Empire Deluxe Combined Edition

Open Source Guide To Map Making And AI Players

Last updated: 08/31/17

Table Of Contents

Introduction

Regarding World Building

What attracted me initially to Empire, was the map builder. The fact that you could control the conditions of a computer game was an exciting concept to me. The game was one of a few (I recall the game Breach and Breach II having scenario editors as well) that had this capability, and that is pretty ground breaking.

With the advent of *Doom* and *Quake* people began to modify games without manuals and access to some source code. It was all hex numbers and hacks. I myself hacked a map/scenario editor for the original *Perfect General*, and even a symmetrical map program for *Empire Deluxe*.

Nowadays mods are much more common place in games. But they still require you to think way out of the box. And the tools will never be as polished and powerful as you would like.

Maps and scenarios can add so much to the game. Allowing an independent Map Making Dll capability can only extend the game as some of the readers her consider things beyond what I could think of. So if you are one of those people, I encourage you to try and build a new world builder, that suits your style of map.

Regarding AI Players

Empire Deluxe is a great game. Played thousands of hours in my lifetime, most of those have been solitary. I really enjoy P v P play, but even with the most facilitating methods, the results will take a very long time. In fact, P V AI play can take a very long time.

I enjoy *Empire Deluxe* due to its simple abstraction. With this simplicity, there is a perception that the AI should be a smart player. But this is a hard thing to do in the setting of being all things to all game styles.

But for the individual mod maker, the limits are learning how to interface with the game. Beyond that they are only limited by their imagination and ingenuity. They do not have to build an AI player to be everything. It can exist with a very specialized purpose.

So I would like to encourage people to look over the AI open source code. At first, don't take on the massive task of building a brand new AI. Instead – fix something in the old one that annoys you. Add something cool to it. Make it different. Make it better.

Empire Common DLL

Both Map Making and AI Players interface with a DLL assembly called the ***EmpireCommon.dll.*** This is the interfacing library between the AI/Map Maker and the game program. This is code "*you can see, but not touch*", as it is built in with the game code.

The source code is provided so that you can compile it and reference it in your AI projects.

World Building Development Details

Making Map in EDCE

Randomly generated maps have always been a staple of Empire Deluxe. This is a great aspect of the game that can be further enhanced by working with the World Building interface of EDCE. EDCE comes with two world builders – one made up of most of the code World Building code from the original Empire Deluxe, and the other from the World building code of EDEE. Both will supply things like capitals, important cities, resources on demand. The enhanced version has several hints options that can be used. The code for both of these World Builders will be available for further use by people wanting to get their hands dirty in the world of world building.

Technology

First, we need to understand the technology used when making a World Builder. Each individual World Builder is a C# Assembly DLL. There is an interfacing library, which is called the EmpireCommon.dll, which contains the source code needed to interface with the game program. A Word Builder developer would need a C# compiler and a project which includes the Empire Common DLL.

The Express Compiler is free and available from Microsoft:

https://www.visualstudio.com/vs/visual-studio-express/

The latest code can be found this directory:

http://killerbeesoftware.com/edce/code/

Though the code in the Empire Common library is available for review, it is not changeable as this dll is part of the game distribution. The Empire Common library holds several classes useful in World Building. The most important of these are:

•*CWorldBuilderFactory*: An abstract class which will give out the set of map hints and a reference to the actual map maker

•*CDLLHints*: A set of hints/parameters that are available for setting and influencing how a map is generated.

•*CDLLInfo*: An individual parameter/hint. Essentially a name value pair.

•*CMapMaker*: The abstract class upon which your map maker will derive from.

•*TempUnitStruct* – A data class which will contain information of cities and other units generated in the map making processing

•*TempMapStruct* – A data class which will contain information of what lies in a map square, like terrains, roads, mines, resources.

•*CMapUtil* – A Useful utility class which can calculate distances and locations on a map

•*CGameMapParameters* – a set of standard map parameters that go into world building.

•*WBQueryI* – Can be used to get various unit set info.

•*StringPollerI* – interface used to pass messages as the World is Built.

•*MapCallbackI* – interface used to pass data back to the game after the map has been generated.

How The World Builder Works At 50, 000 ft

Every WB DLL must have a factory that can return a reference to your *CmapMaker* class. The Factory must be derived from the abstract *CWorldBuilderFactory* class. It must serve a unique key string identifying the Map Maker as opposed to other Map Maker DLLs, as well as provide the *CDLLHints* object containing “hints” that the person can use to influence the map generation.

The hints will consist of a name/value par setting which can be any of the following types:

•Boolean (yes or no)

•A Value in a numeric range (Min/Max included)

•A file name.

•A numeric value

•A string value

Building a world for a Map Maker is a two step process. First, the MapMaker is told to “prepare” but not build the world. This essentially sets up all important object references and data, including the map hints and parameters. It is important that the Map Maker not doing any generation as this prepare call is made on the main thread, and should be completed as soon as possible.

Later, and on another background thread, the MapMaker is told to “buildWorld”. This is where the genesis occurs. During this time, the map maker can report its progress back to the user through the StringPollerI interface.

At the end of the buildWorld method, the MapMaker will have finshed up things by invoking the MapCallbackI interface with “gotMapAndCities. It actually delivers three lists of objects in this call:

The List Of TempMapStruct representing all of the locations on the map. The list must have map width x map height elements, and is indexed using the CMapUtil object.

The list of TempUnitStruct objects representing the cities on the map.

The list of TempUnitStruct objects representing other units that may have been placed on the map.

AI Player Development Details

A very exciting change with EDCE is the interface with the AI player. I have tried to set things up to encourage more AI development, for us to get different AIs into the game. Here I hope explain some of how this is taking place, and do so from the perspective of someone wanting to tweak or create an AI player for EDCE. This is a very high level view, but hopefully gives you some idea as to the process, and starts us on "*Project AI*".

