Chat Diamond

The_Cowboy

June 25, 2023

Contents

BIG FAT NOTE: Chat Diamond is in development stage. Please embrace yourselves for unexpected features and bugs here and there punctuated by accessed none type of logs. If possible please raise an issue at GitHub!

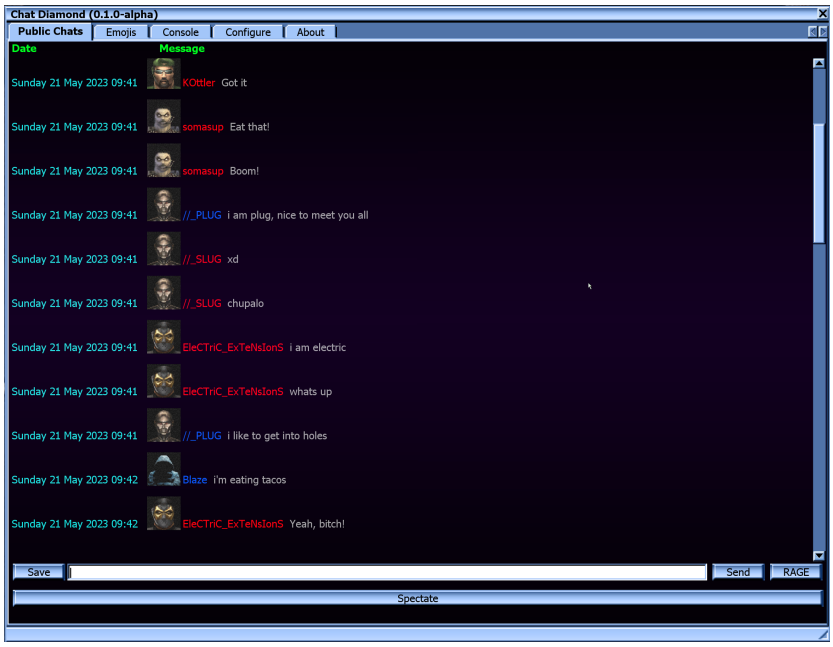

Figure 1: Chat Diamond 0.8 Chat.

1 Introduction

Chat Diamond (version 0.8) is a purely client-side mod for Unreal Tournament G.O.T.Y (UT99) which replaces the default UT console with a more appropriate one. This functionality allows the achievement of the following

- Gathering of raw messages delivered to the console and relevant categorization, along with chat message separation, of them.
- Introduction of appropriate web-query for translation to local or demand of the language.

1.1 Chat Window

Figure 1 shows the current form of the chat window. The following features are supported

- Display of sender's avatar face.
- Static emojis and animated emotes.
- Display of Date and Time (long format for now).
- Display of the Server name for reference purposes.

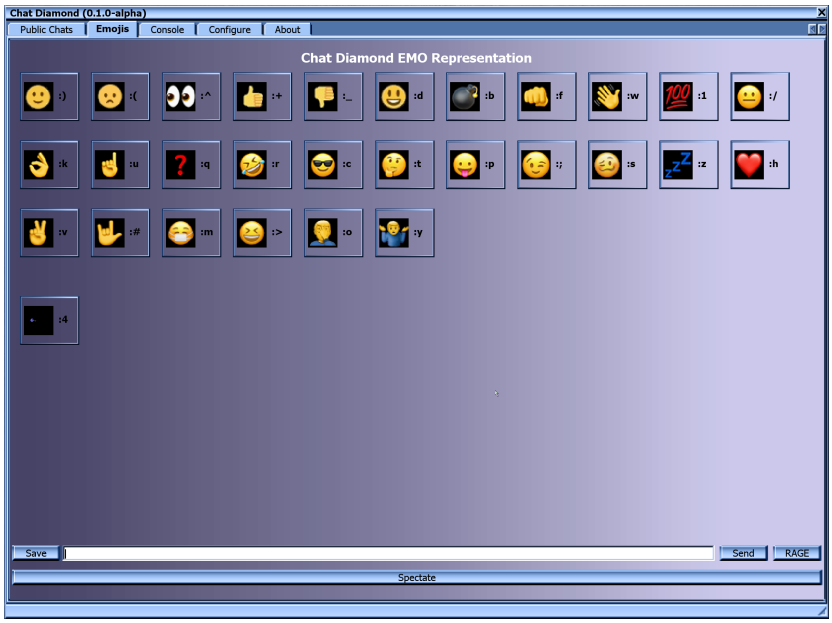

Figure 2: Chat Diamond 0.8 Emojis.

- Ability to copy text and IP address (both game and web server) from chat messages.
  - IPs of the format xxx.xxx.xxx.xxx are considered web server IPs.
  - IPs of the format xxx.xxx.xxx.xxx:xxxx are considered game server IPs.
- Ability to display hyperlinks with clickable feature for relevant navigation and mouse cursor distinction.
  - On clicking web or game server IP, the player can reach the relevant website (http:// only) or UT99 game server.
  - On clicking http://something.com or https://something.com, the player can reach the desired site, if valid URL is posted.

