0.1 About the Source Code

Commander Keen episodes 1-5 source code is unavailable, as the current owner Zenimax[1] has, at the time of writing, shown no interest in selling the intellectual property. 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 owners of Softdisk at the time, 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 itâĂŹs 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 available on `github`. It is essential to use the source code from the shareware version 1.13, or you may encounter issues due to incompatible map and graphs headers[2]. To get the correct source code, use the following commands in the console

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

0.3 First Contact

After downloading the repository from `github`, a folder named 'keen' is created, containing all the source files. The `cloc.pl` tool can be used to analyze the folder and generate statistics about the source code. This tool is useful for getting an idea of what to expect.

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

[2] See 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
---------------------------------------------------------
```

Approximately 85% of the code is in C, with assembly[3] used for bottleneck optimizations and low-level I/O operations such as video and audio handling.

Source lines of code (SLOC) are not particularly meaningful for comparing a single code-base but are helpful for determining the proportion of code in various components. Commander Keen has 19,237 SLOC, which is small compared to most modern software. For instance, `curl` (a command-line tool for downloading URLs) has 154,134 SLOC, Google Chrome has 1,700,000 SLOC, and the Linux kernel consists of 15,000,000 SLOC.

[3]All 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`.

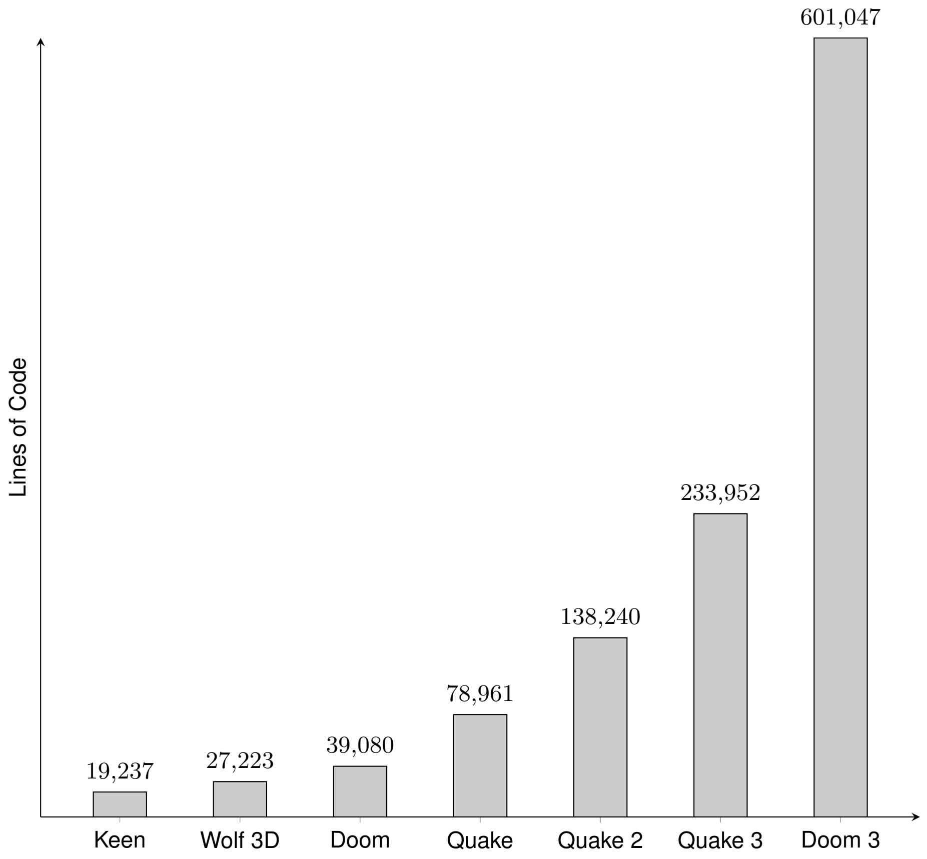


Figure 1: Lines of code from id Software game engines.

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

- A `static` folder with static header files for loading assets (as explained in Section **??**).
- An `lscr` folder for loading and decompressing Softdisk data files.
- A `README` file explaining 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.
- DosBox.
- The Compiler Borland C++ 3.1.
- Commander Keen: Keen Dreams 1.13 shareware (for the assets).

After setting up DOSBox and installing Borland C++ 3.1[4], download the source code from github.

Once DOSBox is running and you've navigated to the `keen` folder, create a folder to store your compiled object files.

