

0.1 About the Source Code

Commander Keen episodes 1-5 source code is not available as the current owner Zenimax¹ has, as of writing this book, no interest in selling intellectual properties. Luckily the ownership of Commander Keen: Keen Dreams was in the hands of Softdisk. In June 2013, developer Super Fighter Team licensed the game from Flat Rock Software, the then-owners of Softdisk, and released a version for Android devices.

The following September, an Indiegogo crowdfunding campaign was started to attempt to buy the rights from Flat Rock for US\$1500 in order to release the source code to the game and start publishing it on multiple platforms. The campaign did not reach the goal, but it's creator Javier Chavez made up the difference, and the source code was released under GNU GPL-2.0-or-later soon after.

0.2 Getting the Source Code

The source code is made available via `github.com`. It is important to take the the source code from shareware version 1.13, otherwise you run into issues due to incompatible map headers². To get the correct source code

```
$ git clone https://github.com/keendreams/keen.git
$ cd keen
$ git checkout a7591c4af15c479d8d1c0be5ce1d49940554157c
```

0.3 First Contact

Once downloaded via `github` a folder 'keen' is created with all source files inside. `cloc.pl` is a tool which looks at every file in a folder and gathers statistics about source code. It helps for getting an idea of what to expect.

¹June 24, 2009, it was announced that id Software had been acquired by ZeniMax Media (owner of Bethesda Softworks).

²See issue #7 on <https://github.com/keendreams/keen>.

```
$ cloc keen
```

```
52 text files.
52 unique files.
7 files ignored.
```

| Language | files | blank | comment | code |
|--------------|-------|-------|---------|-------|
| C | 20 | 4008 | 5361 | 14893 |
| Assembly | 5 | 992 | 1114 | 2688 |
| C/C++ Header | 19 | 508 | 665 | 1603 |
| Markdown | 1 | 18 | 0 | 40 |
| DOS Batch | 1 | 0 | 0 | 13 |
| SUM: | 46 | 5526 | 7140 | 19237 |

The code is 85% in C with assembly³ for bottleneck optimizations and low-level I/O such as video or audio.

Source lines of code (SLOC) is not a meaningful metric against a single codebase but excels when it comes to extracting proportions. Commander Keen with its 19,237 SLOC is very small compared to most software. `curl` (a command-line tool to download url content) is 154,134 SLOC. Google's Chrome browser is 1,700,000 SLOC. Linux kernel is 15,000,000 SLOC.

³All the assembly in Keen is done with TASM (a.k.a Turbo Assembler by Borland). It uses Intel notation where the destination is before the source: `instr dest source`.

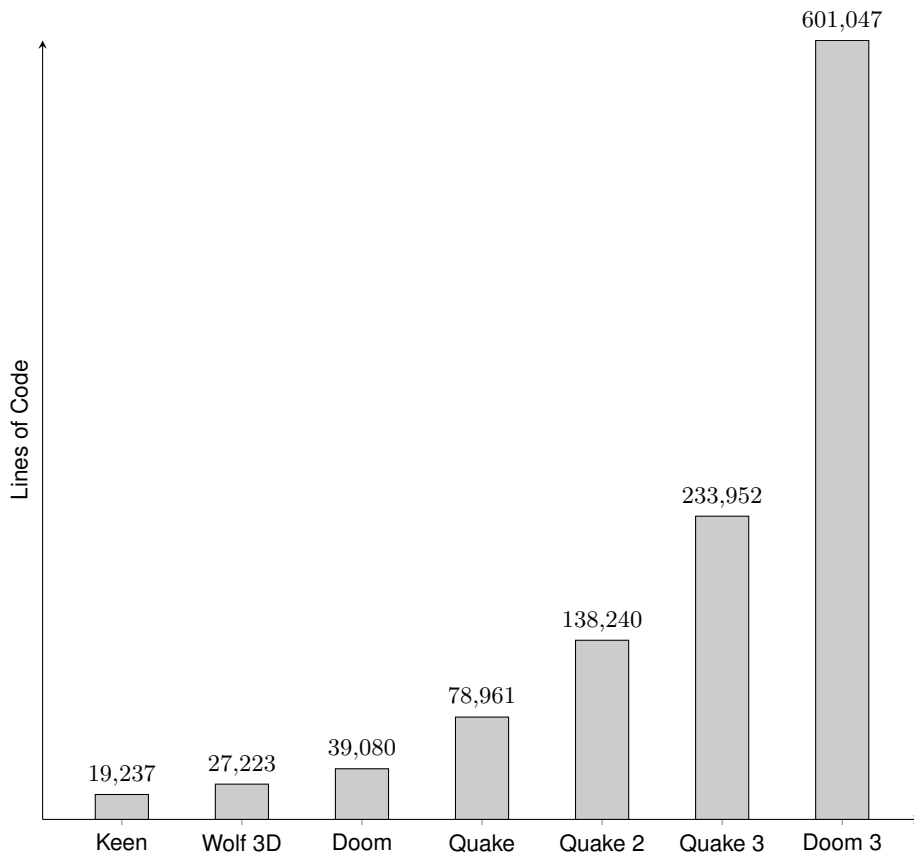


Figure 1: Lines of code from id Software game engines.

The archive contains more than just source code; it also features:

- `static` folder: Static header files for loading assets (as explained in Section ??).
- `lscr` folder: Load and decompress Softdisk data files.
- `README`: How to build the executable.

0.4 Compile source code

Now let's start to compile the source code. To compile the code like it's 1990 you need the following software:

- Commander Keen source code.

0.4. COMPILE SOURCE CODE

- DosBox.
- The Compiler Borland C++ 3.1.
- Commander Keen: Keen Dreams 1.13 shareware (for the assets).

After setting up the DosBox environment, with Borland C++ 3.1 installed (You can find a complete tutorial in "Let's compile like it's 1992" on fabiansanglard.net) download the source code via github.