1.2 Emoji Window

In order to display complete list of supported emojis (and emotes), collectively called *Emos*, the Emoji window 2 (which was texture based for UT Chat) has been written almost completely in unreal script (save the textures). This allows the support for the following

- A curation of selection page for framed Emos.

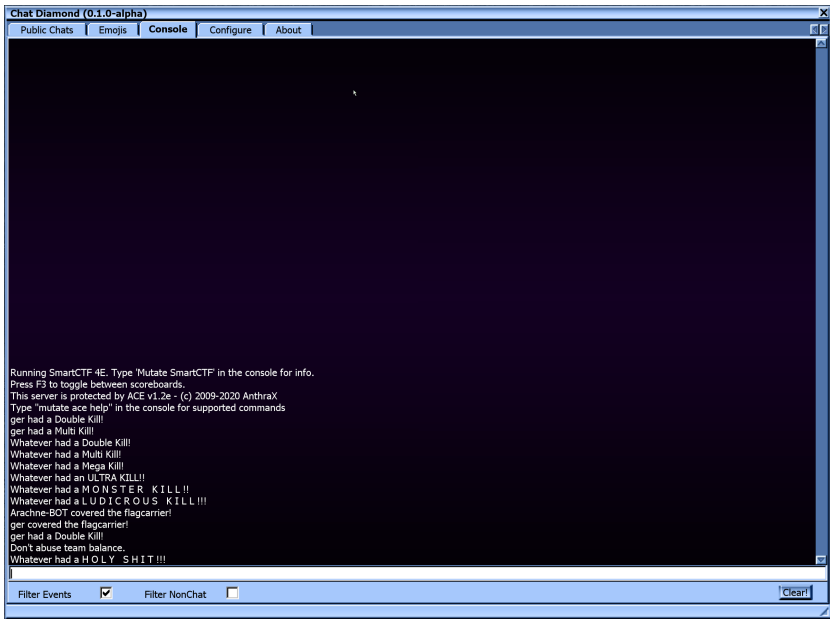

Figure 3: Chat Diamond 0.8 Console.

- An interactive way of display, which depresses the frame being hovered with a sound and generates a distinct clicking experience.
- Emoji and Chat window text areas are synced, meaning, whatever you write or select in one window, which gets registered in the text area, is copied over to the next one.
- My personal favorite, the scrolling capability.

1.3 Console Window

The console now has the following features

- Non-chat filter to filter out server or mutator advertisements, in the console. Although SmartCTF cover and seal messages may also get filtered.
- Event filter censors death messages corresponding to the weapon used.
- A clear button to clean or reset the console.
- A working status bar linked with console configuration modifiers.

1.4 Configure Window

Chat Diamond provides “real time way” to configure various user configurable parameters (figure 4). This implies instant relevant experience on toggling or

tuning those parameters, for instance tuning the RGB values for background color of Public Chat window and Console window results in the relevant change in the background of Configure Window itself. Allow me to provide relevant details for configurations

- *Background Color*: Set the RGB values to come up with a gradation color for backgound. The setting can be instantaneously applied to Public Chats widow and Console window
- *Chat Binding*: Click on the box and the key pressed next shall be assigned for opening Public Chats window.
- *Animation Speed*: Real time adjustmet, by dragging the calibration slider, as per personal taste.
- *Emo Size*: Real time adjustment for setting the size of emoji's and emotes simultaneously.
- *Message Arrival Sound*: Check if you want telegram style sound to be played on message dlivery.
- *Open Chat At The End*: Check if you to open the chat window 1.1 when the match ends.
- *Load Messages*: x is the number of messages in history that you want Chat Diamond to load[1].

1.5 More Pages

Please keep in mind that this is very initial stage of Chat Diamond (0.8). The pages to configure various properties of chat messages and date format shall be made available as we progress with the development cycle.

2 Installation

Chat Diamond is developed and tested with UT99, **469c client**. Although the mod should work on previous versions, I take no pains for maintaining that code simply because I feel that we all should work towards pushing forward the community effort in driving the game (and UE 1) towards modern experience. So by not supporting the code for earlier versions I am supporting the later versions.

With that being written, feel free to send pull requests to Chat Diamond repository, even for earlier game versions.

[1] Chat Diamond has the capacity to store virtually all the messages without the need of erasing history. Loading of more messages would be slow though. Optimal number is 20 - 40 after which you can start noticing the slow down. For complete history of messages, check out ChatDiamond.txt which contains chat and relevant metadat in json strings.

