0.1 Asset Caching and Compression

The floppy disk is not only the slowest component of the PC but also constrained in terms of storage space. Therefore, it is crucial to load and store game assets as efficiently as possible in memory, avoiding long and unnecessary loading times from the disk. To make matters worse, the total amount of assets and maps cannot fit into RAM all at once. This is where a cache manager becomes essential. Its primary purpose is to increase data retrieval performance by reducing the need to load data from the slow floppy disk.

0.1.1 Asset caching

To manage and track the assets to be loaded into memory, the game engine uses a caching level mechanism. An array, sized according to the total number of graphical assets, is maintained to mark whether an asset needs to be loaded into memory.

```
byte    grneeded[NUMCHUNKS];
byte    ca_levelbit,ca_levelnum;
```

The index of this array corresponds to the graphic asset IDs. Using eight bits, the array can manage the required assets for different cache levels. The `ca_levelnum` variable points to the current cache level. Upon executing the engine, the cache manager starts with an empty array and `ca_levelnum` set to 1.

level bit:	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
STARTFONT								
CTL_STARTUPPIC								
CTL_HELPUPPIC								
...								
KEENSTANDRSPR								
KEENRUNR1SPR								
...								
SCOREBOXSPR								
...								
TILE8								
TILE8M								
TILE16 #1								
TILE16 #2								
...								

Figure 1: Initiating `grneeded[]` array, current cache level is 1.

When new resources for a level need to be cached in memory, all required assets are marked by setting the current level bit (bit 1).

```
#define CA_MarkGrChunk(chunk) grneeded[chunk]|=ca_levelbit

void InitGame (void)
{
  CA_ClearMarks ();  // Clears out all the marks at the
   current level

  // Mark assets to be cached in memory
  CA_MarkGrChunk(STARTFONT);
  CA_MarkGrChunk(STARTFONTM);
  CA_MarkGrChunk(STARTTILE8);
  CA_MarkGrChunk(STARTTILE8M);
  for (i=KEEN_LUMP_START;i<=KEEN_LUMP_END;i++)
    CA_MarkGrChunk(i);
}
```

The function `CA_CacheMarks` then iterates through the cache array, checking if any of the required assets are not yet available in memory. If not, it loads and decompresses the asset from disk into memory.

level bit:	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
STARTFONT								■
CTL_STARTUPPIC								
CTL_HELPUPPIC								
...								
KEENSTANDRSPR								■
KEENRUNR1SPR								■
...								
SCOREBOXSPR								■
...								
TILE8								■
TILE8M								■
TILE16 #1								■
TILE16 #2								
...								

Figure 2: Mark and load assets required for the new map in cache level 1.

Now, during gameplay, the user opens the control panel (e.g. to pause the game), which requires to load assets for the control panel into memory. The cache level is increased to level two and for all required control panel assets the second bit is marked.

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

  if (ca_levelnum==7)
    Quit ("CA_UpLevel: Up past level 7!");

  ca_levelbit<<=1;
  ca_levelnum++;
}
```

The `grneeded[]` cache array then appears as below.

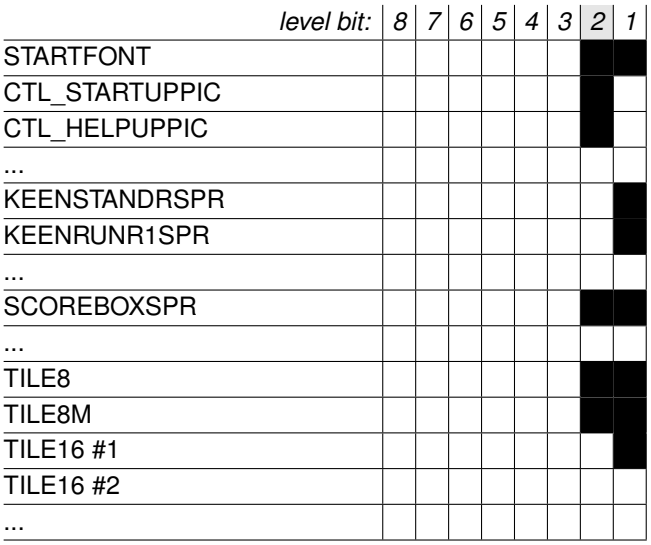


Figure 3: Mark all assets required for the control panel in cache level 2.

The function `CA_CacheMarks()` is called again to load all cache level 2 assets into memory. It iterates over all assets, loading any missing ones into memory. Any asset that is not required for cache level 2, but is already in memory will be marked for purging, meaning the memory manager can remove it from memory in case of insufficient memory.

```
void CA_CacheMarks (char *title, boolean cachedownlevel)
{
  numcache = 0;
    //
    // go through and make everything not needed purgable
    //
  for (i=0;i<NUMCHUNKS;i++)
    if (grneeded[i]&ca_levelbit)
    {
      if (grsegs[i])          // its allready in memory,
   make
        MM_SetPurge(&grsegs[i],0);  // sure it stays there!
      else
        numcache++;
    }
    else
    {
      if (grsegs[i])          // not needed, so make it
   purgeable
        MM_SetPurge(&grsegs[i],3);
    }

  if (!numcache)      // nothing to cache!
    return;
  ...
}
```

In this example, this applies to

- KEENSTANDRSPR
- KEENRUNR1SPR
- TILE16 #1

When the user closes the control panel, the engine lowers the cache level back to 1, where all assets required for playing the level are memorized. Simply calling the function CA_CacheMarks() again reloads any assets that were removed from memory.

