Far Cry Engine Overview

Version 1.0

3 March 2005

CONTENTS

ARCHITECTURAL OVERVIEW

Far Cry engine is made up of several different modules, implemented on the PC platform as separated .DLLs.

Cry3Dengine

The main actions performed by this module are the following:

- Loading and rendering of terrain
- Loading and rendering of static CGF objects and handling rendering of animated objects
- Loading and rendering of indoor areas

Here the term rendering means high-level rendering, in fact the low-level rendering (vertex buffer creation, shaders etc.) is managed by the **XRender** module.

Animated (.CGA) objects are loaded and handled by the **CryAnimation** module.

Interface files reference:

- I3DEngine.h
- IStatObj.h
- ILMSerializationManager.h

For more information about .CGF format, check out the following included in the SDK:

- CGFDump VC++ project
- Max and the Maya exporters.

CryAISystem

- Loading of AI data (graph, nodes) exported from the editor
- Loading of behaviors from script file
- Handling of enemies AI at runtime.

The graph nodes for indoor and outdoor sections are placed in the **CryEditor**, and the triangulation data is created upon exporting time.

Interface files reference:

- AgentParams.h
- Iagent.h

- IAISystem.h

Additional Documentation:

- AI FAQ.pdf
- AI Manual.pdf
- AI scripts
- AI section of the Editor manual

CryAnimation

- Loading and rendering of animated models.
- Attachment of static objects to animated models.

As with the **Cry3Dengine**, the actual manipulation of platform dependent rendering data (DirectX, OpenGL, NULL) is abstracted by the **XRender** module.

Interface files:

- IcryAnimation.h
- Ibindable.h

For exaustive documentation regarding animated objects:

- FullyCustomizableCharacters.pdf
- GuidilineForCharacterPhysicsAuthoring.pdf
- NamingCharacterAnimations.pdf
- ResourceCompiler.pdf
- CCGDump VC++ project
- Max and Maya exporters
- animated samples in the sources/animations folder of the SDK

CryCommon

- Included headers and interfaces for all modules.
- Math library
- Platform specific defines.

CryEntitySystem

- Creation and handling of the entity structure.
- Manage and compute LipSync and Facial Expression Features (which shouldn't be present in this module).

The Entity structure is the base class of any game object placed in the editor.

Enemies, crates, vehicles, elevators, triggers, etc. are all derived from the entity; basically any object which is not a pure rendering object, such as a static tree or a rock, is created as an entity.

The description of all entities properties as well as how they relate with each other (triggers, areas etc.) are exported in the XML level file by the **CryEditor** and loaded by the **CryGame** module, whereas static objects data information (trees, rocks etc.) are exported into a separate file for **Cry3Dengine** usage; all necessary files for each level, such as those described here, are included and exported into a single Level.pak file. Entities include sounds, which are handled by the **CrySoundSystem**.

It contains some basic data structures used by **CryPhysics**.

Interface files:

- IEntitySystem.h

Additional documentation:

- The Editor Documentation, search for the Entity keyword.

CryFont

- Loading and drawing of Unicode true type fonts

Interface files:

- Ifont.h

CryGame

- Loading of game levels
- Single player and multi-player game code and logic
- Handling of game client/server relationship
- Player, AI and vehicles game code
- Weapons code
- Game serialization
- Loading and handling of materials
- Game modding functionalities
- Glue with game scripts

- Time demo recording functionalities
- Game localization code
- Real-time game editing logic
- Handling of cut scenes, menus, UI

This is the biggest module and handles everything related to the game. It interfaces with almost all other subsystems as necessary, and it runs the main loop of the game. The crygame.dll can be override by a MOD. A big chunk of the game properties and game rules for both single player and multiplayer, as well as specific behaviors for each game object, are defined by script. Scripts are loaded and handled by the **CryScriptSystem** module. Different properties for each game script are setup in the **CryEditor** and exported to the XML file. **CryEntitySystem** creates single entities, while relationships between each other are setup by **CryGame**. This can also be done in real-time when the game is in real-time editing mode within **CryEditor**.

Interface files:

- Igame.h

Additional documentation:

CscriptObjectGame.pdf

FC editor manual.pdf

FCMG.pdf

Weapon tweak guide.pdf

Weapons tutorial.pdf

FCFAQ.pdf

Vehicle Manual.zip

FC MenuSkinningTutorial.pdf

Creating Localized Text.pdf

CryInput

- Handling of input events from keyboard, joystick and mouse
- Action maps, key bindings

CryGame uses this module to handle player's movements, to navigate through menus, to setup key bindings etc.