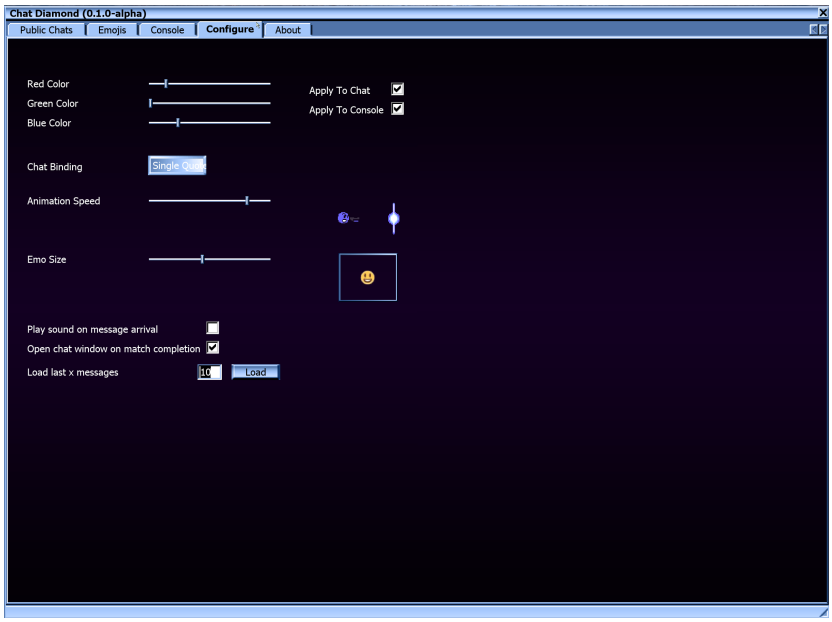

Figure 4: Chat Diamond 0.8 Configure.

Chat Diamond is essentially a console (and not meant for the server). So for installation do the following

- – **Linux**: Place the ChatDiamond.so, ChatDiamond.ini, ChatDiamond.txt, and ChatDiamond.u in System directory.
  – **Windows**: Place the ChatDiamond.dll, ChatDiamond.ini, ChatDiamond.txt, and ChatDiamond.u in System directory.
- Open UnrealTournament.ini and find the section

```
[Engine.Engine]
GameRenderDevice=VulkanDrv.VulkanRenderDevice
AudioDevice=ALAudio.ALAudioSubsystem
...
Console=UTMenu.UTConsole
...
```

- Modify to

```
...
Console=ChatDiamond.CDUTConsole
...
```

- And that is it! You can now summon console by usual '∼' key.

3 Version Histroy

In this section I am presenting the history of Chat Diamond for general awareness.

Changelog

0.8 The_Cowboy (2023-xx-xx)

Added

- Configure Window
- Several emotes
- New server join information in chat window
- Linux support
- Work flow for users to add custom emos

Fixed

- Chat Window text scaling
- Emoji Window emojitext offset (69b7abbc)
- Chat Window emo textures translucency (6e105d2c)
- Emoji Window emo frames mouse hover detection (501e5852)
- Addressed several accessed none warnings in the function LocateChatFaceTexture by commit 1dda53b0

Changed

- Using json format for chat metadata, instead of ":" deliminator which clashed with names
- Using linked lists with dynamic loaded textures to avoid loading each frame
- Message history is non-truncable[2]. Number of last x messages to be shown is configurable.
- Using text file instead of ini for dumping chat metadata (f059be81)

0.1.0-alpha The_Cowboy (2022-12-25) — Initial alpha

[2]Meaning no need to delete messages as they are dumped in .txt file which is reservior for virtually unlimited messages (for years to come).

4 A note on UT Messaging

Chat Diamond's functionality is based upon the tenet "A complete client side user interface with no dependencies with the server". This would imply conforming to the UT standards and any server not conforming to the standards, if so, is not worth visiting.

Based upon the above motivation we have a choice to make between replacing the client HUD or UT console, in order to gather the UT messages client side. My first instinct was to replace the HUD because that would give more precise classification of the messages, in the sense Epic wrote the code, like so (line 1499)

```
1  // Entry point for string messages.
   simulated function Message( PlayerReplicationInfo PRI, coerce
       string Msg, name MsgType )
3  {
       local int i;
5      local Class<LocalMessage> MessageClass;

7      switch (MsgType)
       {
9          case 'Say':
           case 'TeamSay':
11             MessageClass = class'SayMessagePlus';
               break;
13         case 'CriticalEvent':
               MessageClass = class'CriticalStringPlus';
15             LocalizedMessage( MessageClass, 0, None, None, None,
       Msg );
               return;
17         case 'DeathMessage':
               MessageClass = class'RedSayMessagePlus';
19             break;
           case 'Pickup':
21             PickupTime = Level.TimeSeconds;
           default:
23             MessageClass = class'StringMessagePlus';
               break;
25     }
      ...
```

The basic premise of many cheats providing aim assist is to modify the HUD client side and display the adversary's position behind the wall or a radar. This is the reason why modern anti-cheats are not receptive of clients modifying HUD, and thus, work with the concept of "whitelisting".

