Multipaged intermission

by Ivan Mogilko, 15 November 2005

This tutorial on VavoomC programming describes how to make intermissions be multipaged – i.e. to show more than one screen of text. I am not sure that this feature is essential to everyone, but it can be good example of what you can do by modifying progs source.

I should underline, that this tutorial is based on Hexen mod, but I believe that you can do it in Doom and Heretic mods too, however some of the functions and variables mentioned here may have slightly different names.

In this tutorial following file will be modified:

- progs\<your mod name>\client\imission.vc

Task overview

First of all we shall state what we are going to do exactly. As you may know, intermission is a sort of a 'movie', shown at the end of game episodes. It can be found in most doom engine based games such as Doom, Heretic or Hexen. I call it 'movie', however it looks quite simple in original games: just a running text with some graphic at the background. And again, in original games there can be only one screen of text. When it is done printing, you press button and game moves on to the next episode (or to the main menu). And if you modify any intermission message of your game mod so it contain more than intermission screen can hold, and launch the game, then during intermission printed text will just go lower screen's bottom and you will not see its continuation at all. This limit is pretty annoying if you love writing long texts and want your game mod be able to show them. So, we will modify intermission source code and make it able to show a number of pages instead of one. At the end, this will work following way: a text will be printed down until a bottom of a screen, then game will wait for player to press button, then, if it has another one page of text, it will clear the screen and continue printing from the top.

Understanding 'imission.vc'

It is important to know how the original code works in detail, since this will save our time greatly when we begin to modify it.

In Vavoom progs intermission is implemented in 'imission.vc' source. Open it, and look at its code. Here is a brief explanation of the process.

- When the game is launching an intermission, it runs <**IM_Start()** function (it is in the very end of file). IM_Start itself calls **IM_InitStats()** function, that sets up all general parameters, such as intermission text, music played etc, then **IM_LoadPics()** function, that registers all the images need to be drawn during intermission.
- Intermission text is loaded into **ClusterMessage** array (of type *int*), then string variable **HubText** is given a reference to the beginning of that array (so, by using HubText variable you actually read ClusterMessage array contents, first character of HubText is the first byte of ClusterMessage etc). This is made using **ARR2STR** macro, as you may notice (its usage will be explained below).
- After that a loop starts (the loop itself, as far as I know, is implemented in Vavoom executable, so if you want to modify it you should edit Vavoom core source, written on C++, and recompile it). For every *tic* (a very short time period) this loop calls <**IM_Drawer()** function and then **IM_UpdateStats()** function.
- IM_UpdateStats function checks **interstate** flag which displays the state of intermission (running or ending), and **skipintermission** flag, which shows that it is time to end with intermission. Skipintermission flag is set to true by main loop (mentioned above), basically when player presses a button. If skipintermission is true, IM_UpdateStats function sets interstate flag to true, thus telling all other functions in 'imission.vc' that intermission is about to end.
- IM_Drawer function draws all necessary graphics, such as background picture. Then, if it is single player or cooperative game, it runs **IM_DrawHubText()** function, otherwise it runs <**IM_DrawDeathTally()** function. (Note: I won't touch IM_DrawDeathTally function here, because it only draws deathmatch score, so it do not need multipaging.)

- IM_DrawHubText function is setting up text parameters which it sends to *native* function **T_DrawNText**, which finally draws the text on screen. This seems simple, however, there is a thing that is ought to be understood right. There is **intertime** variable, which is set to zero at startup, and which is incremented each tic by IM_UpdateStats function. IM_DrawHubText uses this variable to know how much time has passed and, thus, how much text should be printed currently (as you may remember, during intermission text is printed symbol by symbol over time). I.e. for the first time IM_DrawHubText orders to draw one symbol, for the second time it draws two symbols (the first and the second ones), next it draws three – and so on. Each time it redraws all the text it has already written, adding one more symbol.