Interface files:

- Iinput.h

CryMovie

- Handling of cut scenes.

Cut scenes are created within **CryEdit**, which communicates with this module at runtime during cut scene creation and playback. **CryGame** will use this module to play cut scenes during game play when required.

Interface files:

- IMoviesystem.h

Additional documentation:

- FC editor.pdf (the movie editing section).

CryNetwork

- Handling IP/LAN network connections via UDP packets
- Client/Server low level communication
- Punkbuster
- UBI.COM

CryGame uses this module to handle the multiplayer part of the game.

Interface files:

- INetwork.h

Additional documentation:

- How to use the built in Internet simulator.pdf
- Tools to tweak FarCry MP.pdf

CryPhysics

- Handling of dynamic and static collision data

- Collision detection and response
- Handling of basic collision primitives and rigid bodies, wheeled vehicles, ropes etc.

Static, pure rendering objects and terrain are physicalized when loaded by **Cry3DEngine**.
All entities are loaded by **CryGame**, and physicalized from **CryEntitySystem**.
CryGame further uses this module for handling player movements.

Interface files:

- IPhysics.h

Additional documentation:

- Brief Physics Description.doc
- Physics System Programming Guide.doc
- Guidelines for Character Physics Authoring.pdf
- Guidelines for Vehicle Creation.pdf

CryScriptSystem

- Handling of LUA scripts
- LUA 4.01 code

The scripting system loads LUA script files, compiling and converting them to byte code on the fly. References to script objects are kept either in the Entity structure or in various sections of **CryGame**.

Interface files:

- IScriptSystem.h

Additional documentation:

- LuaManual.pdf
- FC editor manual, the scripting section

CrySoundSystem

- Loading and handling of 2d and 3d SFX effects
- Loading and handling dynamic music

- FMOD 3.61

The sound system handles sounds loaded from **CryGame** or from scripts.

Dynamic music patterns can be defined within the editor.

Sound areas, sound spots, EAX effects etc. Are all defined through scripts and are accessible via **CryEditor**, as well as several custom sound presets.

Interface files:

- ISound.h

Additional documentation:

- FC editor, the sound and music editing section.

CrySystem

Memory manager

Streaming engine

Lua debugger

XML loading code

Zlib

Custom cheat protection code

HTTP downloader

Frame profiler

Pak files handling

Console,Timer

- Glue with **CryScriptSystem**

Platform specific function calls

This module deals with lots of low-level and system specific functions. It is the first component loaded by the main executable **FarCry.exe**

Interface files:

- ISystem.h

FarCry.exe

Performs some basic initialization, then loads the CrySystem module.

FarCry.exe is the main executable which runs the game.

Editor.exe

Loading and saving of levels in editor format

- Handles real-time editing of the game

The editor acts as a replacement of **FarCry.exe**, by sending messages to **CryGame**, to switch in and out of game mode. Apart from loading / saving games and using the in-game menus, everything else that is possible to do within **CryGame**/**FarCry.exe** is possible to do within the editor as well.

This is a standalone executable.

Additional documentation:

- The FC_editor_manual_v1.1.pdf covers the editor in details.
- FCFAQ_SwoopAEon_v219.pdf

FarCry_WinSV.exe

- Handles windows dedicated server, for online playing.

The dedicated server shows only a simple console output, instantiating the NULL rendering subsystem.

Additional documentation:

- How to use the built in Internet simulator.pdf
- Tools to tweak FarCry MP.pdf

Resource Compiler

Compiles objects data into platform-specific optimized rendering format.

At the moment it compiles only animated models.

Additional Documentation:

ResourceCompiler.pdf

Xrender

- Handles shaders and platform specific rendering data and initialization.

The renderer has a common abstract rendering interface, and implements OpenGL, DirectX and NULL devices.

Interface files:

- Irenderer.h

Additional documentation:

- Shaders Specification.pdf

GAME STARTUP AND LOOP

Upon starting FarCry.exe, it will invoke CrySystem, which will check for startup parameters such as DEVMODE and MOD options (for more information about modding and DEVMODE, read the FCMG.pdf).

The game module gets loaded by CrySystem, after initializing all other subsystems and opening basic .pak files in the FCData folder.