If the notion of "cheat-anticheat" interplay exists in the UT standards, then that simply doesn't fit with the tenant we began with. We don't want to be sending request to server administrators for specifically whitelisting Chat Diamond which will be reverted by cold silent treatment to say the most and some naggy jibber jabber about the unknown-ness of Chat Diamond, especially when the mod is not that popular.

So with this thinking in mind, the better option would be to replace the client console, which I haven't seen any decent anti-cheat prohibiting. This not only gets the client out of hook for sending whitelisting requests but also provides a powerful mechanism to mould the console, specifically for generic needs (see the concept of *filters*)[3].

4.1 A minor PhD on console messages

We start with the code

```
event ClientMessage( coerce string S, optional Name Type, optional
    bool bBeep )
{
    ...
    if (Player.Console != None)
        Player.Console.Message( PlayerReplicationInfo, S, Type );
    if (bBeep && bMessageBeep)
        PlayBeepSound();
    if ( myHUD != None )
        myHUD.Message( PlayerReplicationInfo, S, Type );
}
```

and

```
event TeamMessage( PlayerReplicationInfo PRI, coerce string S, name
    Type, optional bool bBeep )
{
    if (Player.Console != None)
        Player.Console.Message ( PRI, S, Type );
    if (bBeep && bMessageBeep)
        PlayBeepSound();
    if ( myHUD != None )
        myHUD.Message( PRI, S, Type );
}
```

It is clear that both the console and HUD should receive same messages with no discrimination (unless if coder specifically demands, as mentioned in 4). The console would display them, in what I would like to think, *raw* form. The *diff*, between console and HUD, is shown in the figure 5

Please note that Chat Diamond's console is not much different from default UT99, visually. The changes are how Chat Diamond interprets the raw messages thrown and utilize them in a constructive way. So this is where magic code comes in. In order to understand that let me first demonstrate UT messaging by a table[4].

[3] Althought some servers may prohibit the use of custom consoles. For that you may have to specifically send the whitelist request. You may want to report the relevant console class, which, for Chat Diamond, should be "ChatDiamond.CDUTConsole".

[4] The "messages" for instance: 'plushie was smacked down by Mental_Instituitions's Rocket Launcher', are actually strings which can be seen in the line 117 of the code. This also shows how some death messages can be suppressed in the console thus differing from HUD.

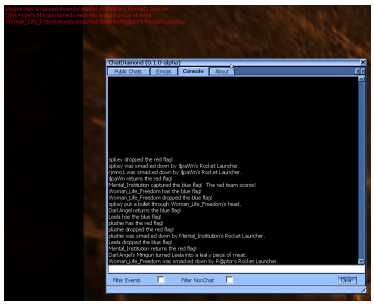

Figure 5: UT99 HUD with Chat Diamond console

The mapping to relevant arguments is like so in Table 2.
The tables need to be explained and pondered.

5 Native Coding

Chat Diamond is a native mod, meaning, a program invoking C++ functions. At the time of writing, minor usage of the library regular expressions is (or should be) evident from the following code

```
native final static function string SpitIpFromChatString(string
    Message, out int ICategory);
```

To understand the implementation of the above unreal script function, please see the code displayed as follows

```
void ACDDiscordActor::execSpitIpFromChatString(FFrame& Stack,
    RESULT_DECL)
{
  guard(ACDDiscordActor::execSpitIpFromChatString);
  P_GET_STR(Message);
  P_GET_INT_REF(IPCategory);
  P_FINISH;

  std::smatch Match;

  // The nice way out, from game and web server mix, seems to make
    an assumption that, in context of UT, the gameserver IP
  // be given by complete port number.

  // https://github.com/ravimohan1991/ChatDiamond/issues/1#
    issuecomment-1356906185
  std::regex GameIPMould("\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d
    {1,3}\\:\\d{1,4}");
  std::regex WebIPMould("\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}"
    );

  std::wstring WString(*Message);
```

```
  std::string SampleString(WString.begin(), WString.end());

  std::string IPString;

  // Look for game IP
  if (std::regex_search(SampleString, Match, GameIPMould))
  {
    for (auto Tempo : Match)
    {
      IPString = Tempo.str();
      break;
    }
  }

  if (!IPString.empty())
  {
    std::wstring WideIPString = std::wstring(IPString.begin(),
    IPString.end());
    *IPCategory = 0;
    *(FString*)Result = WideIPString.c_str();
    return;
  }

  // Look for game IP
  if (std::regex_search(SampleString, Match, WebIPMould))
  {
    for (auto Tempo : Match)
    {
      IPString = Tempo.str();
      break;
    }
  }

  if (!IPString.empty())
  {
    std::wstring WideIPString = std::wstring(IPString.begin(),
    IPString.end());
    *IPCategory = 1;
    *(FString*)Result = WideIPString.c_str();
    return;
  }

  std::wstring WideIPString = std::wstring(IPString.begin(),
    IPString.end());
  *IPCategory = 2;
  *(FString*)Result = WideIPString.c_str();

  unguard;
}
```

There are few remarks

- The C++ code isn't the ordinary one. That is punctuated with variety of macros, which generate, what I would like to call, the first hints of reflection system.