Step One: Page Breaking

Our immediate task is to write down a code, which will break intermission text into pages. As I found out it is unconvenient (or, perhaps, simply impossible) to use string variables to store intermission text partitions (this refers to Vavoom 1.18, may be situation will change in following versions). Thus we will need an integer type array, which will store indices of each page's ending symbols (i.e. show where each page ends).

But first we'll declare some useful macros. Find the source section commented as "MACROS" (in the beginning of the file), and add following to the end of the macros list:

```
#define MAX_INTRMSN_MESSAGE_PAGES 8
#define MAXLINESPERPAGE 20
```

MAX_INTRMSN_MESSAGE_PAGES macro defines maximum pages allowed, and MAXLINESPERPAGE defines maximum text lines each page can contain. Ofcourse you may change these (especially maximum pages macro), however I do not recommend you to make MAXLINESPERPAGE more than 20. I am not sure about Doom and Heretic, but Hexen can normally fit up to 20 lines of text (at a 320×200 virtual resolution, implemented in Vavom 1.18).

I also recommend you change the definition of MAX_INTRMSN_MESSAGE_SIZE macro. It is 1024 by default, that means that intermission text can have no more than 1024 characters. This is approximately two 320×200 screens full of text, so perhaps you will wish to change it. For the full eight pages 4096 will be quite enough.

Next is "DATA DECLARATIONS". Here we place our pagebreaks array and two other vars:

```
int PageBreaks[MAX_INTRMSN_MESSAGE_PAGES];
int page;
int lastpage;
```

Page variable stores zero-based index of current page, and **lastpage** variable will store an index of the last page in current intermission.

Now we create a new function named **PageBreak**. The function itself should be placed at the end of file, below all other functions (it is optional, but this will make it easier to find our new added functions).

NOTE: I added many comments to it, but they are not required to be copied, ofcourse.

```
void PageBreak(void)
{
   int line = 0; // this variable stores current line index int i; // it is just a dummy variable
   page = 0; // initializing page variable with zero
        /* here we start the main pagebreaking loop; as you may notice it counts from
        zero to the length of HubText string (that is  size of ClusterMessage array actually);
        in other words, this  loop looks through every symbol in intermission text. */
   for (i = 0; i < strlen(HubText); i++)
   {
      // if the null-terminator encountered...
```

```
        if (strgetchar(HubText, i) == 0)
        {
            // then write the index of the NEXT symbol into array and end the loop
            PageBreaks[page] = i + 1;
            break;
        }
            // if linebreak symbol encountered...
            if (strgetchar(HubText, i) == 0x0A)
        {
            // ...and if this line is the last one allowed in the page...
            if (line == MAXLINESPERPAGE - 1)
        {
             // then write the index of the NEXT symbol into array
             PageBreaks[page] = i + 1;
             // if it is the last page allowed - end the loop
             if (page == MAX_INTRMSN_MESSAGE_PAGES - 1)
               break;
             // else take following page, and reset line to zero
             page++;
             line = 0;
        }
            //  if it isn't the last line of the page, then just jump to the next line
            else line++;
        }
    }
          /* when the loop ends, we must set lastpage equal to our current page,
          on which we have stopped; then we should check whether we have a
         pagebreak for the last page, and if there isn't, write it down;
         NOTE: last thing will happen most probably every time, if not intermission
         text is larger than MAX_INTRMSN_MESSAGE_PAGES number of pages can contain. */
        lastpage = page;
        if (PageBreaks[lastpage] == 0)
        PageBreaks[lastpage] = i + 1;
}
```

Now, when we have this function, we should make IM_InitStats() call it at startup. Seek for IM_InitStatsfunction, find following lines and put our function call just after them:

```
HubText = ARR2STR(ClusterMessage);
HubCount = itof(strlen(HubText)) * TEXTSPEED + TEXTWAIT; PageBreak(); // <--- our function call
```

Good. Now at startup an array of page-ending indices will be written down. But it is only a half of job. Next, we are going to teach 'imission.vc' how to turn pages.

Step Two: Page Turning

At first we are going to add some new variables.