Once you start DosBox and change directory to the `keen` folder, first create the folder where we create our compiled object files.

```
mkdir OBJ
```

Then we need to create the static OBJ header files.

```
chdir STATIC  
make.bat
```

Once the static object header files are created, move back to the `keen` folder and open Borland C++. Open the `kdreams.prj` project file. Before we can start compiling we need to set the correct directories. Select Options -> Directories and change the values as follow:



Figure 2: Borland C++ 3.1 directory settings

Now it's time to compile. Go to Compile -> Build all, and voila! The final step is to copy `kdreams.exe` to the Keen shareware folder. Now you can play your compiled version of Commander Keen.

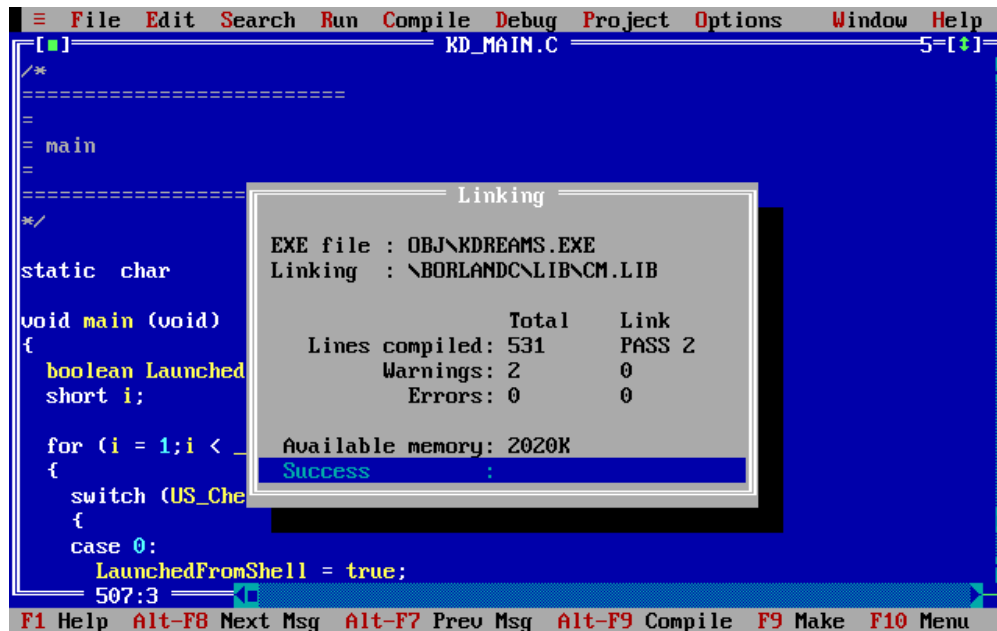


Figure 3: Commander Keen compiling

0.5 Big Picture

The game engine is divided in three blocks:

- Control panel which lets users configure and start the game.
- 2D game renderer where the users spend most of their time.
- Sound system which runs concurrently with either the Menu or 2D renderer.

The three systems communicate via shared memory. The renderer writes sound requests to the RAM (also making sure the assets are ready). These requests are read by the sound "loop". The sound system also writes to the RAM for the renderers since it is in charge of the heartbeat of the whole engine. The renderers update the screen according to the wall-time tracked by `TimeCount` variable.

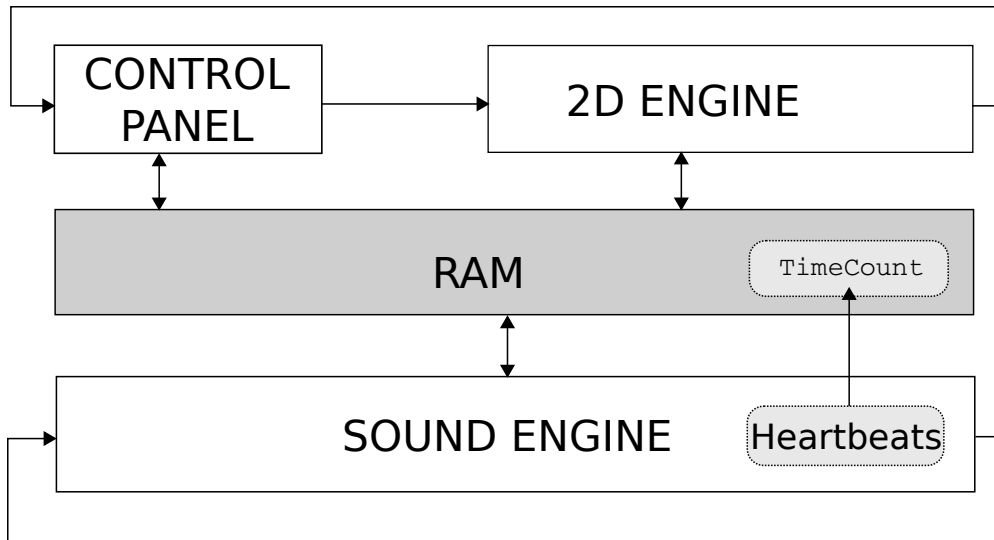


Figure 4: Game engine three main systems.

0.5.1 Unrolled Loop

With the big picture in mind, we can dive into the code and unroll the main loop starting in `void main()`. The control panel and 2D renderer are regular loops but due to limitations explained later, the sound system is interrupt-driven and therefore out of `main`. Because of real mode, C types don't mean what people would expect from a 16-bit architecture.

- `int` and `word` are 16 bits.
- `long` and `dword` are 32 bits.