- In order to linearize the story of native coding and start producing practical code, we need to first learn few "game-dev" terminologies and practices.

5.1 Why Native Code?

Unreal script is a highly managed language which comes with limitations, which is characteristic of any managed language actually, such as slow(er) runtime (when compared with C++), ability to fuse with well defined C++ applications (like Discord), and lack of programmers' choice or taste. Don't get me wrong, the language is best for what it does, scripting mods for the game. Since this is 2020 decade, along with native support, there should be no 2000's environment limitation, especially with the new community patches.

5.2 Build System

I have been using combination of Wot Greal (unrealscript code) and Microsoft Visual Studio 2022 (C++ code) IDEs for developing Chat Diamond. Therefore I have organized the repository with the conforming hierarchical structure 6.

What we have achieved by doing so is a reconciliation of UT99's unreal script modding environment with C++ dev environment yielding a version controlled (a Github repository) avidity with UT's root folder. All the C++ code goes to the folders marked with, well, C++ and unreal script code goes to top level ChatDiamond \Classes directory.

The Core is the directory containing "header-only" library elements, meaning .h files and the already compiled binary Core.lib. Few remarks are to be worded

- Such libraries, which are prebuilt and whose implementation is hidden with only headers (the API) visible, are known as "`Interface`" libraries in CMake's scope. Engine is another example of directory containing such `Interface` library. Also both of the Engine and Core directories contain the directories Inc, Src, and Lib sub directories.

- From research, Core.lib and Engine.lib, which are static libraries, seem to be the "basic necessities" for linking with a native mod.

- In the hierarchy, I am only showing the top level CmakeLists.txt. If you are familiar with the build utility, you can comprehend the presence of one each inside ChatDiamond, Core, and Engine directories.

- Currently, I am developing on and for Windows (should be evident from GenerateProjectFiles.bat). It goes against my philosophy of cross-platform development[5] given the 32-bit jugaad for Linux libraries, I may do something in future. MacOS seems like a workable goal since I own a MacBook Pro.

[5] For solid enough work please see Karma.

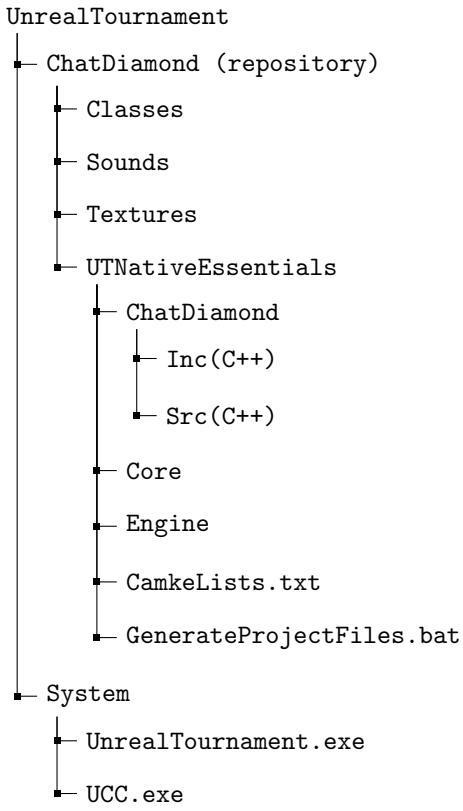

Figure 6: Chat Diamond project hierarchy.

Make sure that your machine has CMake installed. For uninitiated programmers (in my POV read premakers), let me supply a crash course on CMake, for our purposes here.

5.2.1 UT and CMake

In a C++, or any good programming language, there is a proper tool-chain which pipes the human written code to Turing machine understandable arrangement of bits (zeroes and ones). This usually involves lot of garbage intermediary files which most of the part are useless, unless you suddenly, out of the blue, decide to do scavenging some day.

The responsibility of a good build system is to understand such redundancies and provide a manageable way of dealing with chunks of huge memory files. Let me demonstrate by simple logistics.

At the time, the Chat Diamond C++ component has about three to four MB worth of code files (.h and .cpp). Visual Studio 2022 generates a solution build of about 144 MB worth of memory and the output is ChatDiamond.dll which is about two MB. The intermediary files are for IDE's (Visual Studio for instance) and compiler (MSVC or Clang) use only and therefore should be dealt separately by a decent build system[6].

Given the variety of compilers, with even more variety of platforms, CMake is the most suitable build system and friend that a programmer can, well, make and use. Along with the clear distinct folder generation for intermediate files, CMake comes with high degree of configurability, required for build, and compatibility with known compilers[7]. CMake supports the Windows, Unix, and Linux based platforms.

