DOOM ENGINE DECONSTRUCTION

- Analysis of ZDoom engine
(Source port of original DOOM to windows)

Bhumitra Nagar
Yufan Lu

nagar.b@husky.neu.edu
lu.yuf@husky.neu.edu

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 remove
};
```

```
// 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.

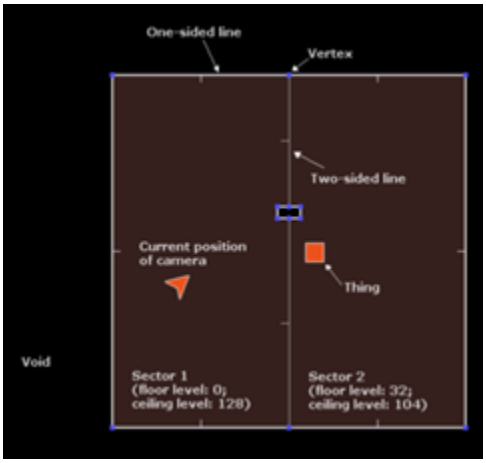
The Garbage Collection System (GC) in DOOM engine is interesting too. It has some kinds of tags:

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.
2. 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.
3. 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.

IV. 3D RENDERING/DRAWING

The Doom rendering engine is not a true-3D engine. It is not possible to look up and down. Also, no two sectors can be placed one above another.



The above figure shows how the levels are represented in the doom engine. It is a top down view of a level and hence we cannot have rooms above rooms in Doom. This map contains basic elements for its construction that is vertex, linedefs and sectors.

BSP: The Zdoom engine makes use of BSP (binary space partitioning) and a node builder to build BSP data for a level. This is handled in the file p_glnodes.cpp. BSP divides this data into a binary tree where each node in the tree represents a particular area of the map. The root node represents the entire area. At each branch a line divides an area of a node into two subnodes and linedefs into line segments called segs. The leaves of the tree are

convex polygons called as subsectors and are bound to a particular sector which has segs associated with it. The BSP system sorts these subsectors for rendering using the following algorithm[7]:

1. Start at the root node.
2. Draw the child nodes of this node recursively. The child node closest to the camera is drawn first. (This can be found by looking at which side of a given node's dividing line the camera is on.)
3. When a subsector is reached, draw it.

This results in drawing huge maps as no time is wasted in drawing things that are far away from the camera. This algorithm is implemented in the r_bsp.cpp file which includes all the headers r_draw.h, r_things.h and r_main.h for rendering all actors and objects in the game. Sky rendering is done in the r_sky.cpp file. R_main.cpp files initializes the renderer for texture mapping, fixed color maps, and free look.

When rendered, the map in the previous figure looks like this[7]:

Rendering: The doom engine renders the walls as it traverses the BSP tree by drawing the sectors closest to the camera first. This is stored in a linked list so that later on it can be used again instead of redrawing it. There is a limit (256) to which these sectors can be drawn. If excess segs are there, they are not drawn, but are generally not visible as they are far away. As the end of a wall is reached, the engine builds up a map of up to this part and the distant parts invisible to the player are chopped off. Then the wall textures stored in the WAD file are drawn.

The floors and ceilings are drawn using "flood fill" algorithm. These textures for floors and ceilings are called as flats and are drawn horizontally at the given ceiling and floor height.

The sprites which are there in a sector are placed in a list of sprites to be drawn. The engine stores this list and sorts it before rendering anything. Again, max number of sprites that can be drawn is 128. The edges of sprites are clipped by checking the list of segs previously drawn. Also the sprites here are stored in the same column-based format as wall textures, which again is useful for the rendering engine as the same functions can be used to draw both walls and sprites.

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.

```
void AActor::Serialize (FArchive &arc)
{
   ...
```

Above code snippet 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 death, etc.

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.

The default settings of the Zdoom engine treat the mouse events as key events and doom has auto aiming and shooting the enemies. So irrespective of the elevation, the enemy takes damage. Function D_PostEvent in D_Main.cpp handles this. It can be modified to have a functionality of freelook.