The first thing the program does is set the text color to light grey and background color to black.

```
void main (void)
{
    textcolor(7);
    textbackground(0);

    InitGame();

    DemoLoop();           // DemoLoop calls Quit when
                          // everything is done
    Quit("Demo loop exited???");
}
```

In `InitGame`, a validation is performed to check if sufficient memory is available and brings up all the managers.

```
void InitGame (void)
{
    int i;

    MM_Startup ();        // Memory Manager

    US_TextScreen();      // Show intro screen

    VW_Startup ();        // Video Manager
    RF_Startup ();        // Refresh Manager
    IN_Startup ();        // Input Manager
    SD_Startup ();        // Sound Manager
    US_Startup ();        // User Manager

    CA_Startup ();        // Cache Manager
    US_Setup ();

    CA_ClearMarks ();     // Clears out all the marks

    CA_LoadAllSounds ();  // Load all sounds
}
```

Then comes the core loop, where the menu and 2D renderer are called forever.


```
void DemoLoop() {
    US_SetLoadSaveHooks();
    while (1) {
        VW_InitDoubleBuffer ();
        IN_ClearKeysDown ();
        VW_FixRefreshBuffer ();
        US_ControlPanel (); // Menu
        GameLoop ();
        SetupGameLevel ();
        PlayLoop () ; // 2D renderer (action)
    }
    Quit("Demo loop exited???");
}
```

PlayLoop contains the 2D renderer. It is pretty standard with getting inputs, update screen, and render screen approach.

```
void PlayLoop (void)
{
    FixScoreBox ();      // draw bomb/flower
    do
    {
        CalcSingleGravity ();    // Calculate gravity
        IN_ReadControl(0,&c);    // get player input

        // go through state changes and propose movements
        obj = player;
        do
        {
            if (obj->active)
                StateMachine(obj); // Enemies think
            obj = (objtype *)obj->next;
        } while (obj);

        [...]           // Check for and handle collisions
                        // between objects

        ScrollScreen(); // Scroll if Keen is nearing an edge.
                        // Draw new tiles to master screen in
                        // VRAM, and mark them in tile arrays

        [...]           // React to whatever happened, and post
                        // sprites to the refresh manager

        RF_Refresh();    // Copy marked tiles from master to
                        // buffer screen, and update sprites
                        // in buffer screen.
                        // Finally, switch buffer and view
                        // screen

        CheckKeys();     // Check special keys
    } while (!loadedgame && !playstate);
}
```

The interrupt system is started via the Sound Manager in `SDL_SetIntsPerSec(rate)`. While there is a famous game development library called Simple DirectMedia Layer (SDL), the prefix `SDL_` has nothing to do with it. It stands for Sound Low level (Simple DirectMedia Layer did not even exist in 1990).

The reason for interrupts is extensively explained in Chapter ?? "??". In short, with an OS

supporting neither processes nor threads, it was the only way to have something execute concurrently with the rest of the engine.

An ISR (Interrupt Service Routine) is installed in the Interrupt Vector Table to respond to interrupts triggered by the engine.

```
void SD_Startup(void)
{
    if (SD_Started)
        return;

    t0OldService = getvect(8); // Get old timer 0 ISR

    SDL_InitDelay(); // SDL_InitDelay() uses t0OldService

    setvect(8,SDL_t0Service); // Set to my timer 0 ISR

    SD_Started = true;
}
```

0.6 Architecture

The source code is structured in two layers. `KD_*` files are high-level layers relying on low-level `ID_*` sub-systems called Managers interacting with the hardware.

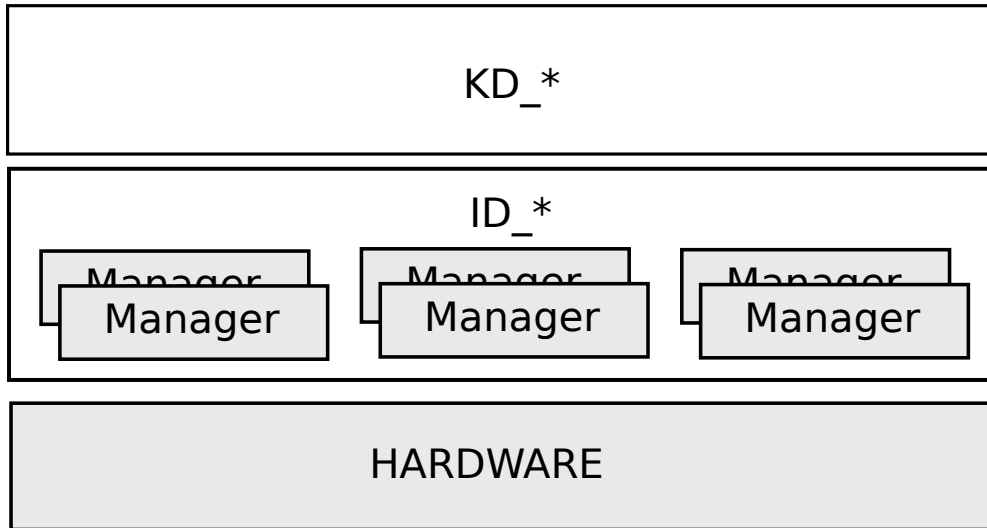


Figure 5: Commander Keen source code layers.

There are six managers in total:

- Memory
- Video
- Cache
- Sound
- User
- Input

The `KD_*` stuff was written specifically for Commander Keen while the `ID_*` managers are generic and later re-used (with improvements) for newer ID games (Hovertank One, Catacomb 3-D and Wolf3D).