```
mkdir OBJ
```

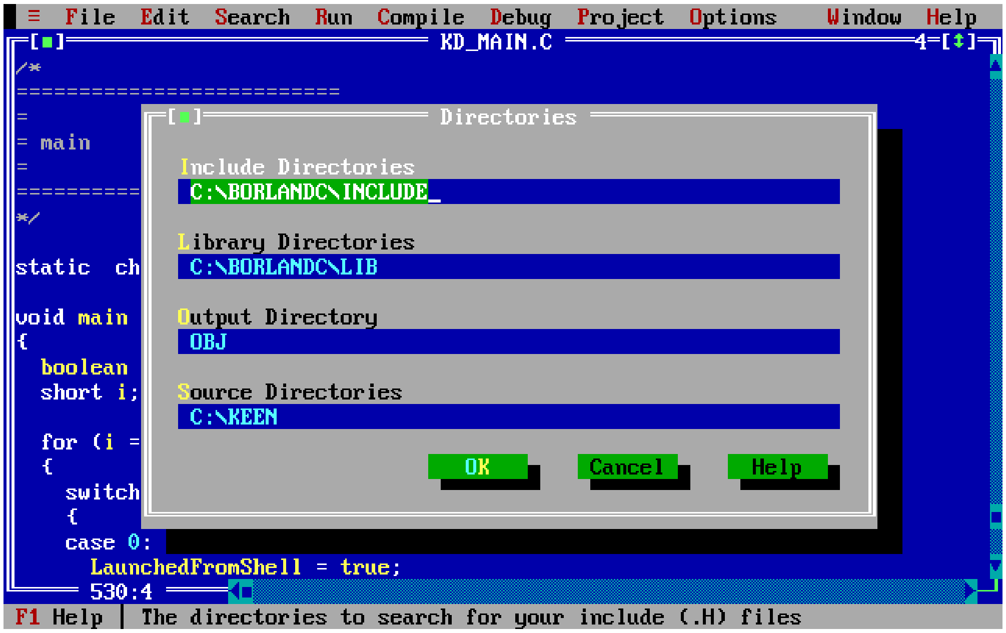
Then, generate the static `OBJ` header files.

```
chdir STATIC
make.bat
```

After that, return to the `keen` folder and open Borland C++. Open the `kdreams.prj` project file. Before compiling, adjust the directory settings by selecting Options -> Directories and change the values as shown on the next page.

Now it's time to compile. Go to Compile -> Build All, and voilà! The final step is to copy `kdreams.exe` into the Keen shareware folder. ou can now play your compiled version of *Commander Keen*.

[4]you can find a complete tutorial in "Let's compile like it's 1992" on fabiensanglard.net


≡ File Edit Search Run Compile Debug Project Options Window Help
KD_MAIN.C
Directories
OK
Cancel
Help
F1 Help

```
/*
==========================
=
= main
=
==========
*/

static  ch

void main
{
  boolean
  short i;

  for (i =
  {
    switch
    {
    case 0:
      LaunchedFromShell = true;
```

```
530:4
```

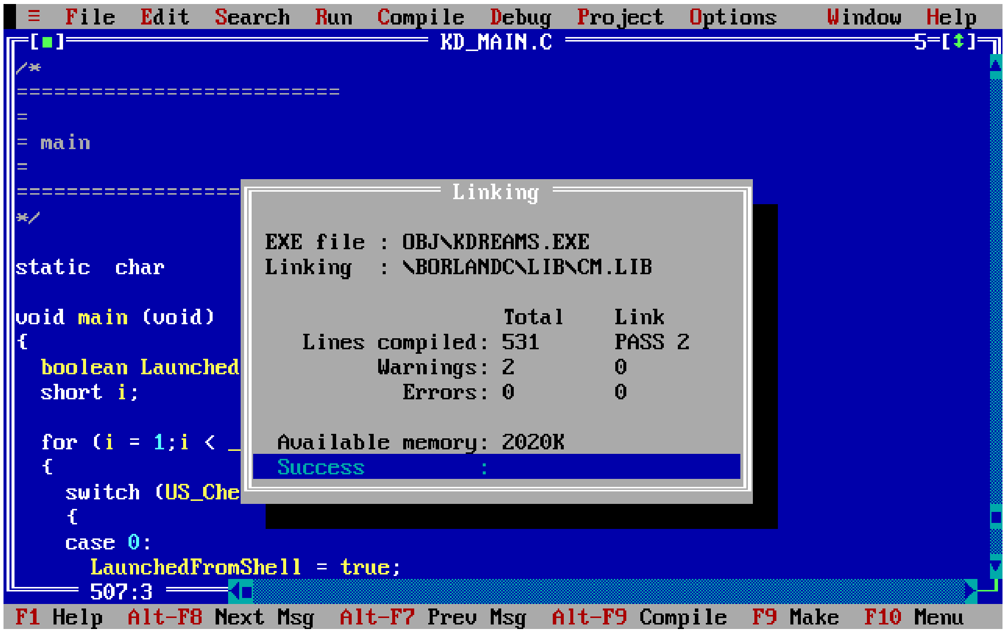
Include Directories
C:\BORLANDC\INCLUDE

Library Directories
C:\BORLANDC\LIB

Output Directory
OBJ

Source Directories
C:\KEEN

The directories to search for your include (.H) files


≡ File Edit Search Run Compile Debug Project Options Window Help
KD_MAIN.C
Linking
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
/*
==========================
=
= main
=
==================
*/

static  char

void main (void)
{
  boolean Launched
  short i;

  for (i = 1;i < _
  {
    switch (US_Che
    {
    case 0:
      LaunchedFromShell = true;
```

```
507:3
```

```
EXE file  : OBJ\KDREAMS.EXE
Linking   : \BORLANDC\LIB\CM.LIB

                  Total     Link
   Lines compiled: 531       PASS 2
         Warnings: 2         0
           Errors: 0         0

 Available memory: 2020K
 Success         :
```

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.