At CryGame initialization time, the following operations are performed in order:

- Game script objects are initialized (look at CscriptObjectGame.pdf)
- Entity Class registry is initialized
- Execution of the main script, Main.lua, which in turn loads other basic scripts. The Init() function of Main.lua is called.
- The surface manager is initialized
- Input map and key bindings are initialized (if not in dedicated server mode)
- Main language and string tables are loaded
- HUD interface is created
- Game.cfg is loaded
- Material scripts are loaded
- If not in editor or dedicated server mode, then the UI system is created, using the file "Scripts/MenuScreens/UISystem.lua". This will load the UI menu configuration and language menu tables, as well as sounds and all other necessary menu data and pages. It will also load and playback the starting video sequence.

After the game is initialized, the script Devmode.Lua is loaded, if the game is in DEVMODE. After that, the flow control goes back to the game module, which keeps looping inside the CXGame::Update() function till the user decides to quit.

GAME UPDATE LOOP

During game's update, the game main loop will perform several tasks:

- First of all, it will check whether the game is in menu or game mode and it will enable or disable the 3d engine rendering accordingly.
- It will then pauses or restart the movie system, if the game is in pause mode or not.
- It checks current sound areas for music and sound playback.
- It updates the system and removes updating of dead player for physics.
- It checks for triggering of areas
- It updates the client, if there is one connected. In single player mode, there is always a client and a server, the client is connected as a loop back on the same local host machine.
- The server, if any present, is updated.
- The network module is then updated. It is used to update things like the UBI.com services.
- When in dedicated server mode, the dedicated server checks for remote command calls.
- The rendering of game visuals, menu or both now takes place.
- After rendering, the HUD is updated.
- Game messages coming from the CrySystem module or from scripts are now processed.

C++/LUA GAME SCRIPTING

For many reasons, often you would like to write a function in C++ and make it accessible from LUA scripts, or vice versa.

Game's C++ and Lua scripts are able to communicate and pass data between each other through the use of ScriptObjects. When you look at the CryGame module, you will see that many scriptobjects have been already created, for example ScriptObjectAI.cpp, ScriptObjectGame, ScriptObjectLanguage etc.

You can find a description of LUA functions of ScriptObjectGame in the file CscriptObjectGame.pdf.

Let's see how to create from scratch a new script object and expose C++ functions to LUA.

- We will create a new class, for example ScriptObjectTest, consisting of a .cpp and an .h header file. This class will implement scripting functions for exposing some functionalities we want to be accessible through script for easy gameplay tweaking and/or editing.

- In the class header file, include the following headers:

```
#include <IScriptSystem.h>
#include <_ScriptableEx.h>
```

In order to be able to derive a scriptable class.

Then derive the class:

```
class CScriptObjectTest :public _ScriptableEx<CScriptObjectTest>
{
  public:
  ....
```

- Add a Create() function, where you will initialize the scriptobject and pass additional parameters you may need (like a pointer to CryGame for instance), and a static InitializeTemplate() needed by the Scriptable object template.

```
void Create(IScriptSystem *pScriptSystem,CXGame *pGame);
static void InitializeTemplate(IScriptSystem *pSS);
```

- Then add the functions you need. The functions should always return an int and pass an IfunctionHandler as parameter.

```
int PrintTestMessage(IFunctionHandler *pH);
```

- Now let's move on to the .cpp file. Add the following line, after the include headers, to declare the scriptable template:

```
_DECLARE_SCRIPTABLEEX(CScriptObjectTest)
```

- Implement the Create() function:

```
void CScriptObjectTest::Create(IScriptSystem *pScriptSystem,CXGame *pGame)
{
  m_pGame=pGame;
  InitGlobal(pScriptSystem,"TestObject",this);
}
```

Now this will cause this script object to be globally accessible through scripts using the namespace TestObject.

- Let's implement now the InitializeTemplate function:

```
void CScriptObjectTest::InitializeTemplate(IScriptSystem *pSS)
{
  _ScriptableEx<CScriptObjectTest>::InitializeTemplate(pSS);

  REG_FUNC(CScriptObjectTest,PrintTestMessage);
```

Here you will simply register the functions you want to be accessible from LUA. You can add here global values as well, like:

```
pSS->SetGlobalValue("GVAR_TEST", 0);
```

- Finally let's implement the C++ function callable from script. Here we will simply output a message to the console and log file.

```
int CscriptObjectTest::PrintTestMessage(IFunctionHandler *pH)
```

The function handler contains all parameters coming from the LUA script. In this case we will pass a string with the message to print out.

