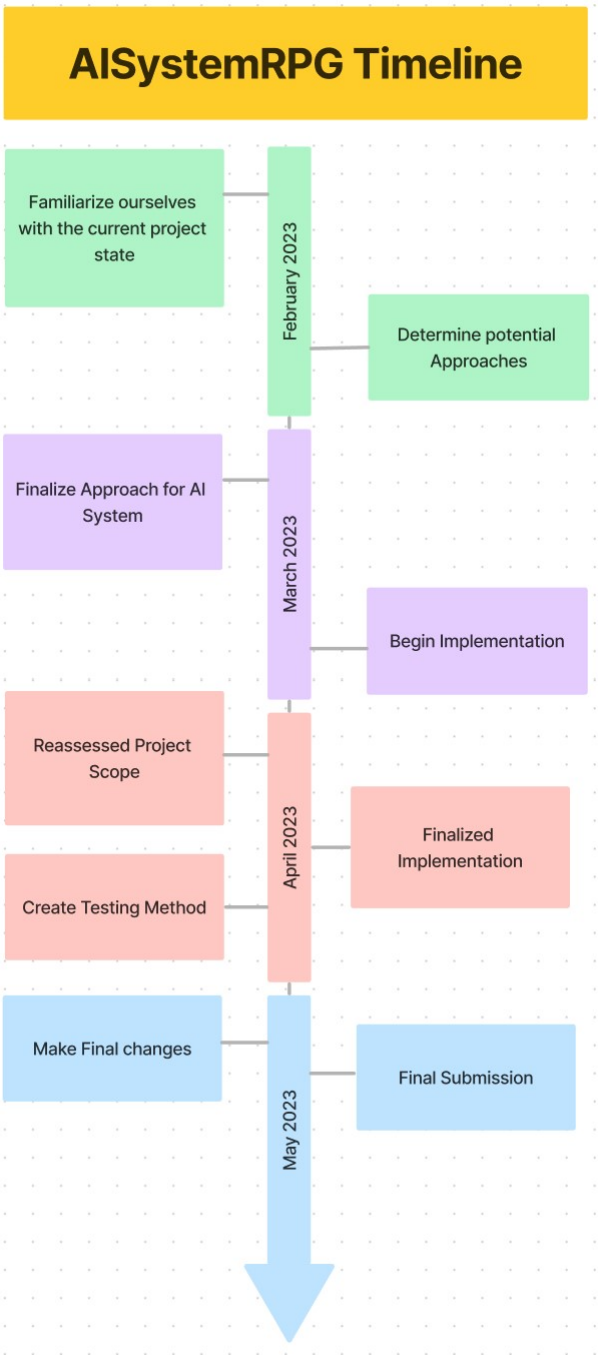

Figure 9.1: Project Timeline

base, but since it is so large, learning the entire code base is just not a plausible reality. But we were still able to slowly build a better understanding of the organization as we explored it more every time we had to find something within the code base. That let us be quicker at finding something the next time. Other useful solutions included finding other areas in the code that also had to call upon the same information that we needed, or using built in tools like ctrl-f to search the entire code base if we knew specific names relating to what we were looking for.

9.3 Future Work

While planning how we were going to execute our task, we quickly realized that we were not able to fully satisfy the wants of our client in the short time that we had. Accordingly, we wanted to construct our solution in a way that would easily allow others to come in and either add-on to our solution, or even completely change certain aspects without having to restructure too much of the code to refit it. This was mainly achieved by creating the firm separation between the Black Box and the routines. The single point of integration is the output given by the Black Box, which has been made easy to change, as it is in its own struct.

The initial future work we would want to complete are the fixes described in Chapter 6, which defines problems that were discovered in testing.

The other future work for the project include ways to add more input to the Black Box, so that the AI can exhibit even more adaptability to situations. We also feel that adding more machine learning components to both the Black Box side and the routines side could enhance the complexity of the AI's behavior. This would help make the decision making elements of the AI more robust and help optimize ability selection, thus better achieving the goal for this project.

Additionally, there are features that we haven't fully implemented from the original design, such as the Black Box's first check to ensure the current routine, if there is one, is still the preferred choice. This has not been implemented, as we had pushed back the implementation of routine interruption so that we could focus on making the routines more robust. Therefore, the ability queue currently must be emptied before the Black Box is called again.

Acknowledgements

We would like to thank our client, Professor Zach Duer, for proposing this project and providing us with the freedom and guidance to create a system which aligns with his vision for the game. In addition, we would like to thank Dr. Edward Fox for his assistance and feedback while we work on this project. Their contact information is as follows:

- Client: Zach Duer
  zachduer@vt.edu
- Instructor: Dr. Edward Fox
  fox@vt.edu

References

1. Duer, Zach. 2023. “Home.” *Zachduer homepage*, https://www.zachduer.com/, accessed 2 March 2023.

2. Duer, Zach. 2023. “Zach Duer.” *Virginia Tech School of Visual Arts (SOVA) web page*, https://sova.vt.edu/faculty/zduer/, accessed 2 March 2023.