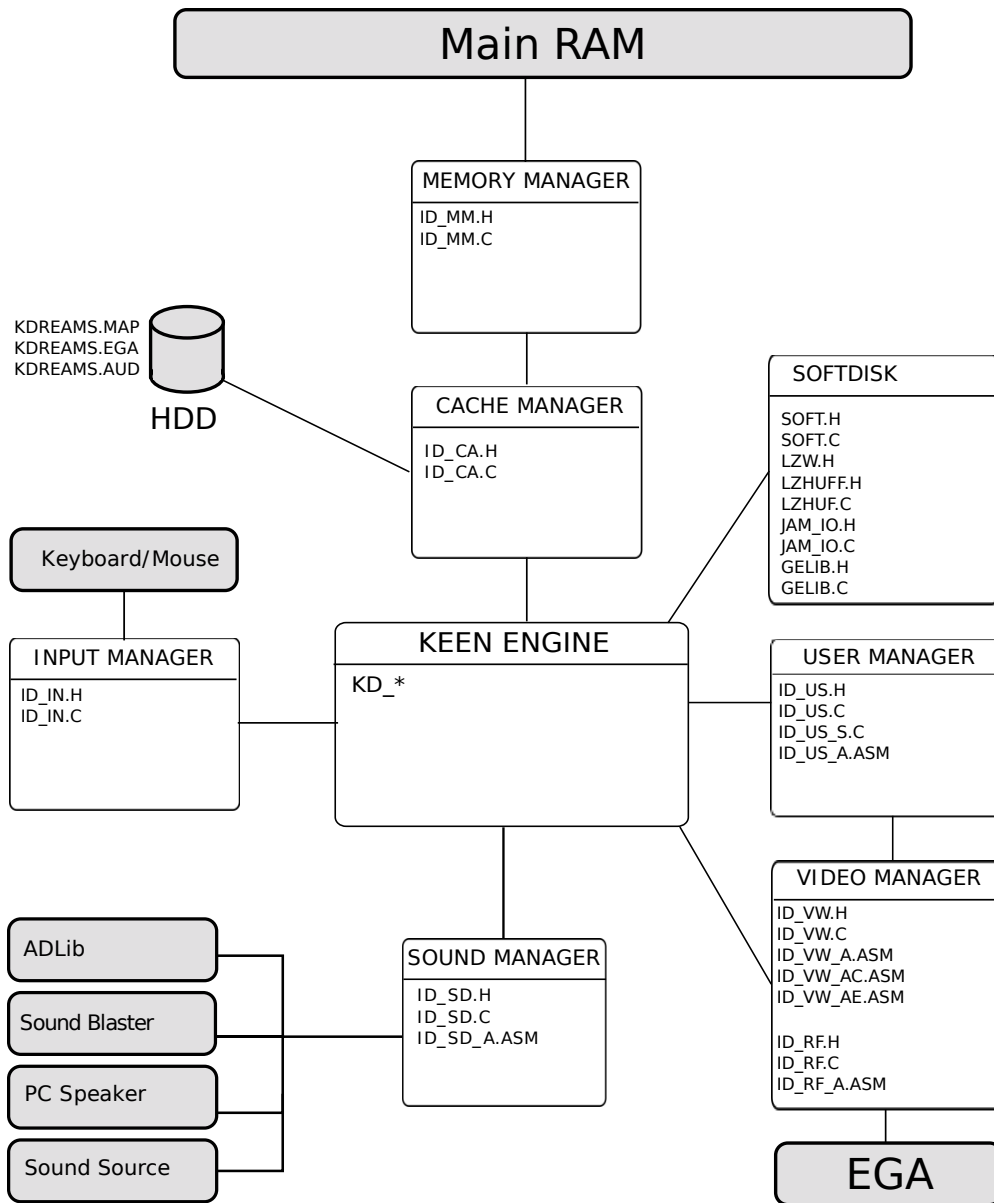


Figure 6: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

Next to the hard drives (HDD) you can see the assets packed as described in Chapter ??.

0.6.1 Memory Manager (MM)

The engine has its own memory manager, to have more control over memory fragmentation and optimization. The memory manager is made of a linked list of "blocks" keeping track of the RAM. A block points to a starting point in RAM, has a size and can be marked with attributes:

- LOCKBIT : This block of RAM cannot be moved during compaction.
- PURGEBITS : Two levels available, 0= unpurgeable, 3=purge first.

```
typedef struct mmblockstruct
{
    unsigned    start,length;
    unsigned    attributes;
    memptr      *useptr;
    struct mmblockstruct far *next;
} mmblocktype;
```

The memory manager starts by allocating all RAM, starting from 00000h, and assigning each block to the linked list. Since the engine is compiled using the medium memory model, there are two heaps available: the near heap between the global variables and stack, and the far heap starting right behind the stack. The total available free memory space for the near heap can be obtained by

```
length=coreleft();
start = (void far *) (nearheap = malloc(length));
```

The memory manager allocates all memory blocks in size of segments, meaning each block is provided in step size of 16 bytes. Therefore, the start and length of the memory block must be segment aligned.

```
length -= 16-(FP_OFF(start)&15); // round to 16 bytes
seglength = length / 16;         // now in segments
segstart = FP_SEG(start)+(FP_OFF(start)+15)/16;
```

The first allocated block is the unusable block of memory from segment 0000h to the start of the near heap.

```
GETNEWBLOCK;
mmhead = mmnew;      // this will always be the first node
mmnew->start = 0;
mmnew->length = segstart;
mmnew->attributes = LOCKBIT;
```

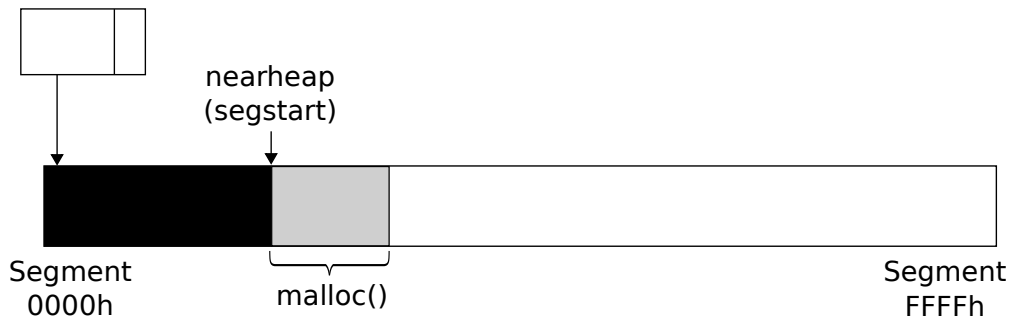


Figure 7: First locked block of unusable memory.