At "DATA DECLARATIONS" section add the following:

```
bool wantnextpage;
bool turnpage;
```

Wantnextpage flag will say to program that end of page is encountered, so it should delay printing until player presses a key. **Turnpage** flag, if set to true, will show that program is allowed to turn page already.

A bit later we will create our second function **PageTurn**. Before it we should do some modifications to existing functions.

In function IM_Start() find following lines and add code:

```
interstate = 0;
skipintermission = false;
intertime = 0.0;
// following the code that we add here
page = 0;
wantnextpage = false;
turnpage = false;
```

This code will initialize our new variables and reset page variable, since our text must start from the first (zero index) page.

In IM_UpdateStats() function find the following:

```
if (skipintermission || (!cl->deathmatch && !HubCount))
{
    interstate = 1;
    cnt = 0.3;
    skipintermission = false;
}
```

...and REPLACE it by this code:

```
if (skipintermission || (!cl->deathmatch && !HubCount))
{
    PageTurn();
    cnt = 0.3;
    skipintermission = false;
}
```

This will call our new function PageTurn() if player press any button, instead of just setting interstate flag to true.

In IM_Drawer() function find lines:

```
if (interstate)
{
    return;
}
```

...and add this code just after:

```
if (turnpage)
{
    wantnextpage = false;
    turnpage = false;
    intertime = 0.0;
}
```

This will ask program if it is a time to turn the page, and if yes, it will reset intertime variable. As you may guess, this will make IM_DrawHubText() function to start printing from the first character again (if you may not, see explanation of IM_DrawHubText function above).

Now, seek for IM_DrawHubText() function, find there lines:

```
T_SetFont(font_small);
```

```
T_SetAlign(hleft, vtop);
T_DrawNText(10, 5, HubText, count);
```

...and place a new code just BEFORE them:

```
if((page && count > PageBreaks[page] - PageBreaks[page - 1]) || count > PageBreaks[page])
{
    wantnextpage = true;
    if(page)
        count = PageBreaks[page] - PageBreaks[page - 1];
    else
        count = PageBreaks[page];
}
```

This code will make IM_DrawHubText function to see if there is an end of current page. This is made by comparing local **count** variable, which defines an exact number of symbols to print at this moment (it depends on intertime of course), and distance between two previously set pagebreak indices – from the current page and from the previous page. In other words, function checkes if current number of symbols, that it have to print, all belong to one page, or it's already over page's limit. In the last case it sets wantnextpage flag to true and makes sure that count is equal to total number of symbols on current page (for it should draw them repeatedly until player presses a button).

And finally (OH YES! 🙂) we should write down our function PageTurn():

```
void PageTurn(void)
{
    // if it is a deathmatch OR it is the last page of our text...
    if(deathmatch || page == lastpage)
    {
        // ...then set interstate flag to true, and so end the intermission
        interstate = 1;
        return;
    }
    // ...else switch to the next page
    HubText = ptrtos(&ClusterMessage[PageBreaks[page] / 4]);
    page++;
    turnpage = true;
}
```

I suppose that string with HubText needs more explanation. As it was already mentioned above, intermission text is actually stored in ClusterMessage array, and string variable HubText is given a reference to the beginning of that array. I also mentioned that it is made using ARR2STR macro. If you look into 'progs\common\builtins.vc' file you will find following definition:

```
// Pointer to string hack
native string ptrtos(void *ptr);
#define ARR2STR(array) ptrtos(&array[0])
```

What the hell is that? Well, that's not too hard to understand, really. We have function **ptrtos** that has a void pointer as a parameter and returns a string type. By using this function you may reference your string variable just any array of any type available, so that by reading your string you will actually read that array, i.e. bytes of that array will be converted to text characters. And ARR2STR (I believe it is an abbreviation for "Array To String", and "ptrtos" means "Pointer To String") macro is made for convenience. It returns a pointer to the first (zero) index of array.

