University of Alabama in Huntsville

LOUIS

Honors Capstone Projects and Theses

Honors College

5-1-2017

Multiple Engagement System Engine

James Andrew Rothenflue Jr.

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation

Rothenflue, James Andrew Jr., "Multiple Engagement System Engine" (2017). *Honors Capstone Projects and Theses*. 549.
https://louis.uah.edu/honors-capstones/549

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.

Multiple Engagement System Engine

by

James Andrew Rothenflue Jr.

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

4/28/2017

Honors Capstone Director: Dr. Rick Coleman

Lecturer in Computer Science

James A. Rothenflue Jr.	4/27/2017
Student (signature)	Date
[illegible]	5/1/2017
Director (signature)	Date
[illegible]	5-1-2017
Department Chair (signature)	Date
[illegible]	5/5/17
Honors College Dean (signature)	Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

James A. Rothenflue Jr.

Student Name (printed)

James A. Rothenflue Jr.

Student Signature

4/27/2017

Date

Table of Contents

Abstract

The Multiple Engagement System Engine is an implementation of an original, turn based combat system independent of any specific game, allowing programmers to use or adapt it freely into their own games. The engine reads stage data from a file, takes the remaining data through an API, then manages the logic and data of a battle. It is built in such a way as to make it game independent, allowing different programmers to define different skills and other elements within the framework of the engine. If the user desires any features not supported directly by the engine, the engine provides sufficient tractability to allow the programmer to build a shell around the core engine to implement those features. Additionally, because the engine is open source, the programmer may make direct changes to the engine should the need arise.

The primary unique element of the system is the multiple engagement mechanic. The engine is designed for small squads on each side, each side consisting of approximately 4-10 units. The multiple engagement mechanic causes two units to become “engaged” upon one attacking another. A unit’s abilities will drop based on how many units with which it is currently engaged, falling sharply beyond the first engagement. This mechanic introduces an element of strategy to the system, as the player decides how to use this dynamic to weaken the enemy without placing their own units in danger. With this engine, a game can use this unique mechanic without having to build the details of the system themselves, yet can still tailor level design and other elements to their own specifications.

Introduction

Creating a video game is a large scale task. Major game developers have their own tools to simplify the process and large teams to divide the load, but amateur game developers may not have such advantages. There are full game engines available to them, which make the process a lot easier, but these engines are designed to make building games easier by simplifying processes that are actually very complex. None of the actual features of a game are fully implemented, and not all amateur games are built with game engines. The Multiple Engagement System Engine is a useful, pre-implemented combat engine that fully implements one aspect of a game in a form that may be easily used by programmers who are not using a complete game engine.

The Multiple Engagement System Engine provides a basic implementation of my original Multiple Engagement Combat System, which is described in detail in the engine's instruction manual. The engine provides the actions necessary to customize the basic mechanics into enough forms to give individual games a unique feel. Furthermore, the open source code allows programmers to directly change those features that cannot currently be changed externally. While there are still customization features and design changes that could improve upon the current version, the package comes complete with all the basics necessary to build a fully functional combat system.