The stack is the next unusable block of memory. As the stack will grow/shrink during game execution, an additional part of the near heap free memory must be reserved for the stack.

```
#define SAVENEARHEAP 0x400 //space to leave in data segment
length -= SAVENEARHEAP;    //Reduce length of near heap
```

The next step is allocate the far heap, which can be obtained by

```
length=farcoreleft();
start = farheap = farmalloc(length);
```

The second block of unusable memory is the start of the stack memory, including the reserved memory block, until the start of the far heap.

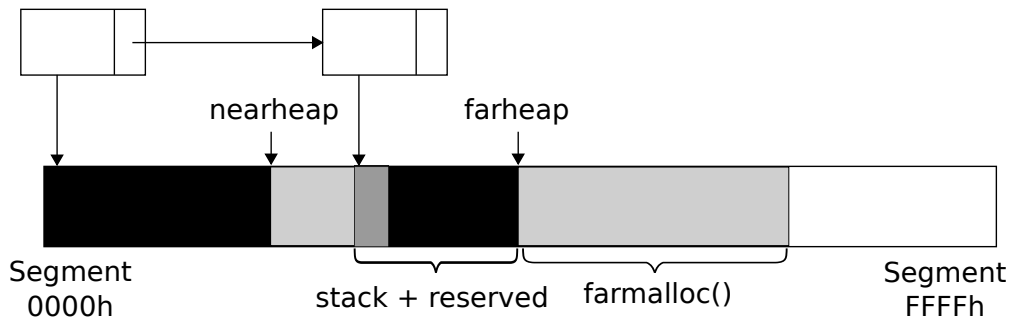


Figure 8: Second locked block: stack.

Finally, the last block of unusable memory is between the end of the far heap and the end of the 1MiB memory, the area that is typically for VRAM, ROM, etc. Now the entire memory starting at segment 0000h up to FFFFh is allocated and chained together, and under full

control of the memory manager.

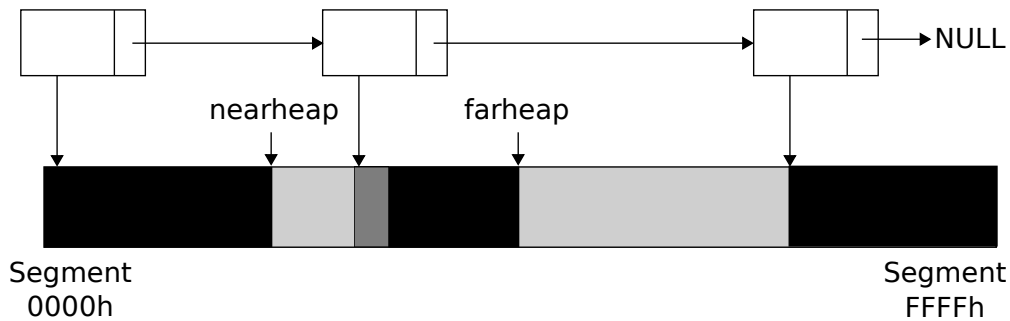


Figure 9: Final initial memory manager state.

The engine interacts with the Memory Manager by requesting RAM (`MM_GetPtr`) and freeing RAM (`MM_FreePtr`). To allocate memory, the manager searches for "holes" between blocks⁴.

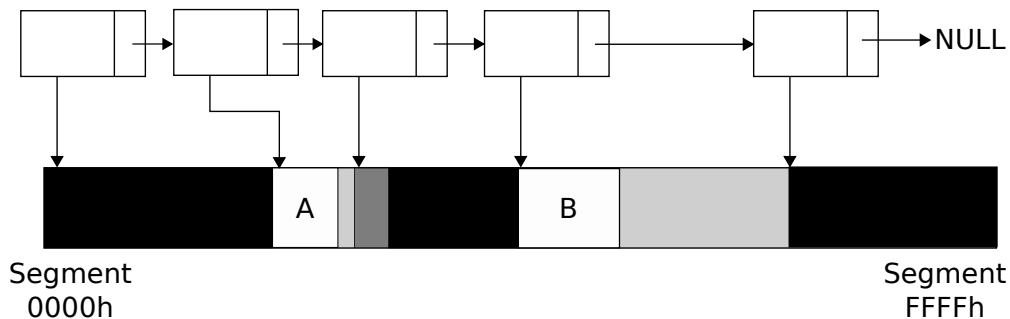


Figure 10: Allocation of memory.

By calling `MM_ShowMemory` you can inspect at any moment during game play the current memory usage. Each pixel represents 16 bytes (one segment increment). The red area represents blocked memory, black is available memory, blue is unpurgeable and purple is purgeable memory. The small white pixels are the start of a new memory block from the linked list. Notice how small the size of the near heap is (first black line) compared with the large heap!