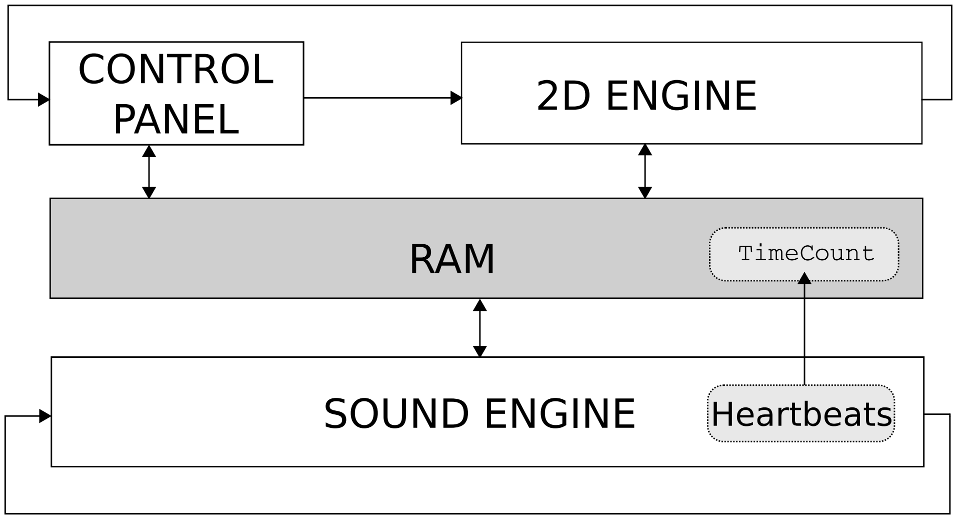


Figure 2: Game engine three main systems.

0.5.1 Unrolled Loop

With the big picture in mind, we can dive into the code and unroll the main loop, starting with `void main()`. The control panel and 2D renderer follow regular loops, but due to limitations discussed later, the sound system is interrupt-driven and therefore operates outside the main loop. In real mode, C types do not behave as expected on a 16-bit architecture.

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

The first action taken by the program is to set the text color to light grey and the background to black.

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

  InitGame();

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

In `InitGame()`, the program checks whether there is enough memory 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);

    [...]              // Handle collisions between objects
    ScrollScreen(); // Scroll if Keen is nearing an edge.
    [...]              // React to whatever happened.
    RF_Refresh();      // Update buffer screen and 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.

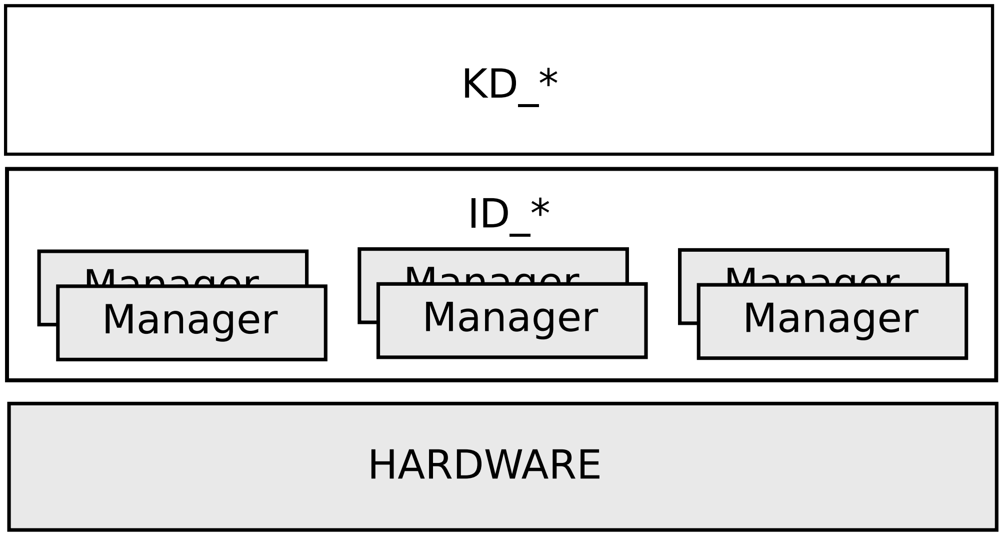


Figure 3: Commander Keen source code layers.

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).