UT99 was officially released and supported on multiple platforms. The community clearly intends on keeping that flame intact. Therefore a rocket scientist isn't really needed to gauge the obvious link between UT and CMake.

With enough convincing we are now in a position to understand the organization of this project (or some UT project organization, in general). This may serve as a template. Consider the hierarchy 6. Then the code

```
####################
# Link Directories
####################
link_directories(ChatDiamond Core/Lib Engine/Lib)

#############################################
# Target Definitions / Custom Modding Stuff
#############################################
add_subdirectory(ChatDiamond)
target_include_directories(ChatDiamond
```

[6] Actually this is the reinforcement of the concept of Universal Turing Machine. That a good compiler should be able to generate same set of instructions, i.e. a program, which should behave that same way, no matter a laptop or desktop or smart phone. Please see von-Neumann architecture for a lucid connection between Universal Turing Machine concept and PCs or Macbooks.

[7] For a comprehensive list, please visit here.

```
11      PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}/Core/Inc
13      ${CMAKE_CURRENT_SOURCE_DIR}/Engine/Inc)

15

########################
17 # Target Linking Rules
########################
19 target_link_libraries(ChatDiamond PUBLIC Core Engine)
```

becomes relevant as follows.

In line 4, we are telling CMake to generate relevant project files (for MSVC or QTCreator) such that the linking can be done with the libraries stored in the Core\Lib and Engine\Lib directories[8].

Next from line 9 to 13, we intend to tell CMake to add the native mod project, specified by CMakeLists.txt present in sub-directory ChatDiamond.

Next we proceed towards the file mentioned in previous paragraph (taking a level deep plunge in the folder hierarchy). The code is

```
1  cmake_minimum_required(VERSION 3.0.0)

3  project(ChatDiamond)

5  file(GLOB CD_HEADERS Inc/*.h)

7  add_library(ChatDiamond SHARED Src/ChatDiamondNative.cpp ${
       CD_HEADERS})

9  target_include_directories(ChatDiamond
       PRIVATE
11         ${CMAKE_CURRENT_SOURCE_DIR}/Inc
       PUBLIC
13         $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/Inc>
           $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
15 )

17 target_compile_definitions(ChatDiamond PUBLIC ChatDiamond)
```

Line 3 declares the mod project name. Then we specify all the .h files by collectively assembling them with the command `file` and labeling them *CD_HEADERS*. Then we specify the library name via `add_library` command in the first field, followed by the type (`SHARED` means dynamic library) followed by all the source (.cpp) files. Then we mention headers so that they may appear nicely in the project. For instance, in XCode the project ChatDiamond looks like

Furthermore we apprise the CMake that the target, ChatDiamond library, needs the includes of Inc and provides the same folder as API for various libraries

[8]Note that directories are relative to the current CMake operation directory which is basically the location of the CMakeLists.txt with above code.

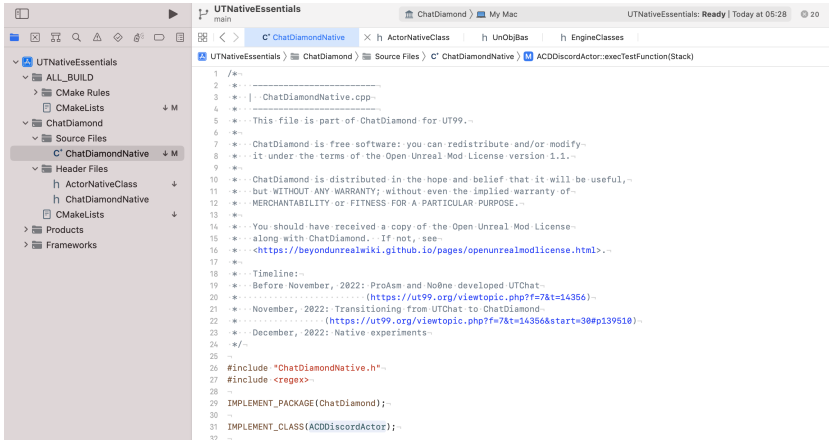

Figure 7: A Chat Diamond XCode project.

to utilize the functionality. Finally we set the public usable name, which is same, of the library.

Now we come back to the topmost CMakeLists.txt, line 10, which can be regarded the continuation of `target_include_directories` of the previous one. Here we want to tell CMake to include the library directories (the basic essentials) Core\Inc and Engine\Inc in our ChatDiamond project and in line 19 we apprise CMake to link those libraries thus generating the wholesome ChatDiamond.dll, if we are on Windows[9].

5.3 C++ for UT

We are basically in the territory of interface between unreal script and C++ for the game UT99. Remember the C++ standard that the UT99 was written in could have been C98 (or SO/IEC 14882:1998) and unreal script could have had influences.