So, back to our bizzare HubText line.

```
HubText = ptrtos(&ClusterMessage[PageBreaks[page] / 4]);
```

Here we give HubText string a reference not just to the beginning of ClusterMessage array, but to specific symbol, that is the first symbol of the next page (which we are switching to). Now IM_DrawHubText() function will print text starting not from the VERY first character, but from that page's first character.

As for the division on 4, it is needed because of the difference in types between HubText and ClusterMessage. HubText is a string and its every character fits in 1 byte. And ClusterMessage is an integer array, and its every index contains value written on 4 bytes. For those who is not aware of how computer memory is organised, I'll give a small table:

```
bytes || 0 || 1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 || 9 || 10 || 11 || 12 || 13 || 14 || 15 || 16
HubText indices || 0 || 1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 || 9 || 10 || 11 || 12 || 13 || 14 || 15 || 16
ClusterMessage indices || 0 || - || - || - || 1 || - || - || - || 2 || - || - || - || 3 || - || - || - || 4
```

So, as you can see, if we adress 3rd ClusterMessage's index (2), we will actually get 9th byte of text (8). If we adress 5th ClusterMessage's index (4), we will get 17th byte of text (16). Et cetera. Thus, if we need to adress any Xth character of the text through ClusterMessage array, we must instead take (X / 4) index.

Adding "Press any button" graphic

This feature is absolutely not required, so you may skip it. I just thought it would be great to add a small graphic, that will tell player it is an end of page already and that he should press any key to continue.

You may create your own image, as for me, I have chosen flickering arrow cursor from Hexen's main menu.

That is how it may be done.

First, declare following variables:

```
int patchNextPageArrow1;
int patchNextPageArrow2;
int patchNextPageArrowCur;
int nextpagearrowtics;
```

Two first variables will store IDs of two images that represent two animation frames, **patchNextPageArrowCur** var will define current animation frame and **nextpagearrowtics** will define tics remaining until frame change.

In IM_Start() function place a line somewhere near other variables' initialization:

```
nextpagearrowtics = 0;
```

In IM_LoadPics() function put images registration before or after existing code (it does not matter absolutely):

```
if(!deathmatch)
{
   patchNextPageArrow1 = R_RegisterPic("M_SLCTR1", PIC_PATCH);
   patchNextPageArrow2 = R_RegisterPic("M_SLCTR2", PIC_PATCH);
   patchNextPageArrowCur = patchNextPageArrow1;
}
```

This will register two additional pics named "M_SLCTR1" and "M_SLCTR2", and store their IDs in mentioned variables. As for patchNextPageArrowCur variable, it will store first frame ID for now.

In IM_Drawer() function there will be the main code for this “arrow” feature. You should place it at the very end of function, beneath all other code:

```
if (wantnextpage)
{
    picinfo_t arrow;
    if(nextpagearrowtics == 0)
        {
            if(patchNextPageArrowCur == patchNextPageArrow1)
                patchNextPageArrowCur = patchNextPageArrow2;
            else
                patchNextPageArrowCur = patchNextPageArrow1;
                nextpagearrowtics = 10;
        }
    R_GetPicInfo(patchNextPageArrowCur, &arrow);
    R_DrawPic(320 - arrow.width - 4, 200 - arrow.height - 8, patchNextPageArrowCur);
    nextpagearrowtics--;
}
```

First, program checks whether it is time to change frames and if it is, then it does so, setting our animation timer to 10 tics. Then it draws the image in the bottom-right corner of the screen. **Arrow** variable of type *picinfo_t* is used to know our image’s width and height, so that it would be positioned properly (picinfo_t is a rather simple structure, declared in ‘progs\common\types.vc’). And, of course, do not forget about timer decrement in the end.

The whole thing will work only while wantnextpage flag is set to true, all other time it will be skipped.

Results

Compile your game progs, modify any intermission message in your WAD so it have more than 20 lines of text, a-a-and run it!

I should also add, that you must be careful with line spacing when you write your lengthy intermission messages. Thus, if on one page there are all 20 lines of text, then do not make line space before next page’s text or it will be printed from the 2nd line instead of 1st. Use simple linebreak in this case.