# Table of Contents

To be written… right now just a layout of document

Overview

* Visual studio solution overview
  + Where to download
  + Overlook at the projects
  + Notes on compiling the source
* Architecture
  + Basic diagram
  + Notes on code style
* Unrolling the main loop

Event System

Scripting

AI

# Overview

The source code is distributed through Github and available to clone with git via this url <https://github.com/TTimo/doom3.gpl.git>. This project compiles pretty easily with Visual Studio 2010 Professional, but there are some initial compile issues if you are using Visual Studio 2010 Express because it lacks MFC. There are workarounds for removing the MFC dependencies located here <https://bugzilla.icculus.org/show_bug.cgi?id=5290>. **NOTE:** Removing the MFC dependencies do disable the tools from running from the visual studio version. One other thing that had to be done in order to compile is change line 167 of snd\_system.cpp located in the Doom3 project from this:

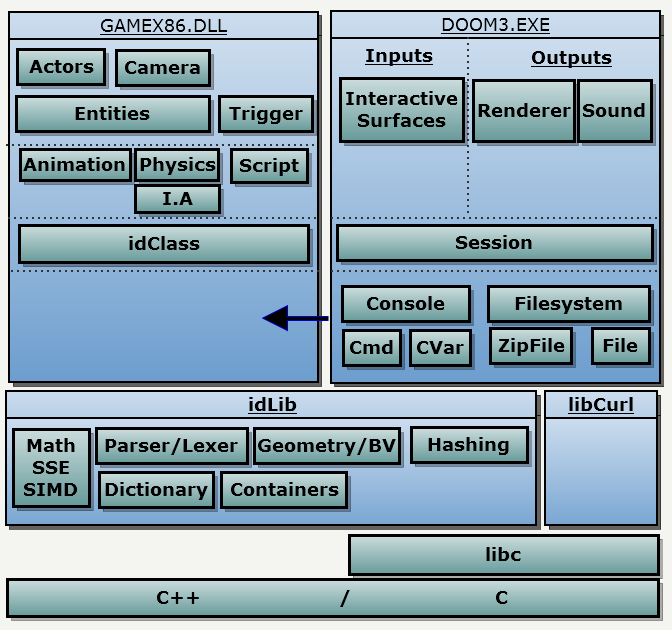
common->Printf( "%8d kB total OpenAL audio memory used\n", ( alGetInteger( alGetEnumValue( "AL\_EAX\_RAM\_SIZE" ) ) - alGetInteger( alGetEnumValue( "AL\_EAX\_RAM\_FREE" ) ) ) >> 10 );

To this:

common->Printf( "%8d kB total OpenAL audio memory used\n", ( alGetInteger( alGetEnumValue( (ALubyte \*) "AL\_EAX\_RAM\_SIZE" ) ) - alGetInteger( alGetEnumValue( (ALubyte \*) "AL\_EAX\_RAM\_FREE" ) ) ) >> 10 );

The solution is located in the neo folder of the repository, and is split into separate projects that highlight the overall architecture of the engine.

|  |  |  |
| --- | --- | --- |
| **Projects** | **Output** | **Observations** |
| Game | gamex86.dll | This project contains all the basic gameplay for Doom 3. |
| Game-d3xp | gamex86.dll | This project contains the gameplay added for the expansion. |
| MayaImport | MayaImport.dll | Part of the assets creation pipeline. Loaded at runtime to open maya files and import monsters, camera paths, and maps. |
| DoomDLL | Doom3.exe | Doom 3 engine code |
| TypeInfo | TypeInfo.exe | In-house RTTI helper: outputs a GameTypeInfo.h. This is a map of all Doom 3 class types with each member size. This allows debugging via TypeInfo class. |
| CurlLib | CurlLib.lib | HTTP Client that is used to download files (statically linked against gamex86.dll and doom3.exe). |
| idLib | idLib.lib | Id software library. Contains generic and often reused code including parser, strings, dictionary, and SIMD math stuff.(statically linked against gamex86.dll and doom3.exe) |

The bulk of the game is split between two projects: the engine side of the code is contained within the DoomDLL project, and the game code is contained within Game/Game-d3xp.

Some essential systems that are needed within the Game project like the Filesystem, and Console are contained within the Engine project. These systems are passed to the Game dll during initialization of the game.

