Logic Team 1 Design

Data Communications term project - BCIT

ChrisTOFER Klassen – Julian brandrick – Joel mabbott

2015

Table of Contents

[Overview 3](#_Toc410840160)

[User Interface 3](#_Toc410840161)

[Overview 3](#_Toc410840162)

[State Diagram 3](#_Toc410840163)

[Legend 3](#_Toc410840164)

[Diagrams 5](#_Toc410840165)

[Menu Functionality 6](#_Toc410840166)

[Button Class 6](#_Toc410840167)

[RedirectButton Class 7](#_Toc410840168)

[UtilityButton class 8](#_Toc410840169)

[ActionButton class 8](#_Toc410840170)

[Diagram 9](#_Toc410840171)

[Chat Functionality 9](#_Toc410840172)

[Pseudocode 10](#_Toc410840173)

[Diagram 10](#_Toc410840174)

[HUD Components 11](#_Toc410840175)

[Pseudocode 11](#_Toc410840176)

[Network Message Protocol 11](#_Toc410840177)

[Overview 11](#_Toc410840178)

[Message class 12](#_Toc410840179)

[Pseudocode 12](#_Toc410840180)

[EntityMessage class 12](#_Toc410840181)

[Pseudocode 12](#_Toc410840182)

[UserMessage class 13](#_Toc410840183)

[Pseudocode 13](#_Toc410840184)

[ConnectionMessage class 14](#_Toc410840185)

[Pseudocode 14](#_Toc410840186)

[Diagram 14](#_Toc410840187)

[Round Structure 14](#_Toc410840188)

[Overview 14](#_Toc410840189)

[Round Procedure 15](#_Toc410840190)

[State Flow Diagram 15](#_Toc410840191)

[Deities and Abilities 16](#_Toc410840192)

[Overview 16](#_Toc410840193)

[Ability Overview 16](#_Toc410840194)

[Ability List 16](#_Toc410840195)

[Pseudo Code 16](#_Toc410840196)

[Sample Ability 18](#_Toc410840197)

[Task Breakdown 18](#_Toc410840198)

[Aspect Selection 19](#_Toc410840199)

[Ability Use 19](#_Toc410840200)

[Scoring Logic 21](#_Toc410840201)

[Attributes 21](#_Toc410840202)

[Inherited Attributes 21](#_Toc410840203)

[Deity Attributes 21](#_Toc410840204)

[Pseudo Code 21](#_Toc410840205)

[Map Generation and Environment 22](#_Toc410840206)

[Overview 23](#_Toc410840207)

[Task Breakdown 23](#_Toc410840208)

[Terminology 23](#_Toc410840209)

[Entity 23](#_Toc410840210)

[Cell 24](#_Toc410840211)

[Coordinate 24](#_Toc410840212)

[Block 24](#_Toc410840213)

[Map Generation 24](#_Toc410840214)

[Descriptions 24](#_Toc410840215)

[Pseudo Code 26](#_Toc410840216)

[Abstract Map Elements 32](#_Toc410840217)

[Hazards 32](#_Toc410840218)

[Effects 32](#_Toc410840219)

# Overview

Chris Klassen

This document contains design information related to the three major categories covered by Logic Team 1: **User / Network Interface**, **Deities**, and **Map Generation**, as well as some minor categories.

The User Interface section is credited to two people due to the category swap from Logic Team 2 to 1. The original work was started by Filip Gutica, and was completed and expanded upon by Julian Brandrick. All designers are properly credited per section.

# User Interface

Julian Brandrick, Filip Gutica

## Overview

This section contains the technical design details related to the functionality of the general user interface. These topics are separated into chat functionality, HUD components and menu functionality.

This aspect of game logic is closely related to the work done by the Multimedia team. A few assumptions are made as to how much of this belongs to game logic.