Chapter 1: Purpose

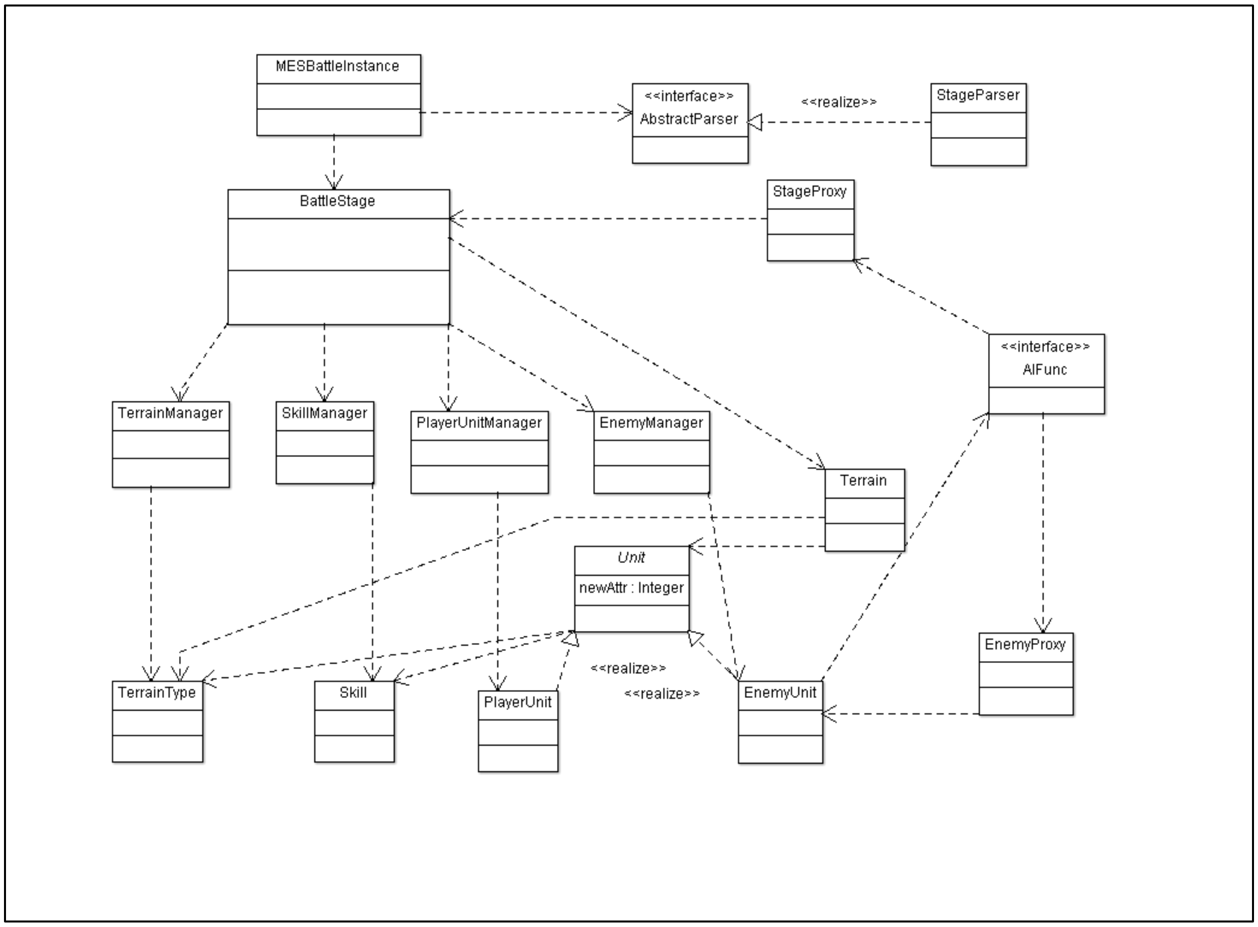
The Multiple Engagement System Engine fully implements the basic battle system I created in a single java package which may be directly imported into a programmer's code. This engine is primarily targeted toward amateur game designers, who lack the resources a larger company has for building games. This engine saves such programmers the trouble of designing and implementing their own combat system from the ground up. This is particularly helpful if the programmer is building the game without using an existing game engine. The Multiple Engagement System Engine is general enough to allow each game that uses it to feel unique, and the open source code makes it easy for programmers to change it into whatever they need it to be. The programmer still needs to implement a combat user interface, and may also decide to tweak the engine source code, but just having a working model on which to start will still greatly alleviate their work load in comparison to building a system from scratch.

Chapter 2: Design

The Multiple Engagement System has four major tiers of hierarchy into which the bulk of its objects can be classified. The lowest tier consists of objects meant to represent individual elements of data, such as a single unit or skill. The next level is made up of manager objects which control access to each of the types of data object. The BattleStage object is the third tier, which serves as the central class of the engine, managing the flow of data between the managers and the MESBattleInstance, controlling the stage data, and performing the high level aspects of actions that affect more than one data element in the engine. The highest tier consists of the MESBattleInstance, which primarily acts as the interface between the engine and the game proper.