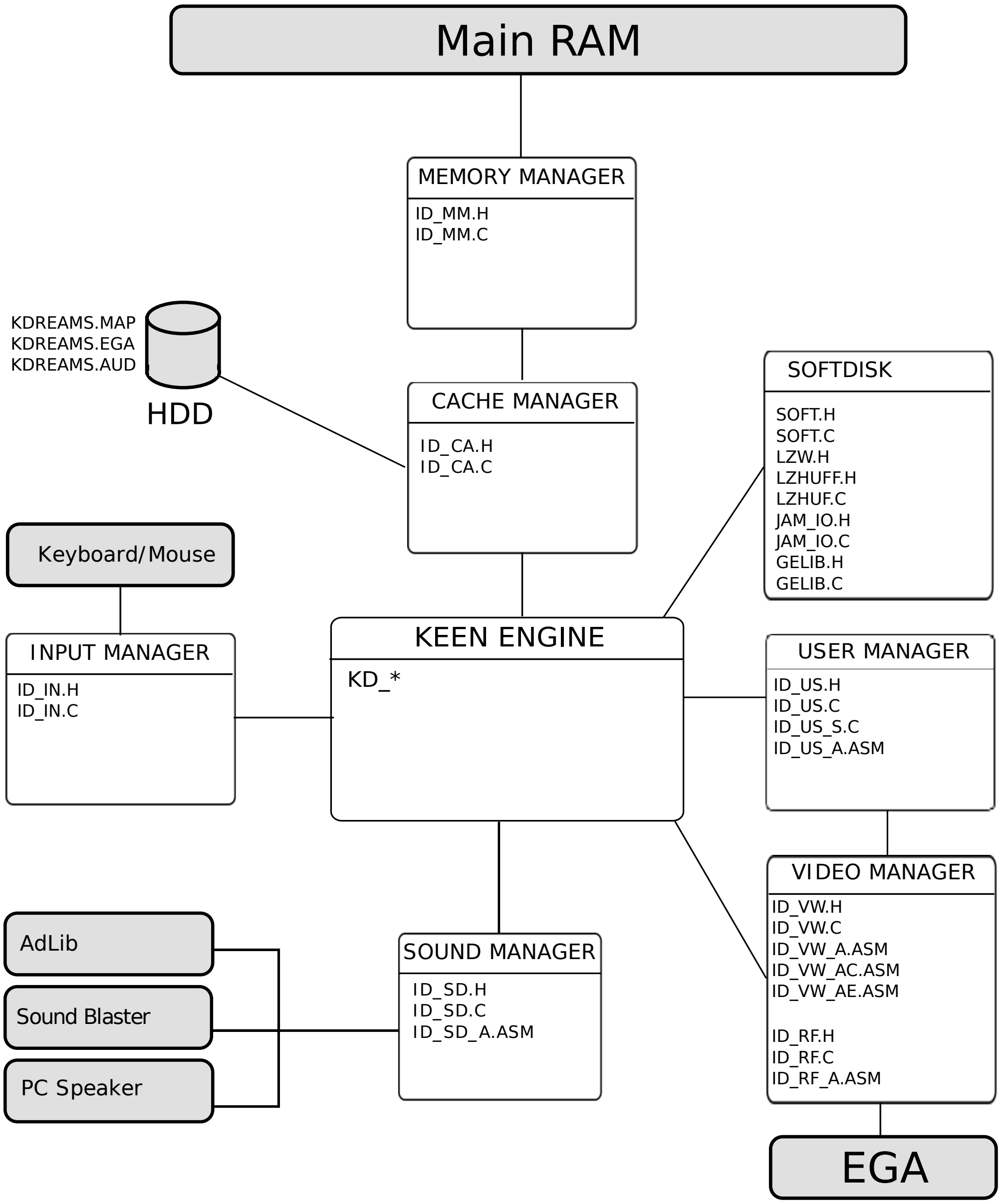


Figure 4: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

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 allocates all RAM, starting from `00000h`, and assigns segments to the linked list. Since the engine is compiled using the medium memory model, there are two heaps: 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 is segment-aligned, meaning it allocates memory blocks in chunks 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 block allocated is the unusable memory from segment `0000h` to the start of the near heap.

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

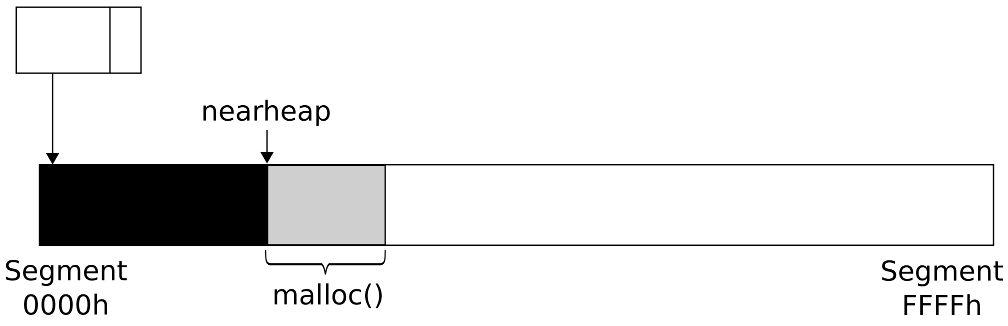


Figure 5: First locked block of unusable memory.