## State Diagram

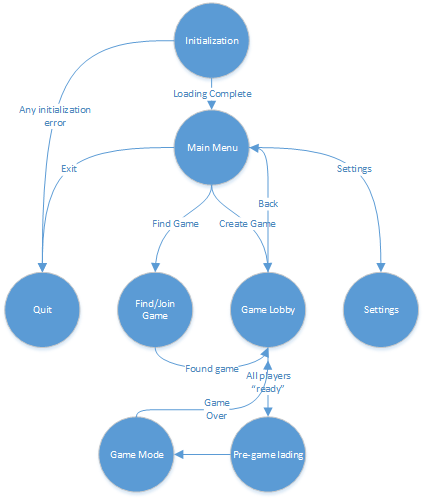
This is the state diagram for the flow of the menu. It will focus more on the functional elements rather than the visual.

### Legend

|  |  |
| --- | --- |
| State Name | State Description |
| Start | When the application is first initialized. |
| Main Menu | The main navigational page |
| Options Button | Redirects from the Main Menu to the Options Menu |
| Options Menu | A page that contains all of the adjustable technical settings for the game |
| Apply Button | Applies the settings the user has specified |
| Connect to Server Button | Displays the Connect to Server Dialogue |
| Connect to Server Dialogue | Allows the user to enter a user name and the IP address of a server |
| Lobby Menu | A page that allows the different users to chat, select their class or aspects and join a game |
| Chat Box | A text input box that allows the user to send text messages to other users connected to the same server |
| Ready Button | Sets the user’s ready status to true and changes their ready symbol from red to green |
| Vessel Button | Redirects the user to the Vessel Menu |
| Vessel Menu | Allows the user to choose a Vessel class and ability |
| Deity Aspect Button | Redirects the user to the Deity Aspect Menu |
| Deity Aspect Menu | Allows the user to pick Deity aspects |
| Exit Button | Signals the application to start freeing its resources |
| End | When all of the applications resources are freed |

### Diagrams



**

## Menu Functionality

The functionality of the menu can essentially be broken down into what kinds of buttons there are and what are their responsibilities. There are three types of buttons classes, all inheriting from a generic Button class.

### Button Class

This is a generic button with no functionality on its own. It holds the buttons ID and name as well as the abstract ButtonClicked function which will run whenever the button is clicked.

#### Pseudo Code

1. class Button
2. {
3. variables:
4. button\_name;
5. button\_ID;
6. functions:
7. ButtonClicked ();
8. }

### RedirectButton Class

This is the simplest type of button. When clicked, a new page is displayed. It inherits all of the Button class’s members and the abstract function ButtonClicked. It has a single native variable called page\_link which holds the descriptor for the new page.

#### Pseudocode

1. class RedirectButton extends Button
2. {
3. variables:
4. page\_link
5. functions:
6. ButtonClicked ()
7. {
8. Go to page\_link’s page
9. }
10. }

### UtilityButton class

This button type is the least solid. It handles all option setting changes for the specific client. This is currently the least solid since there are no concrete option settings. Despite this I can assume that the options menu will have several settings that can all be changed independently and will be saved with an ‘Apply’ button. It makes the most sense for this button to be an Information Expert and simply hold all of the settings data, at least for now.

#### Pseudocode

1. class UtilityButton extends Button
2. {
3. variables:
4. /\*
5. \* Various settings
6. \* resolution
7. \* sound
8. \* etc
9. \*/
10. functions:
11. ButtonClicked ()
12. {
13. Get value from each input control on the page
14. Store them in variables
15. }
16. }

### ActionButton class

This button type is the most complex, it is any button that is connected to data that must be sent to the server. It has two native variables, the first of which is user\_name. This is used to identify the client the network message is coming from. The second is user\_data, which holds the data related to the button click. When ‘Connect to Server’ is clicked it sends the IP address, when a class is selected it sends the selected class.

#### Pseudocode

The UserMessage class is explained in the Network Message Protocol section.