Technology

First, we need to understand the technology used to create an AI player. Each individual AI will be contained in a platform independent C# Assembly DLL. There is an interfacing library, which is called the EmpireCommon.dll, which contains the source code needed to interface with the game program. An AI developer would need a C# compiler and a project which includes the Empire Common DLL.

The Express Compiler is free and available from Microsoft:

https://www.visualstudio.com/vs/visual-studio-express/

Though the code in the Empire Common library is available for review, it is not changeable as this dll is part of the game distribution.

The latest code can be found this directory:

http://killerbeesoftware.com/edce/code/

Included in this library is the AIPlayerData class. This class can handle most of the incoming events from the game, as well as send commands to the program. If you choose to include this, you would make your AI player a derived class of the AIPlayerData class, and override some of the methods of this class to add your extra sauce. All of the AI players included with the EDCE distribution are derived from AIPlayerData. This is not mandatory, but writing your own requires a much greater level of effort and understanding of the Empire Common API. Alternatively, you could begin with the source of the AIPlayerData in a new class of your naming. In the Common Library, there are no dependencies on the AIPlayerData class. This class is included in the library solely for my configuration management purposes. Ultimately, your AI player must be derived from the AI Player class, which the AIPlayerData class is a derivative of.

How The AI Player Works At 50,000 ft

Every AI DLL must have a factory singleton which knows your AI Player class and is able to create one on demand. This factory must be derived from the abstract AIPlayerFactory class. It must serve a unique key string identifying the AI Player Type as opposed to other DLLs, as well as provide dictionary container of "parameters" or "hints" that the person setting up the game can review and set to make the player unique.

The parameters will consist of a name/value par setting which can be any of the following types:

- Boolean (yes or no)
- A Value in a numeric range (Min/Max included)
- A file name in string form.
- A numeric value
- A string value

When a game is started, the factory is sent a set of those parameters as well as some other information (like position and a handle to a logging file) and told to create a player. The game then uses this player object for the game and communicates with it through the Empire Common interface.

The starting state of the AI is initialized with a State Event, which tells it the current set of game objects of which it is aware. The state event is only sent at the beginning of a game, not in a reloading situation.

The AI itself interfaces with the game program via polling for events and making commands which solicit a response, which could contain events. Commands are only allowed to be made during the player's actual turn, when it is in control. The AIs that come with the game all poll only while it is their turn. This is not necessary, but commands sent when it is not that player turn will be rejected.

When it is your turn to go, a runTurn() method that you have overridden will be called. Also, somewhere in your stack of Event stuff when you poll will be a Start Turn Event. It is important to process these events and constantly poll for events after you take and action while it is your turn to make sure you maintain as accurate a game data state in sync with the game as possible.

With runTurn(), you have a thread of control. When you return your turn will be over. All of the AI Players that come with the game are single threaded, but there is no restriction to your AI having threads. Sending commands is not a thread safe activity, and it is highly recommended you strictly control commands and event polling.

During the control of runTurn() is when the magic happens, and you can evaluate the situation based on the data at hand, make commands accordingly. Commands include things like moving units and giving production orders.

Look Over The Common Code

The player will construct a COrder object with the appropriate information and then use one of the order methods in the AIPlayer class. If you look at the method

AIPlayer::setAndExecuteUnitOrder for example, you will also observe that ultimately an event poll is performed before the result is returned. So if you give the unit an order to move one step north, then the event changing the units location, movement points, etc., will already have been processed before the result is returned. Also, if the unit has died, that event was already received. So it is important to not react to events with commands as this could cause a recursive situation. Evaluate the next step after the command has returned.

Events beyond the State and Start turn events generally consist of unit and production updates, death messages, and things like drain and turn values.

A Few Things To Remember About Commands And Events
When responding to Events, it is important that all events are first processed to reflect data changes before proceeding to issue another command, otherwise stack overflow or null object reference errors things can happen. For example: Be aware that the unit you just ordered to move may now be dead. Check accordingly.

If you create an unhandled exception in your AI Player runTurn() method, it will end your turn.

You may also want to be aware of things like the insanity complex, where you issue a command and get a result, feeling like you can do more you issue a command expecting a different result, but it is always the same. Example: “I have movement points, Move North”, “No It's Blocked”, “I have movement points, Move North”, No It's Blocked”, ...

AI Player Saving And Loading
At some point, your AI Player is going to be asked for Save Data. If you are derivd from the AIPlayerData class, it will save the underlying data. The extra data that your player class created and would like to save should be returned at this time.

The player is given a CEncodedOutputBufferI object to which he can write the saved data in an xml-type style. You are responsible for making sure in the end all of your elements are well formed, otherwise they will not reload properly. Make sure to review some of the existing AI source code for examples when released.

Then restoring the AI is a two step process. First, when the new AI Player object is constructed, you receive a CEncodedInputBufferI in the constructor which contains the data you had previously saved in the exact order which you had saved it. A second call is then made if there are further adjustments you need to make before resuming play.

What Happens When I, The AI Player, Die?
If you lose (or win) the game will end and runTurn will never be called again.

I don't know much about programming, but I would like to fix something in the code
Don't be afraid. Download that compiler and install it, and set up a project. You will need to get your feet a little wet, but nothing too distressing. Just look at the code until you believe you have found the area you would like to touch, and make a small change. This could be as

simple as setting the name for the player to use. The branch out from there. Don't forget to ask questions of the community.

The Example AI Player

I have also made available the “ExampleAI” player which currently does nothing. But it is a good example of how a player and a factory is setup. It can be found at

http://killerbeesoftware.com/edce/code/