⁴See the "Game Engine Black book Wolfenstein 3D" for a detailed description how the memory manager is allocating memory

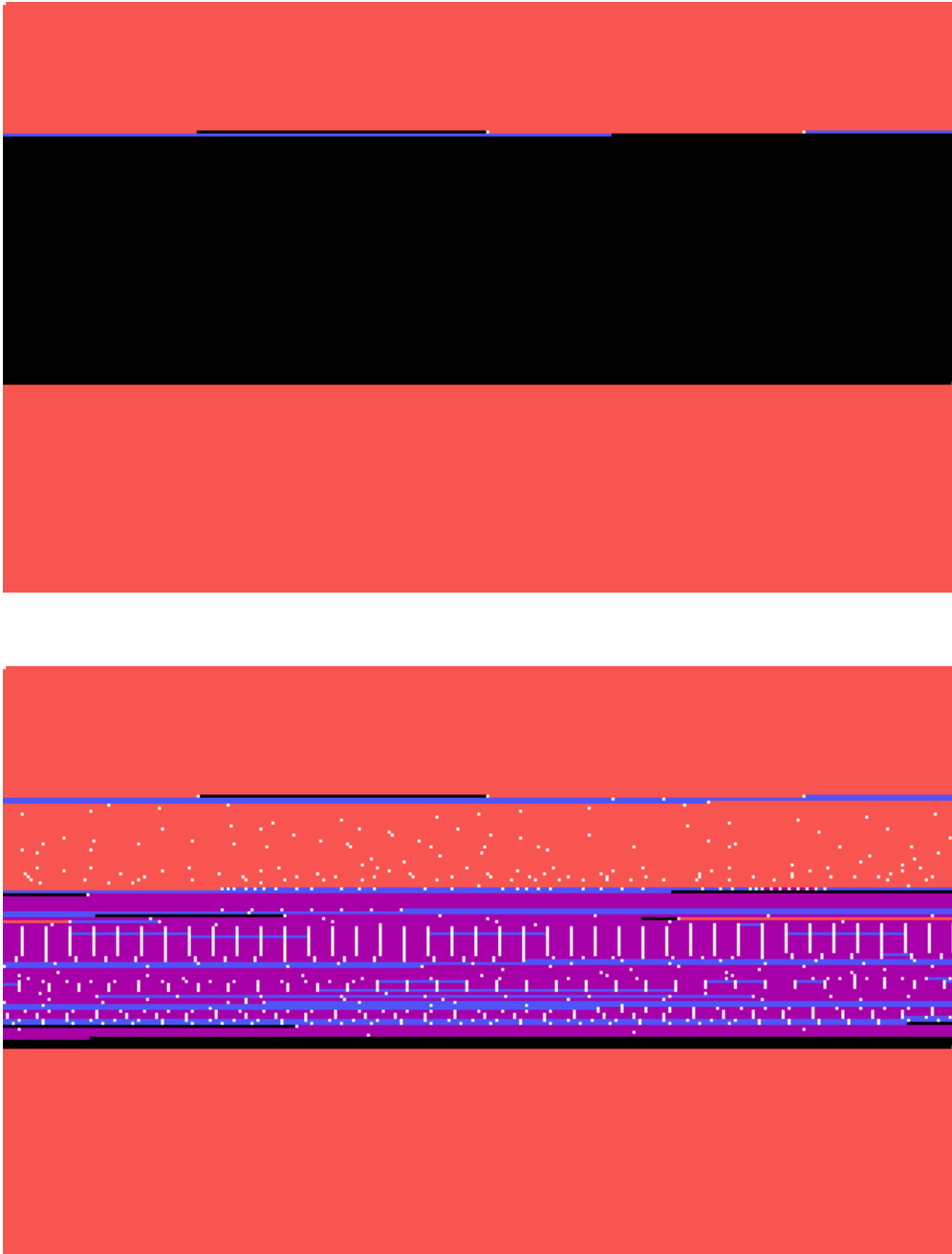


Figure 11: Memory dump after (i) initializing memory manager and (ii) running the game.

0.6.2 Video Manager (VW & RF)

The video manager features two parts:

- The `VW_*` layer is made of both C and ASM, where the C functions abstract away EGA register manipulation via assembly routines.
- The `RF_*` layer is used to refresh the screen, and is also made of both C and ASM code.

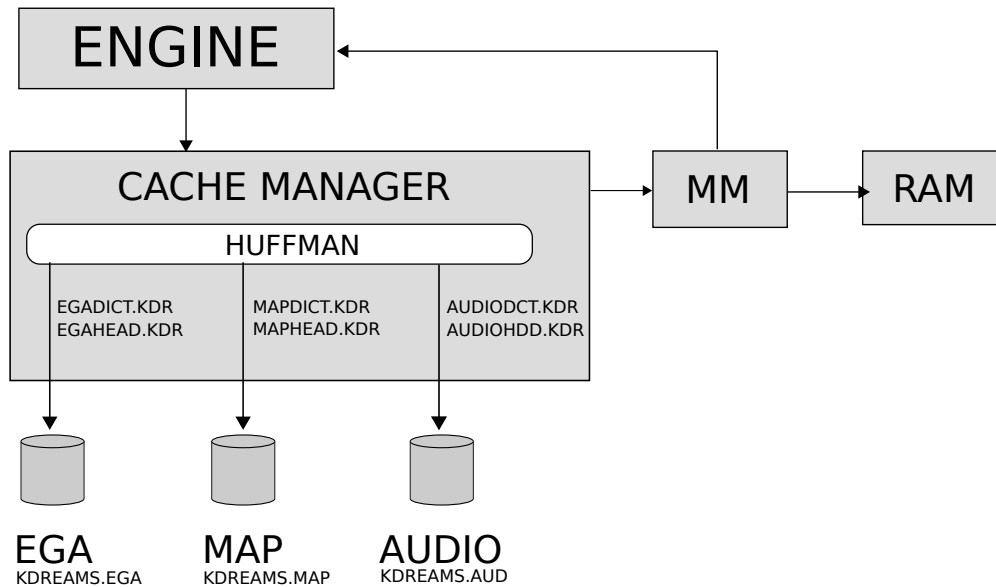
The video manager is described extensively in section ?? on page ??.