When the Doom3.exe starts up:

* It loads the game Dll into memory via LoadLibrary.
* It gets the address of the GetGameAPI function via GetProcAddress (this is the only exported function from the game Dll)
* Calls GetGameAPI

Once GetGameAPI is called the engine and the game exchange object pointers, so that at the end of the exchange, Doom3.exe has a pointer to the idGame object and gamex86.dll has a pointer to gameImport\_t struct that contains references to all the subsystems within the engine that the game might need like Filesystem, etc…. The overall code architecture of the engine is class-hierarchy based. The Doom3 engine and game code is written in C++, and makes good use of abstraction and polymorphism throughout most of the classes. For instance, all classes within the game module are derived from idClass. idClass handles the basic functionality of classes within their game like generating the type information (which allows for RTTI and run time class instantiation through just the classname), saving and loading of basic class information, and processing events.

Some other links to check out before diving into the source code:

* [idTech4 Coding Standard](http://fd.fabiensanglard.net/doom3/CodeStyleConventions.pdf)
* [fully unrolled main game loop](http://fabiensanglard.net/doom3/doom3_unrolled.php)

# Event System

The event system is controlled and managed through the usage of scripts, and two classes’ idEvent and idEventDef. Events can take up to 8 arguments and there are 8 valid argument types. There can be a maximum of 4096 events which are allocated statically in Event.cpp. All of these parameters are controlled through #defines.

#define D\_EVENT\_MAXARGS 8

#define D\_EVENT\_VOID ( ( char )0 )

#define D\_EVENT\_INTEGER 'd'

#define D\_EVENT\_FLOAT 'f'

#define D\_EVENT\_VECTOR 'v'

#define D\_EVENT\_STRING 's'

#define D\_EVENT\_ENTITY 'e'

#define D\_EVENT\_ENTITY\_NULL 'E'

#define D\_EVENT\_TRACE 't'

#define MAX\_EVENTS 4096

Events are declared by creating an eventDef which lays out the format of the event through strings passed in to its constructor.

const idEventDef AI\_FindEnemy( "findEnemy", "d", 'e' );

const idEventDef AI\_FireMissileAtTarget( "fireMissileAtTarget", "ss", 'e' );

The first parameter of the constructor defines the name used to invoke this event. This is the call you will see in scripts that use this specific event. The second parameter defines the format of the arguments for this event. This string gets parsed within the constructor and calculates the correct argument size for the event which the idEvent class uses when allocating the data within the event class. The third parameter specifies the return parameter that is returned to the script that invoked the event.

To bind the events to specific callback functions a couple of macros are used that generates the classes RTTI information that doomScript uses to link script function calls to their C++ callback functions.

CLASS\_DECLARATION( idActor, idAI )

EVENT( EV\_Activate, idAI::Event\_Activate )

EVENT( EV\_Touch idAI::Event\_Touch )

EVENT( AI\_FindEnemy, idAI::Event\_FindEnemy )

EVENT( AI\_FindEnemyAI, idAI::Event\_FindEnemyAI )

EVENT( AI\_FireMissileAtTarget, idAI::Event\_FireMissileAtTarget )

END\_CLASS

Check AI\_events.cpp for more examples of how events are defined.

# Scripting

For a look into how the scripting VM is architected take a look at this webpage

[Scripting VM Architecture](http://fabiensanglard.net/doom3/scripting_vm.php) .

#define SCRIPT\_DEFAULTDEFS "script/doom\_defs.script"

#define SCRIPT\_DEFAULT "script/doom\_main.script"

#define SCRIPT\_DEFAULTFUNC "doom\_main"

When the game starts up idProgram is fed the default script to compile. This default script contains a number #include directives that tells the compiler to compile those scripts as well. All the compiled scripts are contained within idProgram. Scripts are called from C++ through the usage of idThread. idThread contains a idInterpreter which keeps track of the instruction pointer, call stack, and data stack for that particular script. Each idThread is given real CPU time by the engine each frame allowing the script to execute until it hits a multi-frame event or completes it work for the fame.

Scripts are written in an object-oriented language similar to C++ and are used in many areas of Doom 3. It’s particularly used for entity/map definitions, GUI creation (done in a separate scripting system), Weapon/AI (Monster) behaviors, and scheduling events to occur in the game. For more information on the syntax of the language check out this link: [Script File Syntax](http://www.modwiki.net/wiki/SCRIPT_%28file_format%29). Now let’s take a look at how entity definitions and objects are created in scripts.

Entity definitions are simply a collection of key/value pairs with a name. They are normally used to define entities but can really be used to define anything that just needs a collection of key/value pairs to be defined. The meaning of these key/value pairs is completely dependent on the type of object it is, but a couple of key/value pairs remain constant regardless of type.