1. class ActionButton extends Button
2. {
3. variables:
4. user\_name
5. user\_data
6. functions:
7. ButtonClicked ()
8. {
9. Create UserMessage with user\_name and user\_data
10. Call onUpdate network function with Message structure
11. }
12. }

### Diagram



## Chat Functionality

Part of the chat functionality has been explained above, pressing the Enter key to send the text will be handled by an ActionButton. When Enter is pressed it will take the name of the current client’s user and the text from the textbox and send them in a Message structure to the network. It has two variables:

* ActionButton
  + Explained above.
* TextBox
  + A class used to represent an area on the screen that the user can enter text into. It has a buffer variable which holds characters and a KeyboardListener function which will take input from the keyboard and store it in the buffer.

### Pseudocode

1. class TextBox
2. {
3. variables:
4. buffer
6. functions:
7. KeyboardListener ()
8. {
9. Pull input from keyboard
10. Store input in buffer
11. }
12. }
13. class ChatBox
14. {
15. variables:
16. ActionButton
17. TextBox
19. functions:
20. MouseListener ()
21. {
22. if click on TextBox is registered
23. Activate TextBox's KeyboardListener function
24. Get buffer from TextBox
26. if click on ActionButton is registered
27. if chat dialogue is not displayed
28. display chat dialogue
29. if buffer is empty
30. Ignore action
31. else
32. Call ActionButton's ButtonClicked function with buffer
33. }
34. }

### Diagram



## HUD Components

This part of the User Interface is about the functionality behind the users Heads-Up Display. With the previous sections it’s easy to see where the Game Logic responsibility starts and ends, but here it is a bit vaguer. For now I will assume that Game Logic’s focus will be on having the HUD data available and knowing when to change it. There will be an InitHUD function that will set all of the initial values for the HUD at the start of a round. There is also an UpdateHUD function that waits for an event from the client that will signal it to update a HUD element.

### Pseudocode

1. Function InitHUD ()
2. {
3. Set initial values for all HUD elements
4. }
5. Function UpdateHUD ()
6. {
7. Wait for character event
8. Update HUD element based on character event
9. }

# Network Message Protocol

Julian Brandrick

## Overview

This section concerns the messages sent from a client, to the server and back to the other clients. Any time an event occurs with a user that all other user must know about (ex: vessel moved, enemy lost health, deity chats), any time an entity must be updated (ex: minion dies, boss moves, pot gets destroyed) and any when the server wants to verify the connection of a client.

For these purposes there will be three Message classes all inheriting from a generic parent.

## Message class

This is a generic Message which will never be used for functional communication between the server and the client, but only as a base for the three other Message types. It only holds a message\_type variable which will be used to differentiate the three different message types for the server.

### Pseudocode

1. class Message
2. {
3. variables:
4. message\_type
5. }

## EntityMessage class

This class is used for all messages concerning entities in the game environment and will be the most heavily used. It has four native variables:

* entity\_event
  + Describes the event of that prompted this message to be sent. It can either be Create, Delete or Update.
* entity\_ID
  + The unique descriptor of the entity being updated
* entity\_value
  + The value of the entity being changed.
    - If the event is an Update, this value will be one of the entities variables.
    - If the event is a Create, this value will be one of the Entity child classes.
    - If this event is a Delete, this value will be NULL or something of similar meaning.
* entity\_update
  + This variable will currently only be used for Update events. It lets the server know how much the specified value must be changed
    - If the event is not an Update, this variable will be zero. This seems like a waste of a variable, but it is justified since the vast majority of Entity messages will be due to Update events.

### Pseudocode

1. class EntityMessage extends Message
2. {
3. variables:
4. entity\_event
5. entity\_ID
6. entity\_value
7. entity\_update
8. }

## UserMessage class

This class concerns all user-based events that must be communicated to all other users, but do not take place in the game environment. Whenever a user chats or when a user picks a vessel class, all other users must know about this event. This class has three native variables:

* user\_event
  + The event, triggered by the user, which caused this message to be sent. As there currently aren’t that many user-based events that could be sent, it is easy enough to have them each defined as a separate instance.
* user\_name
  + A unique descriptor for the user who triggered the event.
    - If the event was a Chat, the user’s name will appear before the text in the chat window and their Chat privileges will be checked.
    - If the event is a Class Selection, the user’s class will update.
* user\_data
  + The data associated with the specific event.
    - If the event is a Chat, the data will contain the text message.
    - If the event is a Class Selection, the data will contain the name of the class type.