3. Duer, Zach. “Planning Document for Untitled Video Game.” 2023, obtained from author (also located in Appendix C).

4. *Final Fantasy XII*. Playstation 2 version, Square Enix Co., Ltd., 2006.

5. *Final Fantasy XII*: User manual. Playstation 2 version, Square Enix Co., Ltd., 2006, pp. 24, 27, obtained from https://archive.org/details/FinalFantasyXIIManual. Accessed 29 Apr. 2023.

6. Sabanov, Daniel, et al. 2023. *Virginia Tech CS4624 team project repository*, https://git.cs.vt.edu/sdaniel19/airpgsystem, accessed 28 April 2023.

7. Technologies, Unity. “Unity Documentation.” *Unity Documentation*, https://docs.unity.com/, accessed 10 March 2023.

Appendix A

Methodology

A.1 Goals of System

Users: The system will have two types of users: the player, who wants the system to be entertaining, challenging, and immersive; as well as the game developer who wants the system to have a low level of abstraction, capable of managing multiple modular abilities, and encourage the player to work within a specific in-game social role. So, the system needs to be entertaining, challenging, immersive, low abstraction, capable of managing modular abilities, and encourage players to work within a specific social role.

A.2 System Tasks/Subtasks

The goals that were previously identified can now be broken into smaller tasks.

- Entertaining
  - AI constantly Active
    - Constant purposeful movement
    - Cast abilities when given a chance

  For the AI to be entertaining it should constantly provide action for the player. It needs to make attempts to avoid attacks while also forcing the player to attempt to avoid its attacks.

- Challenging
  - The AI should take full advantage of its abilities.
    - AI attempts most optimal choice.
    - AI learns from past choices.
    - AI adapts to player's choices.

For the AI to be challenging it should make a best choice in each situation, learn what abilities worked best in the past, and adapt to previous player choices.

- Immersive
  - AI cannot obtain information that the player cannot obtain.
  - AI must have the reaction speed of a regular human.

  The AI must be immersive, meaning that it should make the player feel as if they are actually in the game. To help achieve this feeling, the AI should only use information that the player can obtain and have the same restrictions as the player so that it feels like another human that the player is playing against.

- Low Abstraction
  - AI has access to the same information as one would in real life.
  - Retain/implement "fuzziness" in regards to thresholds, comparisons, etc.

  The goal of low abstraction requires that the AI receives the same information that would be available if one was truly in the game world. To further lower the abstraction, the numbers of this information should contain "fuzziness" to represent a person having to make estimates from the visual clues they would see.

- Manage modular abilities
  - AI recognition of ability "categories"
    - Categorize abilities
    - AI can understand usage of categories.

  Managing the modular abilities is one of the more important goals for the AI. To do this the AI needs to recognize and identify different categories of abilities and "understand" how to use these different categories.

- Encourages players to work within specific social roles
  - Prioritize abilities that work well in a specific role
    - Assign social role to AI when it is created.
    - Determine available actions that fit well in social role.
    - Prioritize selected actions.
  - Allow AI to interact with environment when optimal
    - Determine interactable objects in environment.

- Match interactable object with action.
- Factor in action when selecting.

An overarching goal of our client is to encourage players to work in social roles. Therefore the AI should help facilitate that; this means that the AI should have its own social role to follow and use its role in making some of its decisions. To further help achieve this the AI needs to not only be able to fight, but also interact with its environment in a role appropriate way.

Appendix B

Full Ability Tag List

B.1 Table of Range Type Tags

Range Type Tags	
Name	Description
SELF	The main deliverable(s) of the ability target the user.
MELEE	The main deliverable(s) of the ability require/prefer the user to be at close range to their target.
RANGED	The main deliverable(s) of the ability require/prefer the user to be at a far range from their target.
AREA_OF_EFFECT	The main deliverable(s) of the ability affect an area around the target. Primarily meant for abilities which target the user but are cast with the intention of affecting nearby entities.

Table B.1: List of all currently implemented range type ability tags

B.2 Table of Effect Type Tags

Effect Type Tags	
Name	Description
MOVEMENT	Some part of the ability causes the user to move. This should likely be improved on in a future iteration to differentiate between targeted and untargeted movement abilities.
CROWD_CONTROL	Some part of the ability causes the primary/secondary target to move. Note: This should also be applied when an ability applies "indirect movement" in the form of creation of terrain.
DMG_SELF	Some part of the ability applies damage to the user. Note that although damage is considered a "status effect" by the ability system, this tag denotes damage separately as a way to prioritize it when applicable.
DMG_TARGET	Some part of the ability applies damage to target.
BUFF_SELF	Some part of the ability applies a beneficial status effect to the user.
BUFF_TARGET	Some part of the ability applies a beneficial status effect to the primary/secondary target.
DEBUFF_SELF	Some part of the ability applies a detrimental status effect to the user. Unlikely to be used, but added for thoroughness.
DEBUFF_TARGET	Some part of the ability applies a detrimental status effect to the primary/secondary target.
HEAL_SELF	Some part of the ability heals the user or replenishes the user's resources. Note that although healing is considered a "status effect" by the ability system, this tag denotes it separately as a way to prioritize it when applicable.
HEAL_TARGET	Some part of the ability heals the primary/secondary target or replenishes the target's resources.
PLACE_OBJECT	Some part of this ability places a new object/entity/terrain on the field. It is assumed that this new object is beneficial to the user.
TARGET_OBJECT	Some part of this ability directly targets an existing object/terrain. When this tag is applied, all other primary/secondary target tags apply to this object instead if possible.