The “spawnclass” key defines the C++ class that spawns this entity. Items use idItem, and monsters use idAI, etc…. Entities cannot be spawned unless the “spawnclass” key is defined.

The “inherit” key tells the game to copy all the key/value pairs from another entityDef. Circular references are not a concern because entityDef’s are only ever parsed once and you will get an error if an entityDef is parsed more than once. The inherit keyword can be very useful for defining base entity types like a default monster or bot.

entityDef monster\_zombie\_base {

//parent EntityDef

"inherit" "zombie\_default"

//Model used for this entity

"model" "model\_monster\_zombie"

//This key/value pair tells the game to construct a script object

//when spawning this entity.

"scriptobject" "monster\_zombie"

//This information is used by idEntity, and idAI.

"size" "40 40 72"

"use\_aas" "aas48"

"team" "1"

"rank" "0"

"health" "80"

"melee\_range" "32"

"turn\_rate" "360"

"mass" "150"

"burnaway" "1.5"

"bone\_focus" "headcontrol"

"bone\_leftEye" "Leyeaim"

"bone\_rightEye" "Reyeaim"

"look\_min" "-90 -125 0"

"look\_max" "25 125 0"

"look\_joint Waist" "0.1333 0.1333 0"

"look\_joint Chest" "0.1333 0.1333 0"

"look\_joint Shoulders" "0.1333 0.1333 0"

"look\_joint headcontrol" "0.6 0.6 0"

"look\_joint Ruparm" "-0.4 0 0"

"look\_joint Luparm" "-0.4 0 0"

"ragdoll" "monster\_zombie\_base"

//More information in def file below omitted for space

}

Script Objects are used to organize variables and functions related to an entity. Script Objects are analogous to classes in object oriented terms. They are declared similar to classes in C++ and even support single inheritance. The entities “spawnclass” strongly defines its runtime behavior, and Script Objects are generally used to create variation on runtime behavior within that framework.

// script Objects support single inheritance from another object like //classes in C++.

object monster\_zombie : monster\_zombie\_base {

boolean can\_run;

void state\_Begin();

void state\_Idle();

void combat\_melee();

// the init function is like the constructor for a object

// it is called once when the script object is constructed

void init();

// the destroy function acts as the destructor for a object

// it is called once when the script object is destroyed

void destroy();

// torso anim states

void Torso\_Idle();

void Torso\_Pain();

void Torso\_MeleeAttack();

// legs anim states

void Legs\_Idle();

void Legs\_Walk();

void Legs\_Run();

// attacks

float check\_attacks();

void do\_attack( float attack\_flags );

}

//functions are defined in a similar fashion to c/c++

void monster\_zombie::init() {

//inheritance from another object gives direct access to variables

//declared in the base class.

run\_distance = ZOMBIE\_RUNDISTANCE;

walk\_turn = ZOMBIE\_WALKTURN;

//some functions are linked from c++ game code from the

//declaration of eventDefs and linked through the EVENT macro

//typically these functions are defined in the spawnclass

//associated with the object

can\_run = hasAnim( ANIMCHANNEL\_LEGS, "run" );

setState( "state\_Begin" );

}

# AI

The AI folder in Doom 3 does not actually contain the AI for monsters those are all defined in scripts. The C++ code acts as the glue that ties these scripts to the rest of the game. The AI logic is run on the same thread as the rest of the game logic, and active entities are updated each frame through a call to their Think(). In addition to the glue for monsters the AI folder also contains files related to the Area Awareness System (AAS) which is the pathfinding system that allows NPCs to navigate the levels in Doom 3.

The Area Awareness System (AAS) is used by both monsters and allied npc characters to pathfind through levels in Doom 3. Map files are compiled with the “runAAS” command which generates an .aas file which contains all the information needed for a AI to navigate the level. Depending on the monster size a different .aas file is generated, generally humanoid and smaller monsters use aas48x48 and larger monsters use aas96x96 (size is in reference to game units).