### 

### Pseudocode

1. class UserMessage extends Message
2. {
3. variables:
4. user\_event
5. user\_name
6. user\_data
7. }

## ConnectionMessage class

The responsibility of this class is to update the server with the connection status of the client. It may be used nearly as much as the EntityMessage class depending on how often the Network team feels the server should be notified. It holds a single native variable:

* connection\_stable
  + A binary type used to tell the server the status of the connection between it and the client
    - If connection\_stable is true, the connection is still intact.
    - If connection\_stable is false, the connection has encountered an error.

### Pseudocode

1. class ConnectionMessage extends Message
2. {
3. variables:
4. connection\_stable
5. }

## Diagram



# Round Structure

Chris Klassen

## Overview

This document contains technical details, design work and task breakdowns for all areas of the game pertaining to **general match procedure**.

Match procedure design includes state flow diagrams that walk through the general procedure of a match and illustrate some **key decisions** and actions to be taken as part of match procedure.

## Round Procedure

Each round follows a distinct procedure from start to finish. This process is defined below using a state flow diagram that illustrates how a player might progress through the game round-by-round.

It is assumed that the player’s game can end at any point due to them exiting or due to the server closing.

### State Flow Diagram



# Deities and Abilities

Joel Mabbott

## Overview

This section deals with the technical aspects of all playable Deity characters. These characters spectate the game field, and influence gameplay through the use of abilities that affect the other players inside the field.

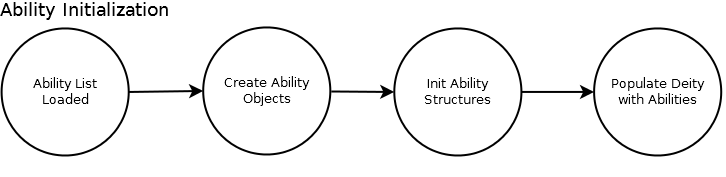
The details shown below are subject to change and are largely based off of logical interpretation. As of the creation of this document the design of Project Spectre is not yet complete.

## Ability Overview

Abilities are not limited to Deities, Bosses and other Player characters will also use abilities. However, most of the abilities in the game are used by Deity players, so Ability structure will be defined here.

### Ability List

Each ability in the game will be defined in a list of abilities. This will be stored as a file somewhere in the programs installation path. When the game world is initialized, the ability list will be loaded into memory, for use within the game.



### Pseudo Code

class Ability

{

variables:

Struct

{

String Name

Short ID

String Icon (contains a path to load the icon image)

String ToolTip

Short TargetType (possibly some enumerated type)

Int Cooldown

Int Duration

Effect (function pointer to the contents of the effect)

Some number of boolean flags

}

constructor

{

use initialize function to set up structure

}

}

Functions:

Initialize Ability List function

{

open ability list from file

error check

error (file not found or unreadable)

return

allocate memory for ability structures

error check

close file

error (unable to allocate memory)

return

initialize ability structures with contents from file

close file

return pointer to array of ability structures

}

Use Ability function

{

check target type

wait for mouse input

on mouse input

check for invalid target

error (target invalid)

execute effect function for selected ability

return

}

### Sample Ability

An ability will contain a structure, defining its specific behavior. Each ability will contain the following information:

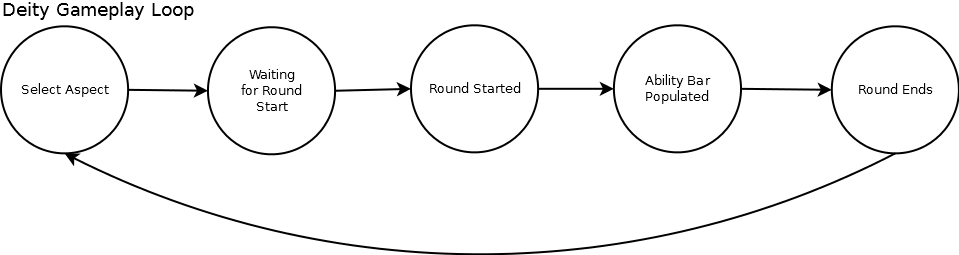
* Name (This appears to the players in game)
* ID (This is used to differentiate between abilities in the code, players don’t see this value)
* Icon (each ability’s icon will appear on the ability bar for the player using that ability. This field can be NULL, for example an enemy boss ability may not need an icon)
* Tool Tip (A short text box that will appear to the player, describing what the ability does)
* Target Type (Abilities will target an entity or a cell)
* Cooldown (The time that must pass before the ability can be used again
* Duration (For abilities with a single, instantaneous effect, this can be NULL)
* Effect (What the actual ability does)
* Flags (May be needed for more complex abilities)

## Task Breakdown

* Deity Class
  + Aspect Selection
  + Ability Use
  + Ability Cooldown Tracking
  + Ability Structure
  + Scoring Logic

## Aspect Selection

Before a game round begins, all players will be presented with a menu option to select the two aspects they will use as a Deity player. When the round begins, the players selected as Deities will have their ability bar populated with the abilities associated with those aspects.