The stack is the next unusable memory block. Since the stack will grow or shrink during game execution, an additional part of the near heapâĂŹs free memory must be reserved for it.

```
#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 block of unusable memory is between the start of the stack memory (including the reserved block) and the start of the far heap.

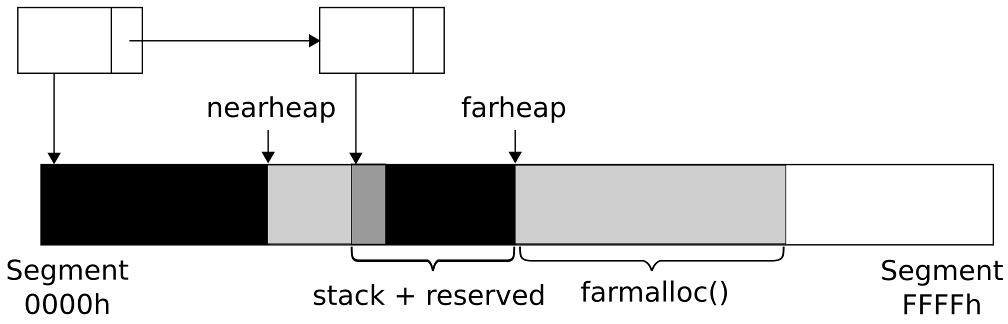


Figure 6: Second locked block: stack.

Finally, the last unusable block of memory lies 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 space from segment `0000h` to `FFFFh` is allocated, chained together, and fully controlled by the memory manager.

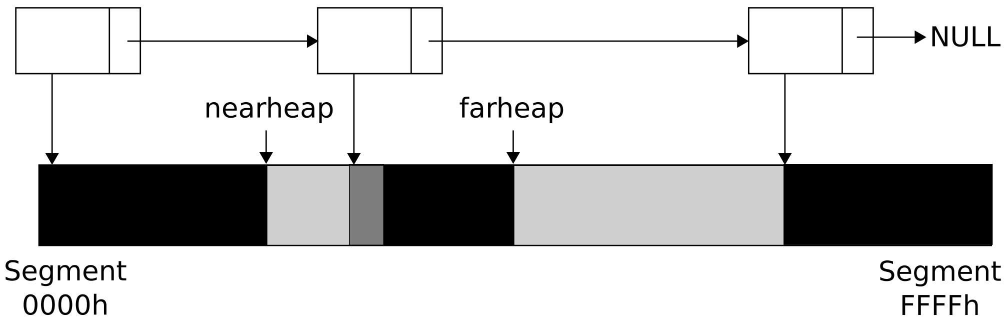


Figure 7: Final initial memory manager state.

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 blocks[5].

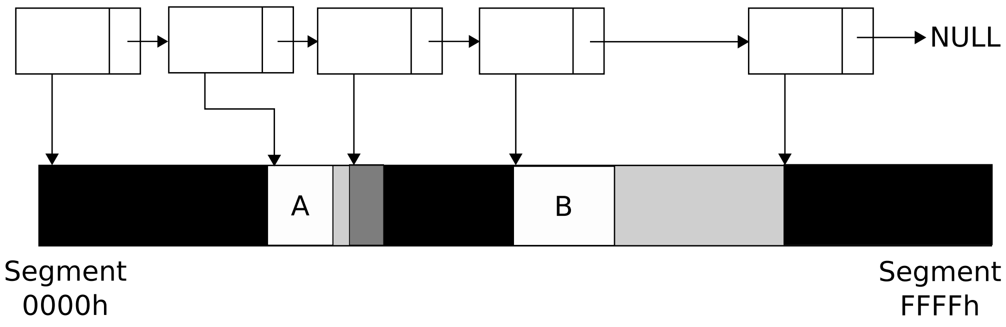
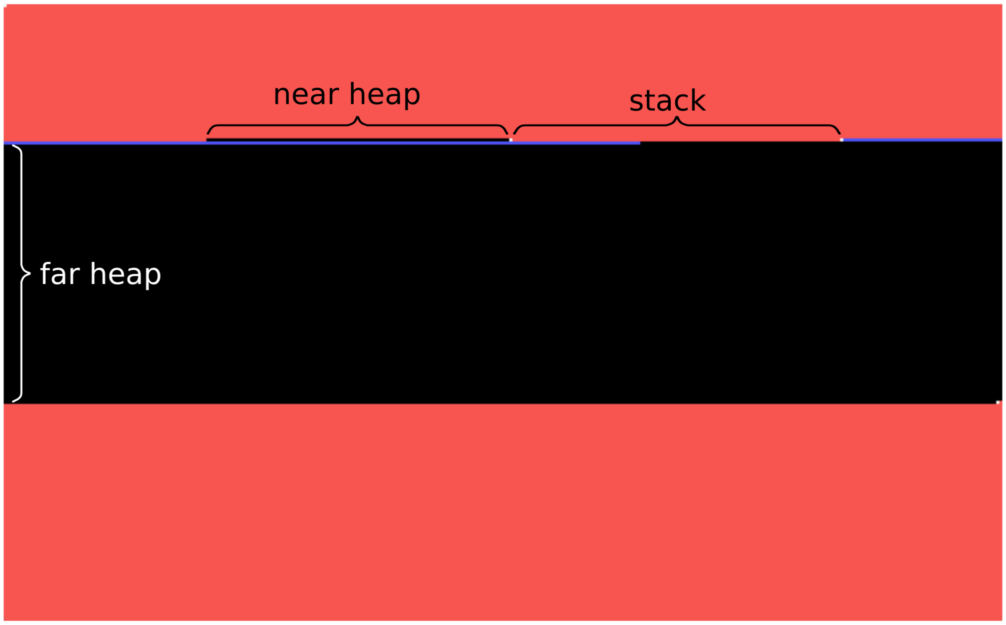


Figure 8: Allocation of memory.

By calling `MM_ShowMemory`, you can inspect the current memory usage during gameplay. Each pixel represents one memory segment of 16 bytes. Red indicates blocked memory, black indicates available memory, blue is unpurgeable memory, and purple is purgeable memory. Small white pixels indicate the start of new memory blocks in the linked list. Notice how small the near heap (the first black line) is compared to the far heap!

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



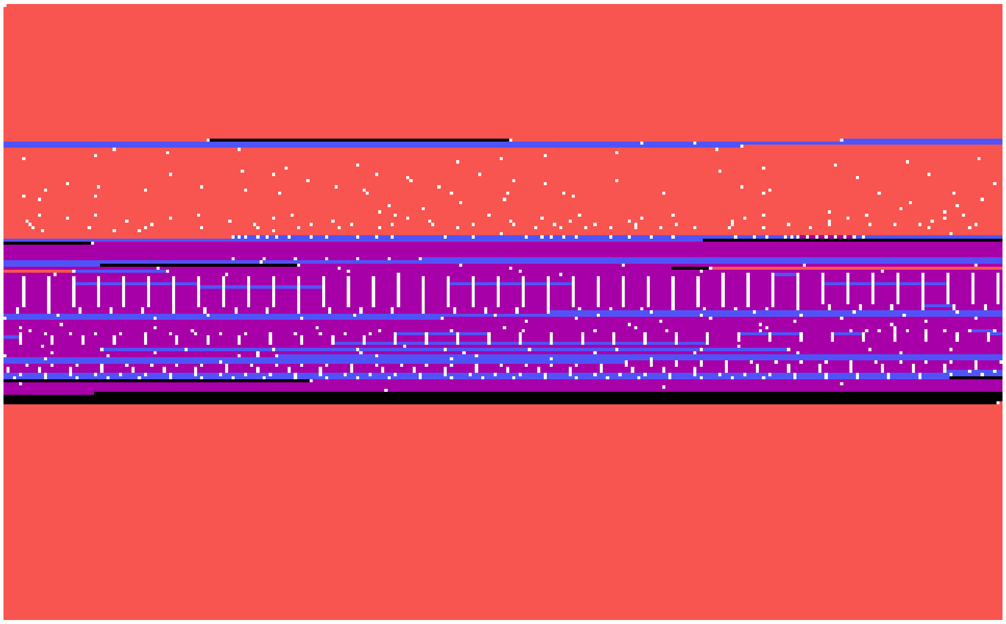


Figure 9: Memory dump after (i) initializing memory manager and (ii) running the game.

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 **??**.

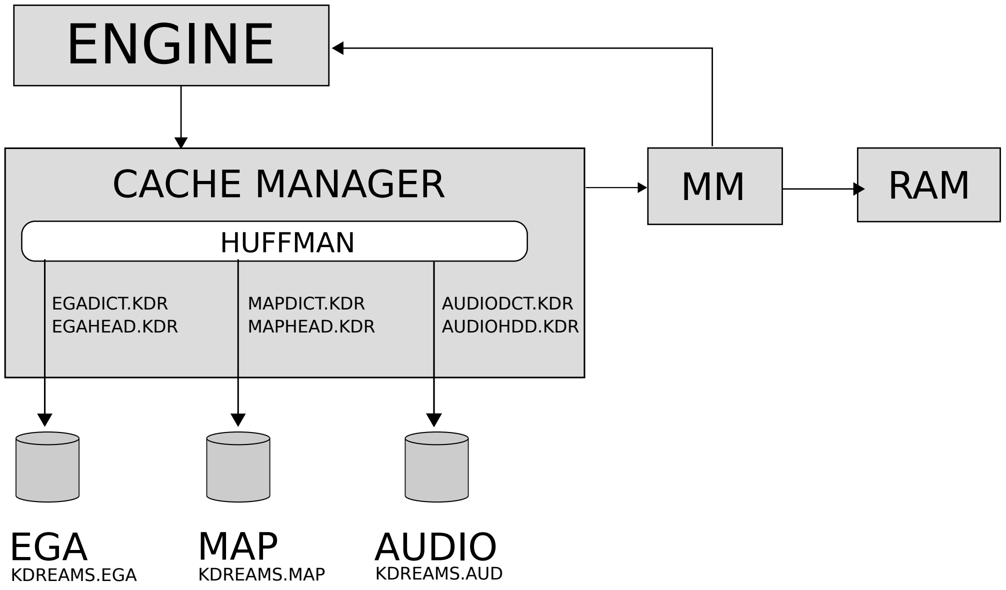
0.6.3 Cache Manager (CA)

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 on each asset file are explained in Chapter **??**. The header and dictionary files are provided in the `static` folder with the `*.KDR` extension. These file types are integrated into the engine code and are required during compilation (converted into an `*.OBJ` file using `makeobj.c`).

The data file containing the assets is not part of the source code and must be obtained by downloading the shareware version. All resources are compressed using Huffman compression, and maps have additional RLEW compression. The cache manager is described extensively in the "Asset Caching and Compression" section.