The P_AimLineAttack function in p_map.cpp takes care of the aiming and shooting part in doom.

```
//  p_map.cpp,  line  3369,  P_AimLineAttack
function
if (t1->player != NULL)
{
    aim.shootz  +=  FixedMul  (t1->player->mo-
>AttackZOffset, t1->player->crouchfactor);
}
else
{
    aim.shootz += 8*FRACUNIT;
}
//  p_map.cpp,  line  3405,  P_AimLineAttack
function
aim.toppitch = t1->pitch - vrange;
aim.bottompitch = t1->pitch + vrange;
```

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.

P_Map.cpp handles the collision as with other actors in the game.

P_interactions.cpp handles the collision with game objects like poison, lightening. Also displays the obituaries and has functions to interact with the items in the game. The P_damage function gives the amount of damage inflicted upon the actor by another actor, or -1 if the damage was cancelled.

VII. GAME OBJECT MODELS

Doom is not a true-3D game. It relies on pseudo 3D techniques to render all objects in the game. Hence we have sprites for everything i.e. actors, weapons, objects, obstacles, damage, things, decoration etc.

All these properties for the sprites are defined in the actor.h header file. A part of the code from this header is given below. These parameters define an actor and are used to write our own DECORATE files which are used for creating mods.

```
fixed_t Speed;
fixed_t FloatSpeed;
fixed_t MaxDropOffHeight, MaxStepHeight;
SDWORD Mass;
SWORD PainChance;
int PainThreshold;
FNameNoInit DamageType;
FNameNoInit DamageTypeReceived;
fixed_t DamageFactor;
fixed_t   radius,   height;   //   for   movement
checking


// info for drawing
// NOTE: The first member variable *must* be
x.
fixed_t x,y,z;
AActor  *snext, **sprev;    // links in sector
angle_t angle;
WORD   sprite;       // used to find patch_t
and flip value
BYTE   frame; // sprite frame to draw
fixed_t       scaleX, scaleY;             //
Scaling values; FRACUNIT is normal size
FRenderStyle  RenderStyle;         // Style to
draw this actor with
DWORD  renderflags;  // Different rendering
flags
FTextureID    picnum; // Draw this instead of
sprite if valid
DWORD  effects;// [RH] see p_effect.h
fixed_t alpha;
DWORD  fillcolor;   //   Color   to   draw   when
STYLE_Shaded
```

VIII. EVENTS AND MESSAGE PASSING

There are two kinds of events which are handled by the doom engine - Keyboard and Mouse Events which are handled in the g_game.cpp file.

All the events are handled in G_BuildTiccmd method. The mouse events are also treated as key events and perform the same action as the keyevents do e.g. movement of player forward.

```
if (Button_Forward.bDown) {
    forward += forwardmove[speed];
    if (Button_Back.bDown)
       forward -= forwardmove[speed];
}
```

Similarly all events are handled here. I_keyboard.cpp handles the keyboard events. Method `FKeyboard :: PostKeyEvent` Posts a keyboard event, but only if the state is different from what we currently think it is. For example the keyboard input sends a key down event every time, so these are discarded.

The header d_protocol.h defines the key events

```
struct usercmd_t
{
    DWORD     buttons;
    short     pitch;
    // up/down
    short     yaw;
    // left/right
    short     roll;
    // "tilt"
    short     forwardmove;
    short     sidemove;
    short     upmove;
};

enum
{
    UCMDF_BUTTONS          = 0x01,
    UCMDF_PITCH            = 0x02,
    UCMDF_YAW              = 0x04,
    UCMDF_FORWARDMOVE      = 0x08,
    UCMDF_SIDEMOVE         = 0x10,
    UCMDF_UPMOVE           = 0x20,
    UCMDF_ROLL                    = 0x40,
};
```

ZDoom engine also supports multiplayer which was missing from the original doom engine. The multiplayer code is embedded into multiple files so that doom supports multiplayer.