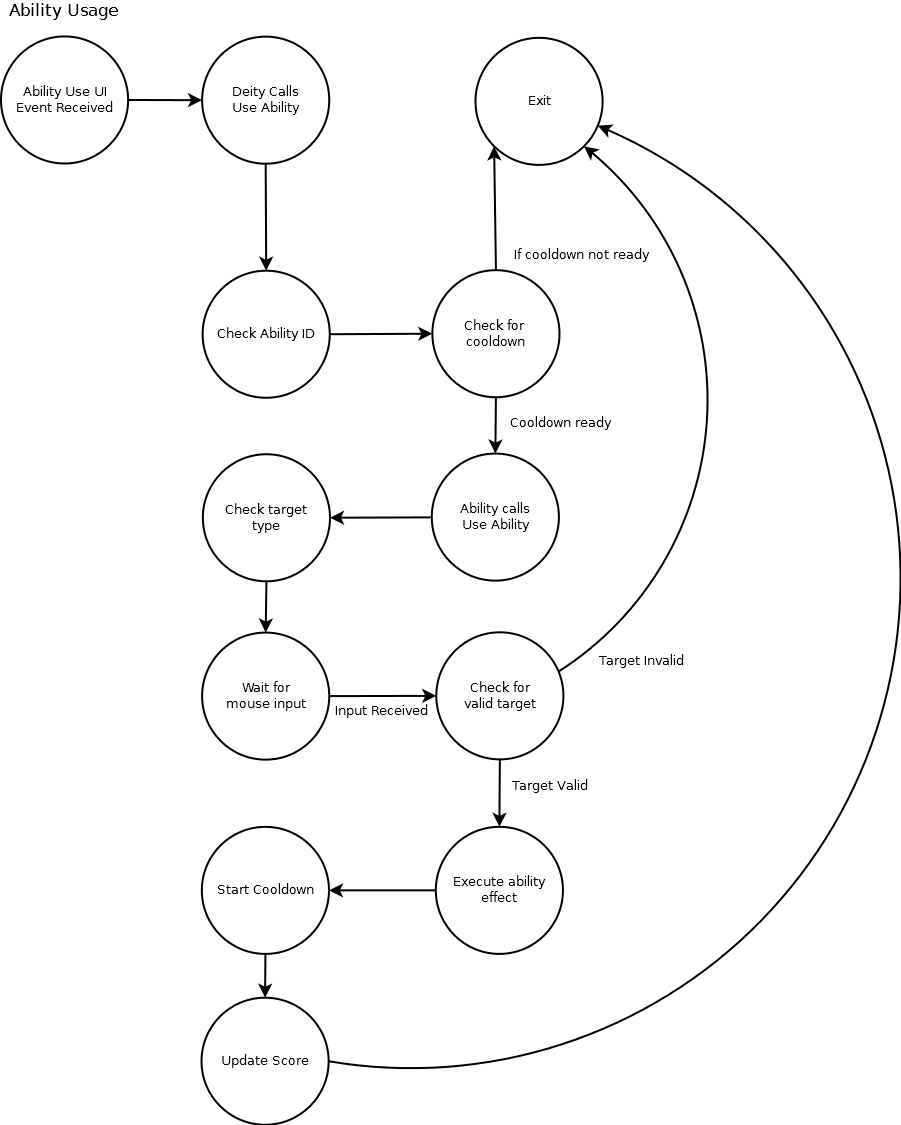


## Ability Use

As the game begins, all Deity abilities will be placed on cooldown. This way, more powerful abilities will not be usable immediately, as their cooldown timers are longer. Once the abilities are available for use, a Deity player can select an ability (via a hotkey or mouse click) then click the target they would like to use the ability on.

If an illegal target is selected (for example, an ability which targets an entity is used on an empty cell) an message will pop up and the ability will not be used. Upon successful use of an ability, its effects will occur, and it will be placed on cooldown once more.

Ability cooldowns will appear to the players as a countdown overlaid across the ability’s icon.



## Scoring Logic

As a player character, each Deity has a score value, which indicates their progress toward winning the game. Correct use of abilities will increase a Deity player’s score.

## Attributes

### Inherited Attributes

This section lists the attributes the Deity branch would inherit from each higher level of the Base Graphical Object hierarchy. This section is not meant to presume on how the hierarchy would be arranged, only to make logical assumptions.

* Location on map
* Camera Control
* Chat Methods

## Deity Attributes

This section lists the attributes that will be contained within the deity branch of the game hierarchy. This is not meant to be a comprehensive list and will likely be added to in the future.

* Deity
  + Aspect Selected
  + Abilities
  + Ability Cooldown
  + Score

### Pseudo Code

class Deity

{

variables:

aspect1

aspect2

abilities

ability cooldowns

score

}

Functions:

Map Movement

{

Behaviour inherited from parent

}

Send Message to Chat

{

Behaviour inherited from parent

}

Use Ability

{

check ID of ability

check if ability is on cooldown

error (cooldown not ready)

call use ability function of ability class

error check

return

call start cooldown function

check for score event

call update score function

return

}

Start Cooldown

{

Get cooldown value for specified ability

Set ability cooldown variable to not ready

Start a timer

On timer end

Set ability cooldown variable to ready

return

}

Update Score

{

Behaviour inherited from parent

}

# Map Generation and Environment

Chris Klassen

## Overview

This section contains details pertaining to map generation and general environment interaction. It contains a task breakdown defining generation tasks, as well as detailed descriptions of each generation step.

## Task Breakdown

This section contains a list of tasks that will be required for the completion environment implementation. This list is not assumed to be wholly comprehensive, but instead aims to identify the major areas of work that need to be addressed.

* Entity placement
* Map generation
  + Create block class
  + Create block data structure
  + Determine zone layouts
    - Assign blocks to zones
  + Player placement
  + Mini-boss location selection
  + Mini-boss type selection
  + Placeholder block type selection
  + Enemy generation
    - Enemy group composition
  + Structure and object generation
* Dynamic tiling
  + Static tiles
  + Unique tiles
* Hazard spawning
  + Hazard spreading
  + Hazard ending
* Effect spawning
  + Effect ending

## Terminology

This document references a number of game-specific terms that are important for understanding the design properly. These terms are listed here with a brief explanation.