```
void CA_DownLevel (void)
{
  if (!ca_levelnum)
    Quit ("CA_DownLevel: Down past level 0!");
  ca_levelbit>>=1;
  ca_levelnum--;
  //recaches everyting from the previous level
  CA_CacheMarks(titleptr[ca_levelnum], 1);
}
```

To avoid frequently used assets, such as fonts and Commander Keen sprites, being reloaded every time from the disk, they are loaded permanently into memory by flagging them as non-movable, unpurgeable blocks of memory.

```
MM_SetLock (&grsegs[STARTFONT],true);
MM_SetLock (&grsegs[STARTFONTM],true);
MM_SetLock (&grsegs[STARTTILE8],true);
MM_SetLock (&grsegs[STARTTILE8M],true);
for (i=KEEN_LUMP_START;i<=KEEN_LUMP_END;i++)
  MM_SetLock (&grsegs[i],true);
```

0.1.2 Asset compression

Given that the floppy disk is limited in both speed (100-250 kbps[1]) and storage capacity (3½-inch disk size is either 720KB or 1.44MB), file compression was essential. Compression ensures that the game occupies less space and loads more quickly. id Software used "Huffman compression" for all asset and map files, with additional "RLE compression" applied to further reduce the size of map files.

Huffman compression involves changing how various characters are stored. Normally, all characters in a given segment of data are equal and take an equal amount of space to store. However, by making more common characters take up less space while allowing less commonly used characters to take up more, the overall size of a segment of data can be reduced.

To illustrate the various aspects of Huffman compression, the following text, where each character is one byte, will be Huffman compressed:

```
Commander Keen in Keen Dreams
```

[1] Kilobit per second is a unit of data transfer rate equal to: 1,000 bits per second or 125 bytes per second.

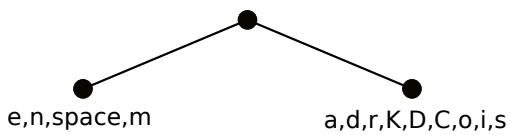
The first step is to make a character frequency table.

Character	Frequency
e	6
n	4
space	4
m	3
a	2
r	2
K	2

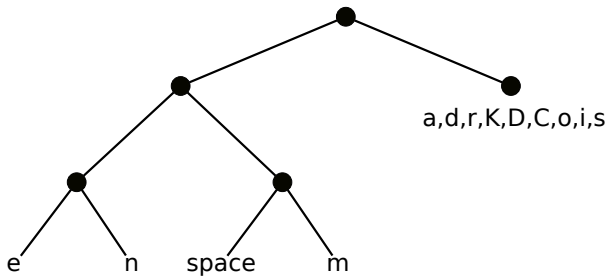
Character	Frequency
d	1
D	1
C	1
o	1
i	1
s	1

Figure 4: Character frequency table.

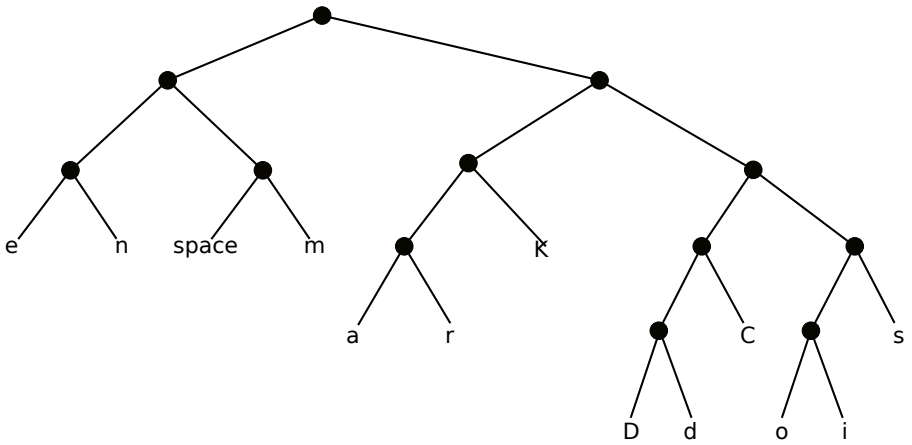
Next, an optimal binary tree, also known as a dictionary, is created. This is done by starting with the most common character and checking if it occurs more frequently than all the other characters combined. If not, then the second most common character is added, and so on. In our example, four characters make up more than half of the total number of characters: 'e', 'n', space, and 'm'. All of these characters are placed on the left side of the root node, while the remaining characters are placed on the right side.



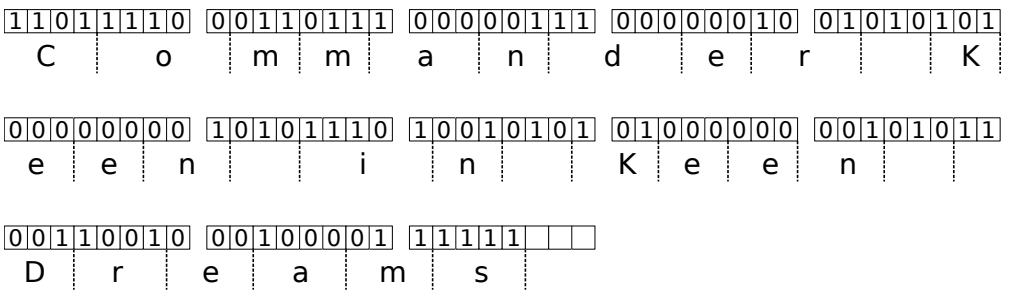
The process continues by creating a new node with left and right branches and repeating the steps. The two most common characters in the left node, 'e' and 'n', are assigned to the left branch of this new node. A third node is created, and since there are only two options, each is given a branch. This process is applied to the remaining characters of the left node, resulting in the structure shown below.



The same procedure is applied to the right node, where 'a', 'd', 'r', and 'K' constitute more than half of the total number of characters in the right node. After following the same steps, the final binary tree looks as follows:



Now, the entire text can be encoded by moving left (bit 0) or right (bit 1) through the tree, starting from the top node. For example, the character 'e' is `000`, 'm' is `011`, and 'o' is `11100`. The complete compressed message in bit form is now:



This results in a total of 13 bytes used for a 31-byte message, more than a 50% reduction. In Commander Keen, the dictionary is part of the game engine. Therefore, it is important that the asset files from the shareware version correspond with the dictionary files in the source code[2]. Reading the Huffman-compressed asset file is straightforward: you read bit by bit from the file and follow the dictionary, starting from the top node, until you reach an end node. The engine then writes the byte to memory and returns to the top node in the dictionary for the next bit in the file.

[2]That's why a specific source code version is mention in section "**??**" on page **??**.

