Tortuga Technical Document

By Kayne Ruse

Introduction 2

Languages and APIs 2

Gameplay Mechanics 2

Combat Portals 2

Permadeath 2

Player Character 2

Player Character Statistics 3

Items 3

Equipment 3

Server Mechanics 3

Server Structure 3

Data Storage 3

Map Structure 4

Region 4

RegionPager 4

Generator Functors 4

Format Functors 4

Lua’s Region API 4

User Accounts 5

Lua Scripting 5

SQL Scripting 5

Modding Support 5

Communication Protocols 5

Client Mechanics 5

Client Structure 5

Platforms 6

Game Controls 6

Map System 6

TODO List 6

# Introduction

This is the technical document for Tortuga. This is intended to plan out required tasks for the game, as well as serve as an instructional text for modders and others interested in Tortuga’s development cycle.

At this stage, I’m filling in the sections as I go; any empty sections aren’t a concern yet. To see some planned and expected features, refer to the design document. As with the design doc, if you see any italic text, you can consider that to be an incomplete or removed section of text or an inline comment.

# Languages and APIs

The languages of choice for creating this game are C++11 and lua, for their large user bases and wide feature sets. Third party libraries I’m using include SDL (Simple DirectMedia Layer), SDL\_net and SQLite3, for much the same reasons.

# Gameplay Mechanics

When each player connects to the server, they can walk around the procedurally generated world. Now, exactly what…

*TODO*

## Combat Portals

*TODO*

*Multiplayer*

*server control*

*map design and generation*

*exploration*

*roguelike dungeons*

*control of server mechanics and scripts*

*travel between regions (world gates)*

*RPG mechanics like items, equipment, stats, etc.*

*Player Interactions*

*TODO*

## Permadeath

*One of Tortuga’s most influential game mechanics is permadeath i.e. the deletion of a character when the player runs out of life.*

*MORE*

# Player Character

*The player characters (PCs) will be created and customized by users. The PCs will gain levels and stat increases as the players progress with that character. When a character’s health value reaches zero, that character will die and is deleted from the server (see permadeath).*

## Player Character Statistics

*Each PC has their own unique set of statistics (stats). Possible PC stats include:*

*Health - Life Remaining*

*Mana - Magic Remaining*

*Level - Skill Level*

*Attack - Offensive Ability*

*Defence - Defensive Ability*

*Strength*

*Speed*

*Luck*

*Magic Channelling - Magic Regen?*

*TODO*

*stats can be increased by methods other than levels and equipment*

*stats increased by items and levels*

*level progression*

# Items

*TODO*

*There will be many items in the server, whether they’re consumable items, equipment or other types.*

# Equipment

*TODO*

# Server Mechanics

*TODO*

*What can a server do, and how does it do it?*

# Server Structure

*TODO*

# Data Storage

*TODO*

# Map Structure

## Region

The game’s map is divided up into units called “Regions”, which are stored in the Region class. This class is fairly simple, requiring the width, height, depth, X and Y positions on creation, and provides two functions for accessing the three dimensional array of tiles.

This class also defines the type Region::type\_t which is the internal storage type for the tiles. Please note that the value 0 is used for empty tiles. region.hpp also defines three macros, REGION\_WIDTH, REGION\_HEIGHT and REGION\_DEPTH, which are used by the RegionPager objects to define the width, height and depth of the regions. Hopefully, these macros are temporary.

## RegionPager

The paging class is divided into two parts: RegionPagerBase and RegionPager. The abstract base class provides access to the correct region objects, via wrapper methods GetTile() and SetTile(). This class also allows the user to set the sizes of the regions, but please note that it is a Very Bad Idea to change this mid program.

The derived class, which takes two template parameters, overrides four abstract methods used for creating, and saving and loading the region objects. The derived class doesn’t do much itself, apart from bridging the gap between the base class and the functor classes provided as template parameters.

## Generator Functors

There are currently two generator functors, BlankGenerator and LuaGenerator. The first creates and cleans up a region object, and simply leaves the default values in place. The second also passes the object to a designated lua function for processing either after creating it or before freeing it.

## Format Functors

The format functors require that the program provide the file path to save the map in before it is used; otherwise its behaviour is undefined. The first, DummyFormat, does literally nothing. The second, LuaFormat, will provide an existing object to lua’s save function, or create a temporary Region, and provide that to lua’s load function. Please note that if the load function returns false, then the specified file was not found and the temporary object is freed instead of being passed to the pager.

## Lua’s Region API

The Region library is written to interact with Region objects as needed. The four most common functions are Save(r, saveDir), Load(r, saveDir), Create(r), and Unload(r). Each of these receives a Region object as a lightuserdata type, while Save() and Load() also receive strings containing the map’s save directory.

Accessors and mutators for the Region objects are also provided. The four functions mentioned above are usually called by the functor objects, and are defined as dummy functions by default. If you want to use them, I recommend redefining them in the server’s start up lua script.

The full list of available library functions (at the time of writing) is:

* Region.SetTile(r, x, y, l, v)
* Region.GetTile(r, x, y, l)
* Region.GetWidth(r)
* Region.GetHeight(r)
* Region.GetDepth(r)
* Region.GetX(r)
* Region.GetY(r)
* Region.Create(r)
* Region.Unload(r)
* Region.Load(r, saveDir)
* Region.Save(r, saveDir)

# User Accounts

*Each person who accesses a server must have their own user account. This allows players to keep track of their PCs, items, and other settings. This will also allow a server owner to whitelist or blacklist certain players, as well as other server specific options.*

*Each user account will have a certain number of PC slots. The items, etc. that a character collects stays with that character when a user logs out.*

*The accounts will be stored in a database.*

# Lua Scripting

*Servers can run custom scripts on the clients, but there needs to be a limit to this.*

# SQL Scripting

*TODO*

# Modding Support

*TODO*

# Communication Protocols

The primary method of communication is a custom UDP protocol.

*TODO*

# Client Mechanics

*TODO*

*Available options, how to connect to a server.*

# Client Structure

*TODO*

# Platforms

At this stage, due to a limited scope and budget, this game will only be available on PC.

# Game Controls

*This game will have both keyboard & mouse support, as well as generic controller support.*

*TODO*

*navigate through menus, move, select, etc.*

# Map System

*TODO*

# TODO List

*Clean up this document*

*Page breaks*

*Add more*