### Entity

An entity (in the context of this document) is considered to be any object that exists within the game world. This can be broken down into two categories: Blocking and Non-Blocking.

Blocking entities are objects such as the player, enemies, and environmental obstacles.

Non-Blocking entities are objects such as hazards, effects and abilities.

### Cell

A cell is a small area on the game map that an entity can occupy. An entity can occupy more than one cell at a time and can exist in inter-cell space. If a cell is occupied by a blocking entity, it cannot be entered by another blocking entity.

### Coordinate

A coordinate represents a specific location on the game map (with an origin of 0, 0 in the top left corner of the game map) that an entity can exist at, or be created at. Coordinates are smaller than cells, and can be used to reference specific places inside a cell.

### Block

A block is a small subdivision of the game world. Each block is composed of a number of cells and can be considered either placeholder or pre-set.

Placeholder blocks are filled with randomly-selected content at the beginning of each round and are scattered across the map.

Pre-set blocks will contain the same content across each round.

## Map Generation

Map generation occurs at the beginning of each new round. Maps will be generated using a combination of static and random elements. The steps associated with generation are listed in order below:

1. Calculate the map dimensions
2. Define map zones
3. Place the boss
4. Place the players
5. Place mini-bosses
6. Define placeholder blocks
7. Generate enemies
8. Generate miscellaneous objects
9. Generate tiles

### Descriptions

#### Calculating Dimensions

Map dimensions are calculated in units of blocks. The map must be divisible by the size of a block in both dimensions, and must be comprised of an odd number of blocks in each dimension.

#### Defining Map Zones

The game map is comprised of 3 distinct zones: Grass, Stone, and Arbiter. Each block in the game must be assigned to a zone before generation is done. This selection is done based on the overall size of the map, and relative sizes of each zone.

#### Placing the Boss

The boss should be placed at the centre of the map, in the exact same location each round. The boss is contained within a pre-set block that has the same layout each round.

#### Placing Players

At the start of a round, players should be placed in each of the four corners of the map. This selection is done completely randomly, starting with the player that was added to the player list first. No starting location should be more optimal than the others, and therefore it does not matter where players are placed.

#### Generating Mini-Bosses

Mini-bosses should be placed in a spread-out way across the game world in such a way that they are relatively equidistant from all in-game players. Mini-boss generation is done by dynamically calculating how many should exist in each area of the map, and then selecting the best possible blocks for them.

#### Defining Placeholder Blocks

Each placeholder block in the game world must have its contents defined before the round starts. The contents of a placeholder block are selected from a list of random options. When one of the options is selected, there is further opportunity to randomize the placement or type of content in the block.

#### Generating Enemies

Enemies should be generated semi-randomly throughout the game world at the start of each round. Enemies are generated at the beginning of the round and not again, except in cases of deity abilities.

Generation should be done cell-by-cell, taking into consideration what currently exists in that cell (eg: mini-bosses, the boss, hazards). Enemies should be generated in groups of varying size depending on the enemy type.

#### Generating Miscellaneous Elements

Any final objects placed in the game world, whether they can be interacted with or not, should be placed on a cell-by-cell basis and should be generated based on what type of item they are.

This process could be done via a sort of generation class, or a function within each object type that determines their generation process.

#### Generating Tiles

Tile placement should be done after all cells have been placed, so that new structures and thematic locations receive the same tiling pass as the rest of the map. Tiling should be done in such a way that map edges, divisions and quadrants are automatically identified and tiled correctly. No tiling should need to be done by hand, except perhaps the base game map.

This should be completed by looping through each tile space on the game world, identifying where that space is in relation to the surrounding tiles and the overall game world, and placing the correct tile.

In cases where multiple tiles can be placed, a random selection should be made. This random selection should be more likely to choose nondescript tiles and less likely to pick unique tiles. An example of this is a grassy field where flower tiles are occasionally placed.