```
typedef struct
{
  unsigned bit0,bit1; // 0-255 is a character,
                      // >255 is a pointer to a node
} huffnode;
```

The tile map asset files use a second compression technique, on top of Huffman. Upon closer inspection of a level, you'll notice large chunks of the same tile. For example, consider the first level you enter (Horse Radish Hill), which contains extensive areas of blue sky. Here, *Run-length encoding* (RLE) compression comes in useful.

The essence of this compression method is to compress data by saving the "run-length" of the encountered values, essentially storing the data as a collection of length/value pairs. To illustrate, the string 'aaaaaaaabbb' could be compressed to '8a3b' (8 bytes of 'a', 3 bytes of 'b'). The trick with RLE is to ensure that the data does not end up becoming larger after processing. For instance, using pure length/value pairs, 'abracadabra' would become '1a1b1r1a1c1a1d1a1b1r1a'.

In Commander Keen, this is solved by using a 'tag'. This special tag value instructs the program to take specific action upon encountering it. Every value is passed through unchanged until an RLE tag value is encountered. When this tag is read, instead of directly outputting it like other values, two further values are read. The first indicates the number of times to repeat, and the second represents the value to be repeated. The algorithm used in Commander Keen is based on 16-bits, and therefore is named RLEW, where the 'W' stands for word. The RLEW tag is defined as `ABCDh`.

The overall result of applying Huffman and RLEW compression results in almost 65% file size reduction.

Asset type	**filename**	**Uncompressed size**[3]	**Compressed size**
Graphic assets	KDREAMS.EGA	354,075 bytes	213,045 bytes
Game levels	KDREAMS.MAP	417,714 bytes	65,673 bytes
Sound assets	KDREAMS.AUD	4,572 bytes	3,498 bytes
Total		776,361 bytes	282,216 bytes

[3] See Appendix **??** for details per asset file.

```
void CA_RLEWexpand (unsigned huge *source, unsigned huge *
    dest,long length, unsigned rlewtag)
{
  unsigned value,count,i;
  unsigned huge *end;
  unsigned sourceseg,sourceoff,destseg,destoff,endseg,
    endoff;

  end = dest + (length)/2;    // length is COMPRESSED length
  sourceseg = FP_SEG(source);
  sourceoff = FP_OFF(source);
  destseg = FP_SEG(dest);
  destoff = FP_OFF(dest);
  endseg = FP_SEG(end);
  endoff = FP_OFF(end);

  asm mov bx,rlewtag          // RLEW tag: ABCDh
  asm mov si,sourceoff
  asm mov di,destoff
  asm mov es,destseg
  asm mov ds,sourceseg

  expand:
  asm lodsw
  asm cmp ax,bx               // value is RLEW tag?
  asm je  repeat
  asm stosw
  asm jmp next

  repeat:
  asm lodsw
  asm mov cx,ax               // repeat count
  asm lodsw                   // repeat value
  asm rep stosw

  next:                       // Next word value
  [...]
}
```