There are macros provided to check for the number of parameters passed. Sometimes we need to pass a variable number of parameters, however in this case we want to be sure that only one parameter is passed:

```
CHECK_PARAMETERS(1);
```

If we wanted to know how many parameters were passed:

```
pH->GetParamCount()
```

If we wanted to verify that the first parameter passed was effectively a string:

```
pH->GetParamType(1) == svtString
```

Now let's get the string:

```
const char *szMessage = 0;
pH->GetParam(1, szMessage);

if (!szMessage)
  return pH->EndFunction();
```

And print the message to the console:

```
char szMsgLine[64] = {0};
sprintf(szMsgLine, "Message = \"%s\"", szMessage);
m_pGame->GetSystem()->GetIConsole()->PrintLine(szMsgLine);
```

And to the log:

```
m_pGame->m_pLog->Log("Message= \"%s\"", szMessage);
```

- The function call from any LUA script would be like:

```
TestObject:PrintTestMessage("This is a test message");
```

- We can return a value by passing it to EndFunction() and get it back in the LUA script. We can return multiple values, or a **table**, to the script; although this is used less often (creating tables and allocating memory will cause the GC to be called which in turn causes stalls). However to pass a table to the script we first need to create one. This is done through the SmartScriptObject , which is a smart pointer which will deallocate itself when there are no more references to it:

```
_SmartScriptObject pObj(m_pScriptSystem);
```

Then we can add values to the table, for example:

pObj->SetValue("TestNumber", nValueID);

We can also add tables to the table, for example:

pObj->SetAt (0, pTableObj);

At the end the table should be returned using:

return pH->EndFunction(*pObj);

- Similarly we can call a LUA function from C++, even though, again, this is usually done less often. Here is an example taken from the game code:

```
m_pScriptSystem->BeginCall("Game", "Connect");
m_pScriptSystem->PushFuncParam((int)(m_pGame->m_bLastCDAuthentication ? 1 : 0));
m_pScriptSystem->EndCall();
```

GAME'S LOADING AND SAVING

The current state of the game can be saved and loaded, by making a snapshot of all relevant data, and sending it to a stream, like you would do when sending a network packet containing the current game state – in fact the streaming code is shared between network and save games.

However a single player level usually contains much more data than a simple multiplayer map, therefore the save game stream usually grows much bigger.

When written to disk, the save game file is gz compressed.

The saving of a FC game is implemented in the function:

```
bool CXGame::SaveToStream(CStream &stm, string sFilename)
```

A stream object is already created and passed to the function from the game engine. All you have to do is to fill in the stream with the data you need to restore the game functionalities.

So basically you need to call one of the overloaded functions of stm::Write() with the data you are going to save. For example:

```
stm.Write((BYTE)CHUNK_ENTITY);
stm.Write(pEnt->GetScale());
stm.WriteBits((BYTE *)&tProps,sizeof(CS_REVERB_PROPERTIES));
```

In fact you will see that there is already code in place for saving and loading of game levels, position of entities, physics states, sound states, AI states etc.

I recommend not modifying the C++ code but instead to serialize the specific data you need for each entity from script.

This will ensure first of all that you don't introduce bugs in the basic code that will cause a chain reaction to all entities, plus it is more clean and simple to customize the saving data you need inside the proper script rather than trying to make it from C++ code which doesn't have specific knowledge of the entity being saved.

The C++ code will serialize anyway the entity properties declared in the Properties field of the entity, tweak able by the editor.

Game code is already in place which will call the LUA function OnSave when is time to save the game. For instance look at the script RaisingWater.lua:

```
--------------------------------------------------------------------------------
function RaisingWater:OnSave(stm)
	stm:WriteFloat(self.currlevel);
	stm:WriteBool(self.waterstopped);
end
```

If such a function is declared in the script, it will get automatically called at saving time. The exact same philosophy applies when loading a game. The loading function is called OnLoad(), here is an example from BasicEntity.Lua:

```
---------------------------------------------------------------------------------------------------

function BasicEntity:OnLoad(stm)
	self.animstarted=stm:ReadInt();
	if (self.animstarted==1) then
		self:StartEntityAnimation();
	end
end
```

It is crucial to know that the data is serialized in order as the function calls to OnSave() are made, therefore it must be read back exactly in the same sequence as it has being saved, because the data stream is not hierarchical but linear. However there is no reason that the above shouldn't work, unless there are errors in the script, which could also hide more serious problems. Debugging message facilities are provided, in verbosity level higher than 5, to detect which entity caused the stream corruption, if any.

17 February 2005

Author: Marco Corbetta