0.6.3 Cache Manager (CA)

The cache manager is a small but critical component. It loads and decompresses maps, graphics and audio resources stored on the filesystem and makes them available in RAM. Assets are stored into three files:

- A header file containing the offset to allow translation from asset ID to byte offset in the data file.
- A compression dictionary to decompress each asset.
- The data file containing the assets

Details of each asset file are explained in chapter ?. The header and dictionary files are provided with the source code in the `static` folder and contain `*.KDR` extension. Both file types are integrated in the engine code and required during compilation (they are converted into an `OBJ` file using `makeobj.c`). The data file containing the assets is not part of the source code and must be acquired via downloading the shareware version. All resources are compressed using a traditional Huffman method for (de-)compression, and the maps have an additional RLEW compression.



0.6.4 User Manager (US)

The user manager is responsible for text layout and control panels like loading and saving games, configure controls and setting sound device.

Once we start the game, we move the display to EGA graphic mode 0x0D. Here we can't directly print characters on the screen anymore. So a key function of the User Manager is to print text on a given pixel location. In the graphic assets file the complete font is stored with the following information for each character:

- The width of the character
- The location in memory where each character is stored as a bitmap

Each character has the same height of 10 pixels, but the character width could vary as illustrated in Figure 12.

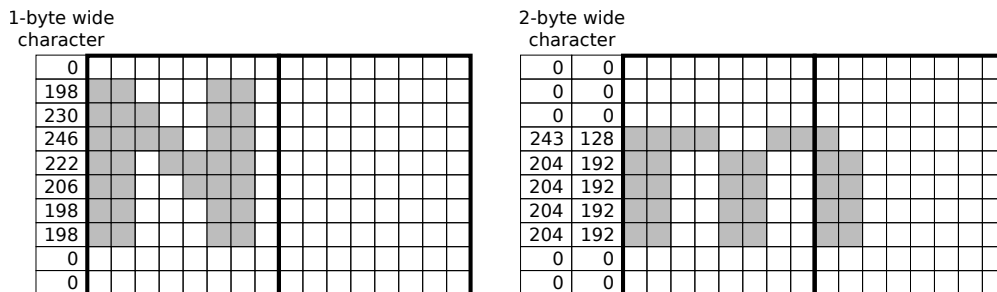


Figure 12: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

As explained in section ?? on page ??, each 8 pixels are represented by 1 byte per memory bank. So how do we print a character which is not perfectly aligned with the memory layout? Here a trick of bitshift tables is being used.

The default table (`shiftdata0`) is defined as integer (16 bits) and contains all values from 0-255. Now, we shift this entire table 1 bit to the right. We can translate the bit shift back into an integer and store these values again in a table (`shiftdata1`). We can do this again when we shift another bit, until we cycled through the 8-bits. So at the end we have created 8 shift tables to fully cycle through 8-bits. In Figure 13 the bitshift for the values '198' is illustrated.

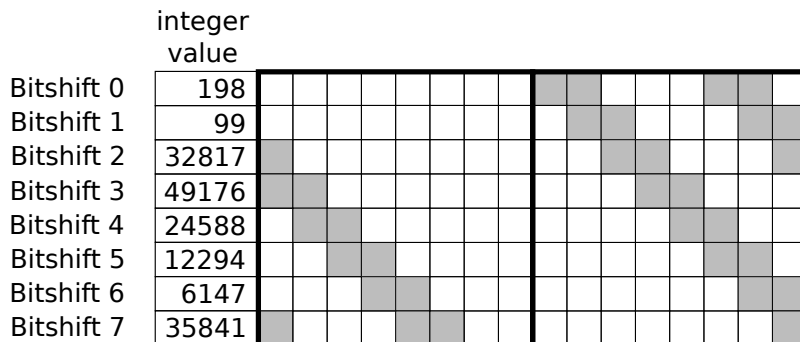


Figure 13: Right bitshift [0-7] for 198.

Each bitshift table is generated in `id_vw_a.asm`, see below bitshift 5.

```

LABEL shiftdata5 WORD
dw 0, 2048, 4096, 6144, 8192,10240,12288,14336,16384,18432,20480,22528,24576,26624
dw 28672,30720,32768,34816,36864,38912,40960,43008,45056,47104,49152,51200,53248,55296
dw 57344,59392,61440,63488, 1, 2049, 4097, 6145, 8193,10241,12289,14337,16385,18433
dw 20481,22529,24577,26625,28673,30721,32769,34817,36865,38913,40961,43009,45057,47105
dw 49153,51201,53249,55297,57345,59393,61441,63489, 2, 2050, 4098, 6146, 8194,10242
dw 12290,14338,16386,18434,20482,22530,24578,26626,28674,30722,32770,34818,36866,38914
dw 40962,43010,45058,47106,49154,51202,53250,55298,57346,59394,61442,63490, 3, 2051
dw 4099, 6147, 8195,10243,12291,14339,16387,18435,20483,22531,24579,26627,28675,30723
dw 32771,34819,36867,38915,40963,43011,45059,47107,49155,51203,53251,55299,57347,59395
dw 61443,63491, 4, 2052, 4100, 6148, 8196,10244,12292,14340,16388,18436,20484,22532
dw 24580,26628,28676,30724,32772,34820,36868,38916,40964,43012,45060,47108,49156,51204
dw 53252,55300,57348,59396,61444,63492, 5, 2053, 4101, 6149, 8197,10245,12293,14341
dw 16389,18437,20485,22533,24581,26629,28677,30725,32773,34821,36869,38917,40965,43013
dw 45061,47109,49157,51205,53253,55301,57349,59397,61445,63493, 6, 2054, 4102, 6150
dw 8198,10246,12294,14342,16390,18438,20486,22534,24582,26630,28678,30726,32774,34822
dw 36870,38918,40966,43014,45062,47110,49158,51206,53254,55302,57350,59398,61446,63494
dw 7, 2055, 4103, 6151, 8199,10247,12295,14343,16391,18439,20487,22535,24583,26631
dw 28679,30727,32775,34823,36871,38919,40967,43015,45063,47111,49159,51207,53255,55303
dw 57351,59399,61447,63495

