**DOOM ENGINE DECONSTRUCTION**

**- Analysis of ZDoom engine**

(Source port of original DOOM to windows)

**Bhumitra Nagar**

**Yufan Lu**

**INTRODUCTION**

***Doom*** is a first person shooter (FPS) developed by id Software in 1993. It is a science fiction horror themed game which is considered to be the “daddy” of FPS genre. Its organic design pumped with high octane action proved to be a huge success and paved the way for the FPS genre as we know today.

This document analyses various portions of the ZDoom version of the Doom engine. ZDoom is a source port of the official DOOM source code to windows.

**I. Time and Game Loop**

D\_DoomMain() is the first thing that is called as soon as the user starts the game. This function initializes the Doom game and sets up the environment. It loads the zdoom.pk3 which contains internal game info (this is essential for the ZDoom engine). Then the Doom IWAD (original DOOM wad file) is loaded which is essential to run any custom wads. The engine then checks for available custom wad files (PWAD). If there are no PWAD files to load, the engine loads the first level from the Doom IWAD. When a PWAD or IWAD is set, the engine initializes and loads the sounds, definitions, mapinfo, textures, custom settings, graphics, menu and configures all other parameters and finally calls the D\_DoomLoop() when everything is successfully loaded. It never returns to this point once it enters the game loop.

The code snippet below is for debugging purposes

if (singletics)

{

. . . .

}

This is for the debug mode. If the single ticks is set to true the game runs in debug mode with more number of ticks per second. Otherwise the TryRunTics() is run which sets the game ticks with the real ticks and is responsible for setting the frame rate and synchronizing the ticks when multiple players are there. The I\_StartTic and D\_Display update the display i.e. the next frame with the current state.

else

{

TryRunTics (); // will run at least one tick

}

// Update display, next frame, with current state.

I\_StartTic ();

D\_Display ();

**II. Human Interface Devices**

Doom supports keyboard, mouse and Joysticks, so does the ZDoom. In the g\_game.cpp, function G\_BuildTiccmd takes multiple inputs and puts them into the buttons object. It handles the inputs from keyboard, mouse and Joysticks and takes appropriate actions.

D\_event.h defines the structures for handling input. Below is the code snippet

//

// Event handling.

//

// Input event types.

enum EGenericEvent

{

EV\_None,

EV\_KeyDown,// data1: scan code, data2: Qwerty ASCII code

EV\_KeyUp, // same

EV\_Mouse, // x, y: mouse movement deltas

EV\_GUI\_Event, // subtype specifies actual event

EV\_DeviceChange, // a device has been connected or removed

};

// Event structure.

struct event\_t

{

BYTE type;

BYTE subtype;

SWORD data1; //keys/ mouse/joystick buttons

SWORD data2;

SWORD data3;

int x; // mouse/joystick x move

int y; // mouse/joystick y move };

**III. Resource management\*\***

The DOOM engine manages its whole memory using an internal allocator called Zone Memory System, instead of the malloc/free function in C++. The engine utilizes the M\_Malloc and M\_Free to alloc/dealloc the memory.

|  |
| --- |
| void \*M\_Malloc(size\_t size)  {  void \*block = malloc(size+sizeof(size\_t));  if (block == NULL)  I\_FatalError("Could not malloc %zu bytes", size);  size\_t \*sizeStore = (size\_t \*) block;  \*sizeStore = size;  block = sizeStore+1;  GC::AllocBytes += \_msize(block);  return block;  } |

|  |
| --- |
| void M\_Free(void \*block)  {  if (block != NULL)  {  GC::AllocBytes -= \_msize(block);  free(block);  }  } |

It’s not calling the low-level memory management routine, instead, it allocates a single, large, continuous block of memory when the game starts. The way it manages the memory is to cut the memory into blocks, and form a linked-list. When two or more free blocks touch each other, they would be merged into a large one, to keep the linked-list size short.

This management keeps the time to search a free block shorter, and can prevents unnecessary memory fragments efficiently.

The Garbage Collection System (GC) in DOOM engine is interesting too. It has some kinds of tags:

1. ***PU\_STATIC***

This is a common tag for most of the memory. With this tag, the memory must be explicitly frees with M\_Free function.

1. ***PU\_CACHE***

The memory with this tag could be freed back automatically to the system when the memory runs out. This tag is mainly used for caching the WAD data. When loading the WAD data into the engine, the memory is tagged with PU\_STATIC, after loading finishes, the tag would be changed to PU\_CACHE, storing them in the engine, when the engine is low in memory, the data would be freed, otherwise they are kept in the system for future use fast.

1. ***PU\_LEVEL***

The PU\_LEVEL tag is used for marking the current level’s memory. When the current level is finished, all the memory marked with PU\_LEVEL would be freed.

The memory management system is like the mix of stacked-based allocator and the linked-list allocator (see the other paper).

**IV. 3D Rendering/Drawing**

**V. Character Animation/Sprites**

Sprites (the actors as we call it in Doom) are essentially everything in DOOM i.e. the player characters, enemies, monsters, decorations, objects, weapons, ammo, etc. Each move of any object in doom is has a sprite associated with it.

The actor.h defines what all an actor can do in the game. The source code contains the description in detail, so there is no need of elaborating it here.

The p\_mobj.cpp handles the object movements and the spawn functions.

The

void AActor::Serialize (FArchive &arc)

{

. . . .

Either reads or writes a sprite.

An archive object reads the values set for an actor and choses the appropriate sprite. For instance the following parameters are needed for an enemy:

* health
* mass
* speed
* Radius
* Height
* painchance
* seesound
* painsound
* deathsound
* activesound
* States: which can be spawn, see, melee, pain and death.

**VI. Physics, Collision Detection & Rigid Body Dynamics.**

The p\_map.cpp handles the Movement, collision handling Shooting and aiming.

Doom works with sprites. There are no rigid body dynamics associated with it. The radius and height of the actor are used by the collision detection code which also checks whether a monster has been hit by a weapon.

<explanation from other paper>

Doom was the first commercial game that used BSP trees.

The BSP tree is extremely efficient in collision detection. With this tree, the collision detection is reduced to tree traversal, or search by rejecting a lot of geometry early. Finally, there are only small amount of planes to test.

(Refer to:

http://en.wikipedia.org/wiki/Binary\_space\_partitioning)

**VII. Game Object Models**

http://doom.wikia.com/wiki/Doom\_rendering\_engine#Node\_building

**VIII. Events and message passing**

key and mouse events.. g\_game.cpp d\_protocol.h

cmd->ucmd

**IX. Game Audio**

s\_\*\*\*\*.cpp/h for sound

**X. Devices/Hardware**

mouse and keyboards