Roguebot

Technical Design Document

Team Terminal Audacity

GAM200F2014

Leonardo Saikali – Producer

Connor Hilarides – Technical Lead

Claire (Jake) Robsahm – Systems Engineer

Troy K. B. de Magro – Lead Designer

Enrique Rodriguez – Gameplay Architect

# Overview

Engine Components

Basic Architecture

* Component Based
* Event Driven

Graphics

* DirectX 11

Scripting

* Ruby scripting via the mruby embedded ruby runtime

User Interface

* Built in the same way as the game (like we do with the Zero engine)

Resource IO

* An abstracted interface for reading in resources in a manner that will allow us to eventually pack all of the game resources into a single file

Game Data

* Game data is described by JSON files
* We have level files and archetype files

Components

* Components are owned by Entities (Game Objects)
* Components can access their owner and the entire level entity structure
* Components written in C++ describe an interface for the scripting engine to interact with
* Most of our components are scripted
* Available to components is an Action list which can either be all-at-once or sequenced
* An entity can have “unlimited” action sequences

# Graphics

We are using DirectX 11 with mesh-based rendering to allow us to provide some interesting shader effects. All of our sprite assets are loaded through the Respack interface, and the loading technique is independent of how the sprites are actually stored in the game. We use the Windows Imaging Component to allow our artists to use any of the supported image formats for their artwork, whatever gives the best balance of quality and file size. Images are loaded in by the TextureComponent or by a user script, and get displayed by the SpriteComponent.

# Behavior

We are building a modular AI system that works in discreet steps due to the turn-based nature of our game engine. All AI calculations are to be done in another thread based on a frozen snapshot of the world generated for each step the AI takes.

# Scripting

All of our games behavior (besides a few components moved to C++ for performance reasons) are written in the Ruby scripting language. The state of each level/floor of our dungeons is stored POROs (Plain Old Ruby Objects) for the convenience of our gameplay programmers to modify to add fields to and such. We’ve implemented many APIs for the scripting language to use for the convenience of gameplay programmers setting up more complicated systems without reducing performance.

# Coding Methods

C++ code should follow our style guide which can be found at {ProjectRoot}/Engine/Style Guide.txt although there will still be some wiggle room on that for your private implementations, as long as it still mostly follows the guide. The Ruby programming language fairly strictly enforces its own style, and that can be found online.

# Debugging

If you’re debugging C++ code, better head over to visual studio. Otherwise, we have the debug console which can be opened in-game (we had it as a windows console back before the game was stable. This can be reenabled if you’re really stuck on a debugging issue). We also have a live property viewer and editor, which can be found by opening the debug console and typing browse. The debug console is just the scripting language.

# Tools

At this moment we do not have any specialized tools beyond standard art programs and text editors, except for our live property viewer. Eventually we may create a UI designer to allow us to better edit our UIs

# Appendix A

## Tools/Editor

We don’t have a traditional editor, but our live property viewer/editor can be accessed by typing ‘browse’ in the debug console. It will open a web browser which contains the editor.