In 2022, we can be using any standard ranging from C++11 to C++20. In CMake above, we defined the standard like so

```
1 set(CMAKE_CXX_STANDARD 11)
```

This is what we mean by "modern", the ability to take good parts from the standards in history of C++ evolution. With this modern approach, we can leverage the latest XCode or Microsoft's Visual Studio features (including smarter intellisense) and produce the code that can and will work in future.

[9]On Linux based OS, the dynamic library has the extension of ".so", which stands for "shared object".

First a shout out to fellas at OldUnreal (and the thread) for the native "Hello World" tutorial and Anthrax for UDemo, a working native mod with useful UT oriented C++ information.

With credits out of the way, let me focus upon the juice. The aim is to write a C++ function which can be called from unreal script and which can print

```
Log: Hello World! S=Test, I=888
```

in the logs. The idea is to compile a uscript[10] code (.u package) which makes the call to native function declared within uscript like so

```
native final function bool TestFunction(string S, int I);
```

written in CDDiscordActor.uc. We need to define this routine in C++, the language the Engine is written in. That requires punctuation of .h/.cpp files with lot of macros dedicated towards making Engine understand and adapt to uscript capabilities (modularity, swift iteration, and prototyping functionality[11]).

We start with the following code analysis

```
#if ((_MSC_VER) || (HAVE_PRAGMA_PACK))
#pragma pack (push,4)
#endif

#ifndef NAMES_ONLY
#define AUTOGENERATE_FUNCTION(cls,idx,name)
#endif
```

Line 1 is preprocessor instruction which says that if I am using MSVC compiler and if `HAVE_PRAGMA_PACK` is set then the alignment of the aggregate members[12] is defined like so

Definition 5.1. Align structure or class members on 4-byte boundaries, or on their natural alignment (see APPENDIX C) boundary[13], whichever is less.

`push` is used to make the alignment contextual. Therefore you shall find the closing counterpart like so

```
#if ((_MSC_VER) || (HAVE_PRAGMA_PACK))
#pragma pack (pop)
#endif
```

[10] I am using unreal script and uscript interchangeably in this document.
[11] You can literally use notepad for uscripts if you know what you are doing.
[12] For revisiting the notion of members, please see APPENDIX B
[13]

Thanks to stackoverflow discussion. I don't quite yet know the relevance to uscript, all I can think is maybe this is UT99's convention of dealing with the electronics of 1999 or something. Also anth (2022) has justification like so.

Lines 5 to 7 are the instructions that seem redundant for our purposes because we haven't used or don't need the macro for printing the log in my foreseeable way.

Now we come to the actual definition of the class. As per UT's naming convention all actor names start with 'A'. Hence we have

```
1 class ACDDiscordActor : public AActor
{
3 public:
  DECLARE_FUNCTION(execTestFunction)
5  DECLARE_CLASS(ACDDiscordActor, AActor, 0, ChatDiamond);
  ACDDiscordActor();
7 };
```

Here we need to first understand the `DECLARE_FUNCTION` and `DECLARE_CLASS` macros. For the former the definition is like so

```
1 #define DECLARE_FUNCTION(func) void func( FFrame& TheStack,
    RESULT_DECL );
```

where `FFrame` is the data structure defined like so. Anth already has full fledged explanation about the inner working and this is how the train of C++ logic runs

- The `UObject::ProcessEvent` is the entry point for unrealscript code, see the `UFUnction` code for instance, where the C++ routine sets a top-level `FFrame`. Of course, the implementation of `ProcessEvent` is hidden as of now.
- The `ProcessEvent` then sets up stack for uscript function parameters and local variables.
- The function parameters are then initialized by copying values from `void* Params` buffer to the top level `FFrame`.
- We then execute opcodes one by one using this top-level FFrame as our uscript execution stack until we reach an `EX_Return` opcode or until we hit the end of the function.

Note: I don't know what opcode is.

`DECLARE_CLASS` is defined like so

```
1 // Declare a concrete class.
#define DECLARE_CLASS( TClass, TSuperClass, TStaticFlags, ... ) \
3  DECLARE_BASE_CLASS( TClass, TSuperClass, TStaticFlags,
    __VA_ARGS__ ) \
```

```
  friend FArchive &operator<<( FArchive& Ar, TClass*& Res ) \
    { return Ar << *(UObject**)&Res; } \
  virtual ~TClass() noexcept(false)   \
    { ConditionalDestroy(); } \
  static void InternalConstructor( void* X ) \
    { new( (EInternal*)X )TClass; } \
```

where **DECLARE_BASE_CLASS** is defined like so

```
#define DECLARE_BASE_CLASS( TClass, TSuperClass, TStaticFlags, ... ) \
public: \
  /* Identification */ \
  enum {StaticClassFlags=TStaticFlags}; \
  private: static UClass PrivateStaticClass; public: \
  typedef TSuperClass Super;\
  typedef TClass ThisClass;\
  static UClass* StaticClass() \
    { return &PrivateStaticClass; } \
  void* operator new( size_t Size, UObject* Outer=(UObject*)GetTransientPackage(), FName Name=NAME_None, DWORD SetFlags=0 ) \
    { return StaticAllocateObject( StaticClass(), Outer, Name, SetFlags ); } \
  void* operator new( size_t Size, EInternal* Mem ) \
    { return (void*)Mem; }
```