0.6.4 User Manager (US)

The user manager is responsible for the layout of text and control panels, such as loading/saving games, configuring controls, and setting sound devices. Once we start the game, we move the display to EGA graphic mode `0x0D`. Here we cannot print characters on the screen using the `printf()` command.

A key function of the user manager is to render text at a specific pixel location. When the engine needs to draw a string, it is passed to `US_Print` which does all measurement (`VW_MeasurePropString`) and then passes this information to `VWB_DrawPropString`, which takes care of drawing the string on screen.

In the graphics asset file, each font character is stored with:

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

Each character is 10 pixels tall, but the width varies, as illustrated in Figure 10.

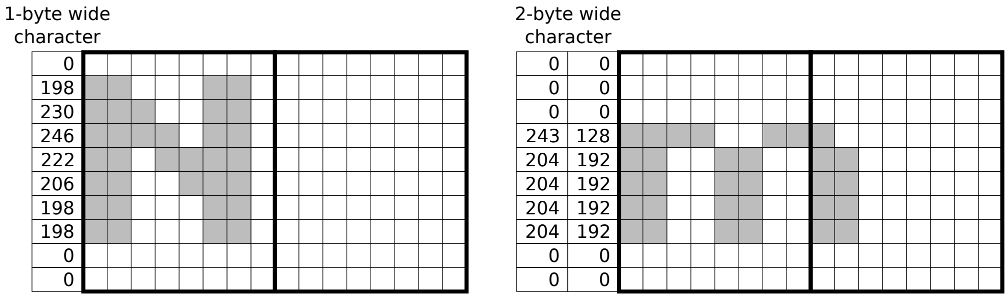


Figure 10: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

On the EGA videocard each 8 pixels take up 1 byte of VRAM (as explained in Section **??** on page **??**). When you need to print characters that aren't perfectly aligned with this 8-pixel grid, how do you manage the alignment? Let's say you want to display the letter 'N' on the screen. If the 'N' starts in the middle of a byte (say at pixel 3 instead of pixel 0), you need a clever way to shift the bits over to ensure it appears correctly on the screen.

Instead of manually shifting every pixel in your character bitmaps, the game engine uses pre-calculated bitshift tables. These tables are essentially lookup guides that help quickly adjust how characters are drawn based on their starting position. Here's how the process works:

1. Start with a base table (called `shiftdata0`) that holds all the possible values for a byte, from 0 to 255, and store this in an integer (16 bits).
2. Next, you generate seven more tables by shifting each value in `shiftdata0` one bit to the right for each table. So for `shiftdata1` the value 198 becomes 99, and for `shiftdata2` it becomes 32817, and so on. This process continues until you have eight tables, each representing a different bitshift.

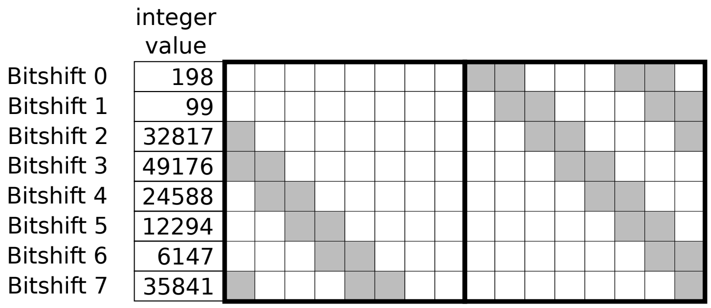


Figure 11: Right bitshift [0-7] for 198.

The pre-calculated bitshift table are stored in `id_vw_a.asm`, below the `bitshift3` table.