```

Now let's take the example of printing 'N', with an offset of 3 pixels. A simple lookup in `shiftdata3` results in the 3-bit shifted 'N'. Note that you first display the low byte and then the high byte value.

| unshifted bitmap value | shiftdata3 bitmap value | | | | | | | | | | | | | | | | |
|---------------------------|----------------------------|----------|--|--|--|--|--|--|--|-----------|--|--|--|--|--|--|--|
| 0 | 0 | | | | | | | | | | | | | | | | |
| 198 | 49176 | | | | | | | | | | | | | | | | |
| 230 | 49180 | | | | | | | | | | | | | | | | |
| 246 | 49182 | | | | | | | | | | | | | | | | |
| 222 | 49179 | | | | | | | | | | | | | | | | |
| 206 | 49177 | | | | | | | | | | | | | | | | |
| 198 | 49176 | | | | | | | | | | | | | | | | |
| 198 | 49176 | | | | | | | | | | | | | | | | |
| 0 | 0 | | | | | | | | | | | | | | | | |
| 0 | 0 | | | | | | | | | | | | | | | | |
| | | Low byte | | | | | | | | High byte | | | | | | | |

Figure 14: Bitshift 'N' over 3 bits using bit shift tables.

Once both bytes are copied to the data buffer, the buffer pointer is increased with the character width and then the next character is copied. Finally, the data buffer is copied to VRAM, so it gets displayed on the screen.

```
charloc    = 2        ;pointers to every character
BUFFWIDTH  = 50        ;buffer width is 50 characters

PROC   ShiftPropChar NEAR

    mov es,[grsegs+STARTFONT*2] ;segment of font to use
    mov bx,[es:charloc+bx]       ;BX holds pointer to
        character data

; look up which shift table to use, based on bufferbit
    mov di,[bufferbit]           ;pixel offset within byte [0-7]
    shl di,1
    mov bp,[shifttabletable+di] ;BP holds pointer to shift
        table

    mov di,OFFSET databuffer
    add di,[bufferbyte]          ;DI holds pointer to buffer
    mov cx,[es:pcharheight]     ;CX contains character height
    mov dx,BUFFWIDTH

; write one byte character
shift1wide:
    dec dx
EVEN
@@loop1:
    SHIFTNOR
    add di,dx                    ; next line in buffer
    loop @@loop1
    ret
ENDP

; Macros to table shift a byte of font
MACRO SHIFTNOR
    mov al,[es:bx]              ; source of font data
    xor ah,ah
    shl ax,1
    mov si,ax
    mov ax,[bp+si]              ; table shift into two bytes
    or [di],al                  ; OR with first byte
    inc di
    mov [di],ah                 ; replace next byte
    inc bx                      ; next source byte
ENDM
```

0.6.5 Sound Manager (SD)

The Sound Manager abstracts interaction with all four sound systems supported: PC Speaker, AdLib, Sound Blaster, and Disney Sound Source. It is a beast of its own since it doesn't run inside the engine. Instead it is called via IRQ at a much higher frequency than the engine (the engine runs at a maximum 70Hz, while the sound manager ranges from 140Hz to 700Hz). It must run quickly and is therefore written in small and fast routines.

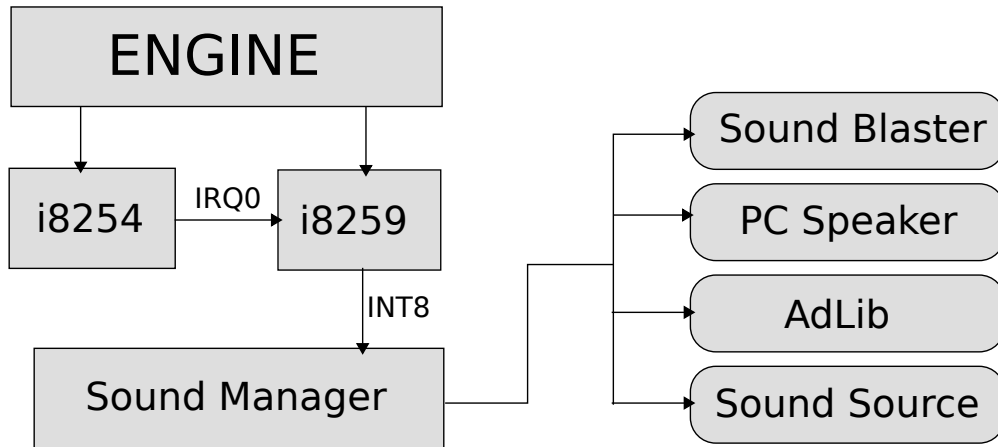


Figure 15: Sound system architecture.

The sound manager is described extensively in the "Sound and Music" section.

0.6.6 Input Manager (IN)

The input manager abstracts interactions with joystick, keyboard, and mouse. It features the boring boilerplate code to deal with PS/2, Serial, and DA-15 ports, with each using their own I/O addresses.

0.6.7 Softdisk files

The only function for the Softdisk files is to load and show the intro screen bitmap, using LoadLIBShape from soft.c. Most of the functions in these files are actually not used and therefor not further discussed in this book.

0.7 Startup

As the game engine starts, it will first load the memory manager. Then it will check if there is at least 335KiB of RAM available. If not, it gives a warning, but you can continue with the game. But most likely somewhere soon the game will either crash or receives an "Out of memory" error.

After successfully starting the game the intro image is displayed, which is a Deluxe Paint-Bitmap image (*.LBM). After the user has hit any key, the intro image is unloaded from RAM to make more room for runtime and the control panel is shown.



Figure 16: Keen Dreams intro screen