Some objects don't fit directly into this hierarchy, such as the parser class and AIFunc interface. Such classes are just as important to the engine, but fill roles that make them hard to group with other classes. A basic class diagram is depicted in figure 1, followed by a class by class description of each class' roles in the engine.

Figure 1: Class Diagram



MESBattleInstance

The MESBattleInstance class represents an individual battle, and is the interface through which a program interacts with the Multiple Engagement System Engine. The object provides the ability to retrieve any visible information from the engine, perform actions inside the battle it represents, or arbitrarily alter any data which may be changed directly by the user. It manages the initialization process for the battle, and tracks only the highest level data of the battle proper, such as turn count.

BattleStage

The BattleStage object is the central object of the engine. It stores the data about the stage's terrain and current state, but also mediates all interactions between the managers. It keeps particularly strong control over the unit managers in order to ensure the data in the Units does not fall out of synchronization with the data on the stage's state. Finally, it is the intermediary for most interaction between the MESBattleInstance and the managers.

AbstractParser

This interface is designed to make it easier to trade out parsers for reading a stage's data file. It defines the data the parser must read from the file, as well as how it must output the data, but places no restriction on how the data is read from the file. As a result, a programmer may choose to implement their own parser if they feel the need to store the data in a different format.

StageParser

The prebuilt implementation of the AbstractParser, will read the stage data from an xml file according to the format described in the Multiple Engagement System Engine's instruction manual.

TerrainType

This data object represents a specific type of terrain a given space on the stage may possess, including stat modifiers and movement cost. TerrainTypes are unchangeable after creation, so the difference between an original and a copy is not meaningful. One TerrainType can apply to as many spaces on the stage as necessary.

TerrainManager

The TerrainManager is the manager object that stores and controls access to all TerrainType objects in the current battle. Only one TerrainManager will exist per battle.

Skill

This data object represents a single skill a given unit may know how to use, including all necessary information on how it modifies an attack. Skills are unchangeable after creation, so the difference between an original and a copy is not meaningful. One Skill can belong to as many units as necessary.

SkillManager

The SkillManager is the manager object that stores and controls access to all Skill objects in the current battle. Only one SkillManager will exist per battle.

Unit

This abstract data object represents a single unit on the stage, including all stats and current state information. The class implements the core functionality shared by all units, such as the low level details of combat and stat management. It does not, however, possess functionality specific to either player or enemy units.

PlayerUnit

A PlayerUnit object represents a single player controlled unit within the battle. In addition to having all the functionality and responsibility of the Unit abstract class it extends, it also tracks whether or not the unit has moved or acted during a given turn.

PlayerUnitManager

The PlayerUnitManager is the manager object that stores and controls access to all PlayerUnit objects in the current battle. It also is responsible for adding and removing engagements from the PlayerUnits within it. Only one PlayerUnitManager will exist per battle.

EnemyUnit

An EnemyUnit object represents a single enemy AI controlled unit within the battle. In addition to having all the functionality and responsibility of the Unit abstract class it extends, it also has various additional features intended to support its AI.

EnemyManager

The EnemyManager is the manager object that stores and controls access to all EnemyUnit objects in the current battle. It also is responsible for adding and removing engagements from the EnemyUnits within it, as well as setting and assigning AI defaults. Only one EnemyManager will exist per battle.

Terrain

A Terrain object represents a single space on the stage. It tracks the TerrainType assigned to the space, as well as whether or not the space is currently occupied by a unit.

AIFunc

This interface allows the programmer to build their own AI for the enemies on the stage.

StageProxy

This object allows programmer implemented AI to access data from the BattleStage and perform actions without exposing any sensitive actions it should not be able to perform inside of an AI function.

EnemyProxy

This object allows programmer implemented AI to access data from the acting EnemyUnit without exposing any sensitive actions it should not be able to perform inside of an AI function.