```
LABEL shiftdata3 WORD
  dw     0, 8192,16384,24576,32768,40960,49152,57344,     1, 8193,16385,24577,32769,40961
  dw 49153,57345,     2, 8194,16386,24578,32770,40962,49154,57346,     3, 8195,16387,24579
  dw 32771,40963,49155,57347,     4, 8196,16388,24580,32772,40964,49156,57348,     5, 8197
  dw 16389,24581,32773,40965,49157,57349,     6, 8198,16390,24582,32774,40966,49158,57350
  dw     7, 8199,16391,24583,32775,40967,49159,57351,     8, 8200,16392,24584,32776,40968
  dw 49160,57352,     9, 8201,16393,24585,32777,40969,49161,57353,    10, 8202,16394,24586
  dw 32778,40970,49162,57354,    11, 8203,16395,24587,32779,40971,49163,57355,    12, 8204
  dw 16396,24588,32780,40972,49164,57356,    13, 8205,16397,24589,32781,40973,49165,57357
  dw    14, 8206,16398,24590,32782,40974,49166,57358,    15, 8207,16399,24591,32783,40975
  dw 49167,57359,    16, 8208,16400,24592,32784,40976,49168,57360,    17, 8209,16401,24593
  dw 32785,40977,49169,57361,    18, 8210,16402,24594,32786,40978,49170,57362,    19, 8211
  dw 16403,24595,32787,40979,49171,57363,    20, 8212,16404,24596,32788,40980,49172,57364
  dw    21, 8213,16405,24597,32789,40981,49173,57365,    22, 8214,16406,24598,32790,40982
  dw 49174,57366,    23, 8215,16407,24599,32791,40983,49175,57367,    24, 8216,16408,24600
  dw 32792,40984,49176,57368,    25, 8217,16409,24601,32793,40985,49177,57369,    26, 8218
  dw 16410,24602,32794,40986,49178,57370,    27, 8219,16411,24603,32795,40987,49179,57371
  dw    28, 8220,16412,24604,32796,40988,49180,57372,    29, 8221,16413,24605,32797,40989
  dw 49181,57373,    30, 8222,16414,24606,32798,40990,49182,57374,    31, 8223,16415,24607
  dw 32799,40991,49183,57375
```

Printing the letter "N" with an offset of 3 pixels involves a simple lookup in `shiftdata3`, resulting in a 3-bit shifted "N" on the screen. By sacrificing a bit of memory, a lot of CPU time is saved.

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

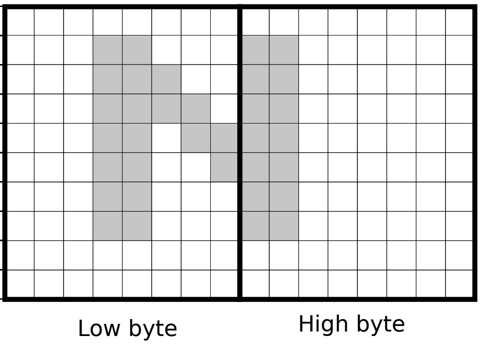


Figure 12: Bitshift ’N’ over 3 bits using bit shift tables.