ZDoom uses UDP/IP for all network play. The game state is tracked on a peer-to-peer system ZDoom's advantage, however, is its low bandwidth usage (averaging 300 bytes per second per node) and lack of the "host-advantage" phenomena (the networking is lock-step). Map "scale" (enemy count, complexity, etc) also has no effect on bandwidth, due to netgames in ZDoom only needing to send player control data.[2]

Zdoom supports two networking modes which is set in the file d_net.cpp. Based on the values of game ticks and network ticks, this file handles the entire networking part of the engine.

```
BYTE NetMode = NET_PeerToPeer;
```

Peer to peer: Netmode value of 0 sets the engine to peer to peer mode which is default for two player games. Here each node communicates with the other node directly and game speed/lag is dependent on the slowest connection among them.

Packet Server: Netmode value of 1 sets the engine to packet server mode which is default for 3+ players. Here each node talks to the arbitrator (first player) and then routes the info to other nodes. This process might slow down the game and may result in a lag but is highly recommended for large number of players.

IX. GAME AUDIO

The files responsible for the sound in doom are all files starting with “s_”.

The sounds associated with sprites are sight, pain, death, active etc. which correspond to the different states we have for the each character.

The Zdoom engine supports the following music and sound formats : MOD, XM, IT, S3M, MIDI, OGG Vorbis, SPC, FLAC, MP3 and MUS.[1]

Description of the files:

s_environment.cpp – This file specifies the reverb properties and the sounds to play on interaction with the environment.

s_advsound.cpp – This file has functions that are responsible for mapping, indexing, registering and retrieving the sound files. The S_GetMusicVolume gets the relative volume for specified track. S_hashsounds fills maintains the working hash table for SFX. S_PickReplacement chooses a sound from the random sound list. S_GetSoundMSLength returns the duration of a sound. Also we have functions for deducing the class and gender of the actor, and to play the ambient sounds.

s_playlist.cpp – This file handles the m3u playlist parsing. It is an implementation of the F_Playlist class.

s_sound.cpp – This file is responsible for the start and stop of music when a level begins. The methods s_start() loads a new music by killing of all the

previous playing sounds and starts a new one. S_init() initializes all parameters related to sound e.g. the volume for SFX and music, sets channels, allocates channel buffers and also sets the s_sfx lookup.

X. DEVICES/HARDWARE

The hardware.cpp file initializes the graphics using the method I_InitGraphics (). It sets the focus on the doom game window and sets the windows size according to the settings – full screen mode or windowed mode `(DFrameBuffer *I_SetMode).`

The mouse and keyboard are two the devices that zdoom engine handles. It also handles the joystick events like the mouse events.

I_keyboard.cpp and i_mouse.cpp are responsible for handling the keyboard and mouse inputs. Refer to section II and VIII of this document for more details on these.

I_main.cpp has the system specific startup code. The method DoMain() sets the console parameters by determining the operating system using the I_DetectOS method of i_system.cpp and according to the system it eventually calls the D_DoomMain() method which runs the game.

XI. REFERENCES

[1] Zdoom Wiki. 2014. “ZDoom” http://doom.wikia.com/wiki/ZDoom

[2] Zdoom Wiki. 2014. “ZDoom” http://www.zdoom.org/wiki/Multiplayer

[3] Nagar, Bhumitra. 2014. “Graphics in Video Games”.

[4] Nagar, Bhumitra and Lu, Yufan. 2014. “Doom Engine Modification”

[5] Lu, Yufan. 2014. ”Modern Game Memory Allocation” http://doom.wikia.com/wiki/Zone_memory

[6] Wikipedia. 2014. “Binary Space Partitioning”. November 2014. http://en.wikipedia.org/wiki/Binary_space_partitioning

[7] Doom Wiki. 2014. “Doom Rendering Engine” http://doom.wikia.com/wiki/Doom_rendering_engine