When it comes to navigating the map using the AAS, routing refers to travelling from one area to another and pathing refers to travelling around obstacles and to locations within an area. Monsters that walk generally use the function WalkPathToGoal to reach a goal position. This function returns a boolean if the goal is reachable, and also fills out a aasPath\_t struct that directs the ai towards the next point along that path. If the goal position lies in a different area this function will route to the goal area and return the idReachability that connects the two areas.

typedef struct aasPath\_s {

// path type

int type;

// point the AI should move towards

idVec3 moveGoal;

// number of the area the AI should move towards

int moveAreaNum;

// secondary move goal for complex navigation

idVec3 secondaryGoal;

// reachability used for navigation

const idReachability \* reachability;

} aasPath\_t;

For normal monsters this struct is used in conjunction with another class called idMoveState which contains additional needed information to tell monsters how to move toward a goal over multiple frames.

class idMoveState {

public:

idMoveState();

void Save( idSaveGame \*savefile ) const;

void Restore( idRestoreGame \*savefile );

moveType\_t moveType;

moveCommand\_t moveCommand;

moveStatus\_t moveStatus;

idVec3 moveDest;

idVec3 moveDir;

idEntityPtr<idEntity> goalEntity;

idVec3 goalEntityOrigin;

int toAreaNum;

int startTime;

int duration;

float speed;

float range;

float wanderYaw;

int nextWanderTime;

int blockTime;

idEntityPtr<idEntity> obstacle;

idVec3 lastMoveOrigin;

int lastMoveTime;

int anim;

};

# Bot Implementation Related

To have effective networked bots, their actions have to mimic that of a player and their knowledge should be similarly limited. For them to be networked their actions have to translated the same way across the network as a player’s would. Normal players have their input translated to actions across the network by filling out a usercmd\_t class which is generated from directly polling the HID each frame.

class usercmd\_t {

public:

int gameFrame; // frame number

int gameTime; // game time

int duplicateCount;// duplication count for networking

byte buttons; // buttons

signed char forwardmove; // forward/backward movement

signed char rightmove; // left/right movement

signed char upmove; // up/down movement

short angles[3]; // view angles

short mx; // mouse delta x

short my; // mouse delta y

signed char impulse; // impulse command

byte flags; // additional flags

int sequence; // just for debugging

public:

void ByteSwap();

bool operator==( const usercmd\_t &rhs ) const;

};

// usercmd\_t->button bits

const int BUTTON\_ATTACK = BIT(0);

const int BUTTON\_RUN = BIT(1);

const int BUTTON\_ZOOM = BIT(2);

const int BUTTON\_SCORES = BIT(3);

const int BUTTON\_MLOOK = BIT(4);

const int BUTTON\_5 = BIT(5);

const int BUTTON\_6 = BIT(6);

const int BUTTON\_7 = BIT(7);

// usercmd\_t->impulse commands

const int IMPULSE\_0 = 0; // weap 0

const int IMPULSE\_1 = 1; // weap 1

const int IMPULSE\_2 = 2; // weap 2

const int IMPULSE\_3 = 3; // weap 3

const int IMPULSE\_4 = 4; // weap 4

const int IMPULSE\_5 = 5; // weap 5

const int IMPULSE\_6 = 6; // weap 6

const int IMPULSE\_7 = 7; // weap 7

const int IMPULSE\_8 = 8; // weap 8

const int IMPULSE\_9 = 9; // weap 9

const int IMPULSE\_10 = 10; // weap 10

const int IMPULSE\_11 = 11; // weap 11

const int IMPULSE\_12 = 12; // weap 12

const int IMPULSE\_13 = 13; // weap reload

const int IMPULSE\_14 = 14; // weap next

const int IMPULSE\_15 = 15; // weap prev

const int IMPULSE\_16 = 16; // <unused>

const int IMPULSE\_17 = 17; //ready to play

Currently my idea for the implementation of multiplayer bots splits into a few different classes each with their own responsibilities. Eventually I will have a diagram for this, but for now I will give a brief description of the class and how it fits into the overall implementation

**afiFakeClient** - Normal clients connected to the server grab the usercmd\_t associated with the client directly from the HID, so a fake client will have to implemented to pull the usercmd\_t from code rather than directly from the HID. Each update the fake client will call GetBotInput() which will fill out the usercmd\_t for the fake client with the usercmd\_t stored within **afiBotManager** for that client.

**afiBotManager** – The bot manager will contain the basic functionality of adding and removing bots from the game, as well as, holding onto the usercmd\_t classes for all the bots.

**afiBotPlayer** – this class will represent the basic fake client ‘body’ of the bot derived from idPlayer. This class will contain the ‘brain’ of the bot which will actually contain the decision making code for the bot. This class will simply receive an input structure each frame from the brain that contains the basic information needed by the bot to act this frame. It takes the data from the input structure, and converts it to valid data for the usercmd\_t class. The fake client will end each Think() frame by pushing its usercmd\_t to the **afiBotManager**.

**afiBotBrain** – this class will represent the base ‘brain’ of the bot. This is the class students would derive from for their specific bot implementation in another DLL. Their derived class would be loaded via a CreateBrain() function that is exported from the DLL, and stored within the **afiBotManager**.