### Pseudo Code

#### Map Generation

**Generate Map Function**

{

calculate the map dimensions

if map dimension calculation fails

{

exit

}

define map zones

place the boss

place the players

place mini-bosses

define placeholder blocks

generate enemies

generate miscellaneous objects

generate tiles

return the map data structure

}

**Calculate Map Dimensions Function**

{

if the map width is not divisible by the block width or

the map height is not divisible by the block height

{

print an error message

return false

}

if the map width is not even or the map height is not even

{

print an error message

return false

}

create a data structure for containing all map blocks

loop through and create each block, adding it to the data

structure

return true

}

**Define Map Zones Function**

{

Set the zone of the centre-most block to Arbiter Zone

loop through all blocks

{

if the block is within the innermost 40% of the map

{

if the block does not have a zone

{

set the zone of the block to Stone Zone

}

}

else

{

set the zone of the block to Grass Zone

}

}

}

**Place Boss Function**

{

Place the Arbiter at a random coordinate within the Arbiter Zone

set the selected block to be the Arbiter block

}

**Generate Mini-Bosses Function**

{

Place a random mini-boss in each of the four Stone Zone corner

blocks

if the Stone Zone is wider than 3 blocks and taller than 3 blocks

{

Place a random mini-boss in all Stone Zone edge blocks that

form a cross with the Arbiter Zone

}

place a random mini-boss in all Grass Zone edge blocks that form

a cross with the Arbiter Zone

set all selected blocks to be mini-boss blocks

}

**Define Placeholder Blocks Function**

{

loop through all untagged Grass Zone blocks

{

select a random block type from the Grass Zone list

set the block to the selected type

}

loop through all untagged stone blocks

{

select a random block type from the stone zone list

set the block to the selected type

}

}

**Generate Enemies Function**

{

loop through all enemy-type Grass Zone blocks

{

select a random Grass Zone enemy grouping

generate a number of enemies in the block across a random

range

}

loop through all enemy-type Stone Zone blocks

{

select a random Stone Zone enemy grouping

generate a number of enemies in the block across a random

range

}

}

**Select Enemy Grouping Function**

{

create an empty list of enemies

if the Zone is the Grass Zone

{

set the maximum block value to the Grass Zone Max

}

else

{

set the maximum block value to the Stone Zone Max

}

set the current value to the maximum block value

while the current value is greater than zero

{

select a random enemy with a value that is less than or

equal to the current value

add the enemy to the list

subtract the enemy value from the current value

}

return the enemy list

}

**Generate Objects Function**

{

loop through all special-type blocks

{

select a structure or object from the list of options

generate objects in the block across a random range

}

}

**Generate Tiles Function**

{

loop through all grass blocks

{

generate grass tiles based on edges and intersections

generate random unique tiles

}

loop through all stone blocks

{

generate stone tiles based on edges and intersection

generate random unique tiles

}

generate Arbiter tiles based on edges and intersections

generate random unique tiles

}

#### Entity Placement

**Entity Placement Function**

{

If the entity is non-blocking

{

create an instance of the entity class

initialize the entity with the necessary values

set the entity's position to the desired x and y

coordinates

add the entity to the cell closest to the x and y

coordinates

}

else

{

if the closest cell to the x and y coordinates is empty

{

create an instance of the entity class

initialize the entity with the necessary values

set the entity's position to the desired x and y

coordinates

add the entity to the cell closest to the x and y

coordinates

}

else

{

return false

}

}

return true

}

## Abstract Map Elements

Abstract map elements are considered to be Non-Blocking entities that have an effect on the game world and other entities. The major categories of these elements are **Hazards** and **Effects**.

### Hazards

Hazards are entities that can be assigned to a specific block or area of the map. Hazards may or may not be able to spread, but are identified by their ability to harm other entities in the game.

### Effects

Effects are entities that are generated as a result of actions taken in the game by players, artificial intelligence-driven entities and hazards. These are generally defined by their lack of interaction with other entities, and the fact that they are primarily animations or particle systems that disappear after completion.