Chapter 3: Challenges

While building the project, I encountered difficulties that took considerable caution and creativity to overcome. In particular, the issue of protecting variables from unauthorized outside tampering was a frequent and tenacious problem. In the process of solving such problems, however, I gained a greater appreciation just how much effort goes into building code which is intended for public use.

My first experience with this issue was simple in nature, but surprisingly hard to avoid in the long run. I persistently found that objects returned from get functions were often originals, especially when they shouldn't be. As a result, any changes to objects the programmer acquired through legitimate means would be reflected inside of the engine, which could cause significant problems. This kind of error was typically easy to fix, but I was surprised by how often it occurred, and how many different ways copied objects could hide original data it should not possess. I had to spend the entire project, beginning to end, on constant guard against such mistakes, and then tested to catch the many mistakes I made anyway.

A more difficult problem arose from my method of allowing programmers to build their own AI. By allowing users to implement their own AI object, then pass those objects into the engine, I created a path by which the internal state of the engine could be changed from outside. If the programmer added any objects within their implementation which would not be copied by a shallow copy, then altering those objects outside of the engine would do the same inside of it. I could not implement a deep copy of an object I did not create either, limiting my options on closing this hole from inside the engine. The option I finally settled on was to force the user to implement a function to copy their AI implementation, with the specification that it must be a deep copy. The solution is not foolproof, since the programmer may still only implement a shallow copy instead, but it should mitigate the chance of such irregularities occurring by accident.

Finally, the AI posed one more data access issue before the project reached completion. Specifically, I needed to grant the programmer's AI implementation access to some of the elements of both the acting unit and stage in order for it to be able to make educated decisions or perform any actions. Granting it direct access to either object, however, would enable to programmer to directly alter data that they should not be able to edit inside of an AI function. To address this, I created proxy classes for both the BattleStage and the EnemyUnit classes, which provided access to the functions the AI would need, but blocked access to those that would be harmful.

Chapter 4: Assessment

In regard to core functionality, I believe that the Multiple Engagement System Engine lives up to its goals. It successfully implements a rudimentary version of the Multiple Engagement System, allows the programmer to customize or alter the data inside the engine freely enough to allow them to give the combat in their games unique twists, and has function by function Javadoc documentation to aid the programmer if he wishes to directly alter the code.

There are, however, a few ways in which I feel the engine could be improved, the most notable of which being the addition of changeable options. Given time, I would like to be able to add additional means of customization to features that are currently hard coded, such as the maximum number of engagements a unit may have without penalty, the amount by which a unit’s stats drop per engagement above this maximum, and whether or not units may move through each other. The fact that the engine is open source mitigates this issue somewhat, since the programmer is able to directly alter the hard coded values, but it would be more convenient to be able to alter these features with built-in function calls.

The other improvement I would like to make would be to better distribute the responsibility between the MESBattleInstance and the BattleStage objects. As of now, I have placed an abnormally large amount of responsibility on the BattleStage class, especially when one considers that the object would logically be limited to managing the stage terrain and unit positions. It would be more logical to move many of its more administrative tasks up into the MESBattleInstance, or even create one or more new objects to better divide the responsibility. In some cases, there are even instances of similar tasks being performed by different objects depending on whether the functions involved were implemented early in the project or near the end. Reorganizing these responsibilities would make the code easier to understand and more easily editable, but would involve a major overhaul to the engine structure. Therefore, it would be impractical to make these changes in this initial version.

After analyzing the finished project, I must acknowledge that the engine still has many ways it can improve. At a later date, either I or others may wish to revisit the engine to realize these ideas, and bring out more of the engine’s potential. As far as meeting the original design goals, however, the engine is definitely complete and usable. It only needs time and effort to elevate it from its current, basic form to a polished product.

Conclusion

Although there are still features that may enhance the engine’s functionality and design, the Multiple Engagement System Engine is currently a complete tool, ready for amateur game developers to use in their games. The core functionality is fully operational, allowing programmers to easily include the standard Multiple Engagement Combat System into their games. Between the existing functionality and programmer’s ability to directly alter the code, programmers should be able to create a large and varied array of combat systems based on this engine, while still saving them the time and effort of implementing a complete combat system from scratch.