Table B.2: List of all currently implemented effect type ability tags

Appendix C

Planning Document for Untitled Video Game

The following is a version of Prof. Duer's game design document as of 10 May, 2023. It is provided as part of this report at the written permission of Prof. Duer:

The ultimate goal of Genexegesis [the working title for the game] is to create an online role playing environment that brings into alignment the socially functional experience of the player and the character they are role playing as. The players' primary interface with the world, abilities, structure their interactions with other players by encouraging them to inhabit a social niche. The world provides challenges that incentivize players to use their abilities. For example, in an area that is dark, a player that can create light is a natural hub for other players that require light to see. Or, a player whose abilities get stronger when they kill other players is structurally incentivized to do so, and other players are naturally inclined to avoid or eliminate them.

At the forefront of the ability system is a combat model, but that is not the only interaction provided through abilities. Characters (both player and non-player) of a particular role will be able to influence the world by acting on elements such as the ecosystem, communication, trading, map information, weather, terrain (maybe), or perhaps even abilities themselves, among others. Each kind of interaction represents a social role that acts on the world and/or other players.

The interactions that the character has with the world should be in line with the interactions the player has. As much as possible, the character's experience should be as similar as the player's experience, so that the player is focused on their social niche rather than on goal optimization. To accomplish this, there are four main pillars of ability design:

1. Abilities are non-abstract. Wherever possible, all interactions with the world should be What You See Is What You Get. For example, placing a cloud of smoke makes it hard for both the player and the character to

see. Teleporting the character changes both the position of the character and the position from which the player is able to interface with the virtual world. Grappling with another character prevents both characters from moving quickly as they hold each other in place, limiting the actions and movement that can be taken by both characters and both players. This means that traditional RPG abstractions such as stats should not be used. Inevitably there will be abstractions because the game cannot represent every detail of a world, but they should be eliminated wherever possible to aid in the overall vision.

2. Character abilities and properties should be directly diegetically represented wherever possible. This is tautological with WYSIWYG abilities. Any abstractions should also be diegetically represented. For example, if characters have health, this should be represented through their appearance in the game environment rather than through a HUD. A player's available subset of abilities should be diegetically represented (for example, through what they are wearing).

3. Abilities should focus on enabling both the character and player to fulfill a social role in relationship to other players, stimulating meaningful interaction. This is on a spectrum. As an extreme negative example, imagine there is an ability to plant trees that only that character can see. This has no impact on the player's interactions with other players.

   Making the trees visible is better, because they might limit other players' vision or movement. Even better, if other players can harvest trees as a resource for using their own abilities, there is now the beginnings of a social interaction web.

4. Social roles should be well-defined and immediately apparent.

5. Player power should mirror character power. In many traditional RPGs there is a leveling system in which the character gets arbitrarily more powerful in their abstractions (by getting more health or hitting harder, for example) but the player is left with exactly the same interactions they had before (i.e., click this button to attack). To accomplish this, there is a horizontal power system where the character (and therefore also the player) are given more abilities as they progress. A broader action space is inherently more powerful provided that each ability has a social function.

6. The interaction system (including the combat system) should be structured to necessitate the improvisational use of abilities with medium-duration or long-duration player decision time. No player actions should be done by rote. Players should be faced with complex decisions that require them to consider which abilities to use and when. Outside of

combat, those decisions might be complex enough to require minutes or hours of deliberation. In combat, the decisions must balance complexity so that the minimum decision time should be approximately 1.5 seconds, but in many situations may be longer. In other words, we want to discourage players from being able to mindlessly perform actions. It should be impossible to follow a guide that shows the optimal abilities to use, because the situations should be complex enough that a guide cannot adequately prepare the player for a situation. This should apply to both in-combat and out-of-combat decisions. To accomplish this, the game requires a deep decision-space provided by a wide array of available abilities that respond in uniquely functional ways to the situation. It also requires either player vs. player combat, or a robust AI that can intelligently navigate the deep decision-space in order to adequately challenge players. AI should have no inherent advantage over players from rapid decision making due to the minimum decision time.

7. Fulfillment of social role is more important than balance. If each character archetype is uniquely good at a certain type of role, then balance doesn't exist. However, since there will be some overlap by necessity, the design should always lean toward making a player feel valuable in their social role.