```
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:
  SHIFTNOXOR
  add di,dx          ; next line in buffer
  loop  @@loop1
  ret
ENDP

; Macros to table shift a byte of font
MACRO SHIFTNOXOR
  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).

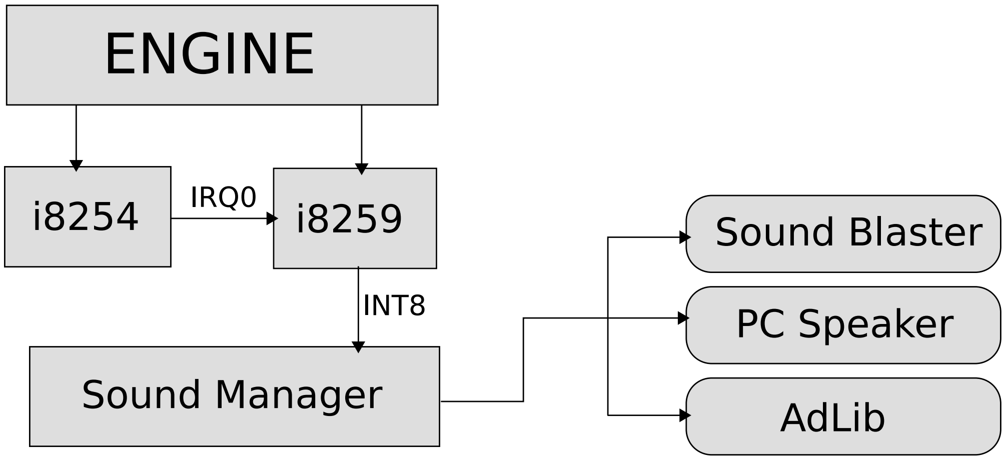


Figure 13: Sound system architecture.

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 primary function of the Softdisk files is to load and display the intro screen bitmap using the `LoadLIBShape` function from soft.c. Most of the functions in these files are not used and are therefore not discussed further in this book.

0.7 Startup

When the game engine starts, it first loads the Memory Manager. It then checks if at least 335KiB of RAM is available. If not, a warning is displayed, but the game can still continue. However, the game will likely crash or display an "Out of memory" error soon after.

Once the game has successfully started, the intro image, a Deluxe Paint bitmap image (`*.LBM`), is displayed. After the user presses any key, the intro image is unloaded from RAM to free up memory for runtime, and the control panel is displayed.

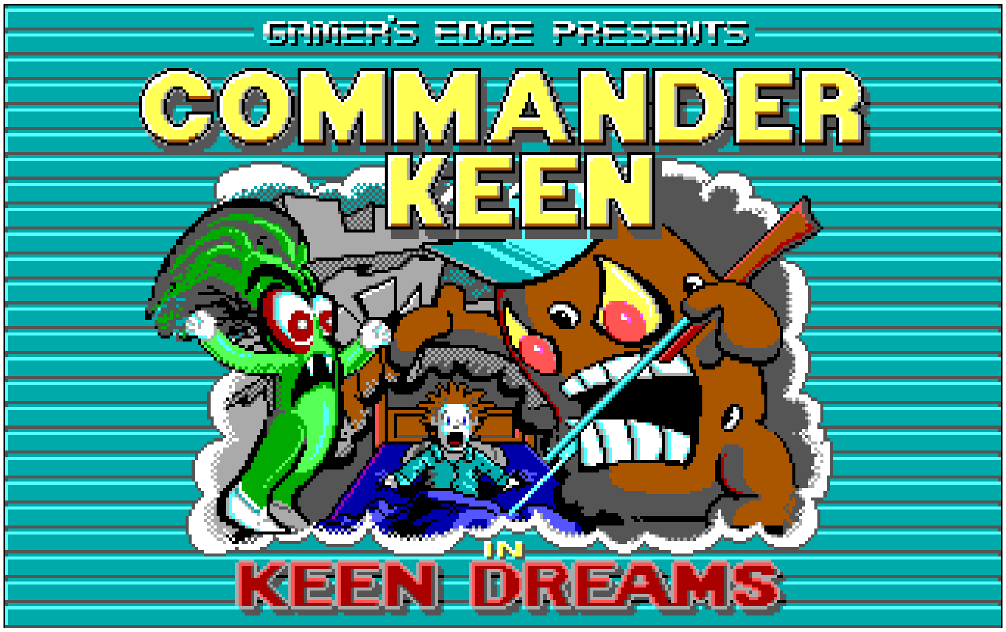


Figure 14: Keen Dreams intro screen