So, on macro expansion, the code takes the following nice lucid form

```
class ACDDiscordActor : public AActor
{
public:
  ACDDiscordActor();

public:
  void execTestFunction(FFrame& TheStack, void* const Result);

public:
public:
  /* Identification */
  enum { StaticClassFlags = 0 };
  private: static UClass PrivateStaticClass;

public:
  typedef AActor Super;
  typedef ACDDiscordActor ThisClass;
  static UClass* StaticClass()
  {
    return &PrivateStaticClass;
  }
  void* operator new(size_t Size, UObject* Outer = (UObject*)GetTransientPackage(), FName Name = NAME_None, DWORD SetFlags = 0)
  {
```

```
    return StaticAllocateObject(StaticClass(), Outer, Name,
    SetFlags);
  }
  void* operator new(size_t Size, EInternal* Mem)
  {
    return (void*)Mem;
  }

  friend FArchive& operator<<(FArchive& Ar, ACDDiscordActor*& Res)
  {
    return Ar << *(UObject**)&Res;
  }
  virtual ~ACDDiscordActor() noexcept(false)
  {
    ConditionalDestroy();
  }
  static void InternalConstructor(void* X)
  {
    new((EInternal*)X)ACDDiscordActor;
  }
};
```

Now we come to the definition of the function done like so

```
void ACDDiscordActor::execTestFunction(FFrame& Stack, RESULT_DECL)
{
  guard(ACDDiscordActor::execTestFunction);
  P_GET_STR(S);//Get the first parameter
  P_GET_INT(I);//and the second
  P_FINISH;//you MUST call this or it will crash.

  GLog->Logf(TEXT("Hello World! S=%s, I=%i"), *S, I);//Log output
    and use printf format.
  //You may also use debugf(TEXT("Hello world!")) since it may be
    easier to remember.
  *(UBOOL*)Result = true;// Return true to UScript, this is how you
    return a result. You cast your result into "Result" --
    whatever it may be.
  unguard;
}
IMPLEMENT_FUNCTION(ACDDiscordActor, -1, execTestFunction);
```

Most parts have been explained in comments. I shall focus on the following

- `P_GET_STR` is defined like so

```
#define P_GET_STR(var)                  FString var;
                  Stack.Step( Stack.Object, &var    );
#define P_GET_INT(var)                  INT   var=0;
                  Stack.Step( Stack.Object, &var    );
```

is used for collecting the arguments of our uscript function `TestFunction(string S, int I)` and order is indespensible.

- `P_FINISH` is defined like so

```
#define P_FINISH                    Stack.Code++;
```

A Messaging Tables

Console Owner State	Death Messages [14]	Server Announcements[15]	Talk TeamTalk
Multiplayer Spectator	plushie was smacked down by MI's Rocket Launcher	Type !cg to visit combogib grapple server	A: Self Sent-RN[16]:Hola B. By Player-RN:SN[17]:Hola C. By spectator[18] SN:Hola

Table 1: Table of Messages.

B Aggregation

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

```
1  class Address
   {
3      public:
       string addressLine;
5      Address(string addressLine)
       {
7          this->addressLine = addressLine;
       }
9  };
   class Employee
11 {
       private:
```

Console Owner State	Death Messages [19]	Server Announcements[20]	Talk TeamTalk
Multiplayer Spectator	MessageType: DeathMessage? PRI: Local?	MessageType: Event PRI: Local	A. MessageType: Event PRI: Local B. MessageType: Event PRI: local C. MessageType: Event PRI: Local
Multiplayer Player	Message Type: Event PRI: Local?	Message Type: Event PRI: None	A. MessageType: Say PRI: Local B. MessageType: Say PRI: SenderPRI C. MessageType: ? PRI: ?

Table 2: Table of argument types.

```
13      Address* address;  //Employee HAS-A Address

15      public:
        int id;
17      Employee(int id, Address* address)
        {
19          this->id = id;
            this->address = address;
21      }
        void display()
23      {
            cout<<id <<"\n";
25          address->addressLine;
        }
27  };
```

In the code above, line 13 is the example of aggregate member `address`.

C #pragma pack

First a definition

Definition C.1. Natural alignment roughly means data's memory address is multiple of data size.

For instance, in a 32-bit architecture, the data may be aligned if the data is stored in four consecutive bytes and the first byte lies on a 4-byte boundary. Thus if there is a class like so

```
1   class Test
    {
3       char a;
        int b;
5   };
```

then, if natural alignment is done (say 32 bit, or x86, architecture), the size of `Test` would be 8 bytes with the following breakdown

- 1 char byte + 3 bytes of padding
- 4 bytes for int