GaumerieLib

GaumerieLib est une librairie écrite en C++ et en assembleur contenant des fonctions avancées de gestion des timers, ainsi que des fonctions rapides d'affichage de bitmaps en niveaux de gris.

1 - Installation

Cette section décrit comment faire fonctionner cette librairie dans un Add-In ClassPad.

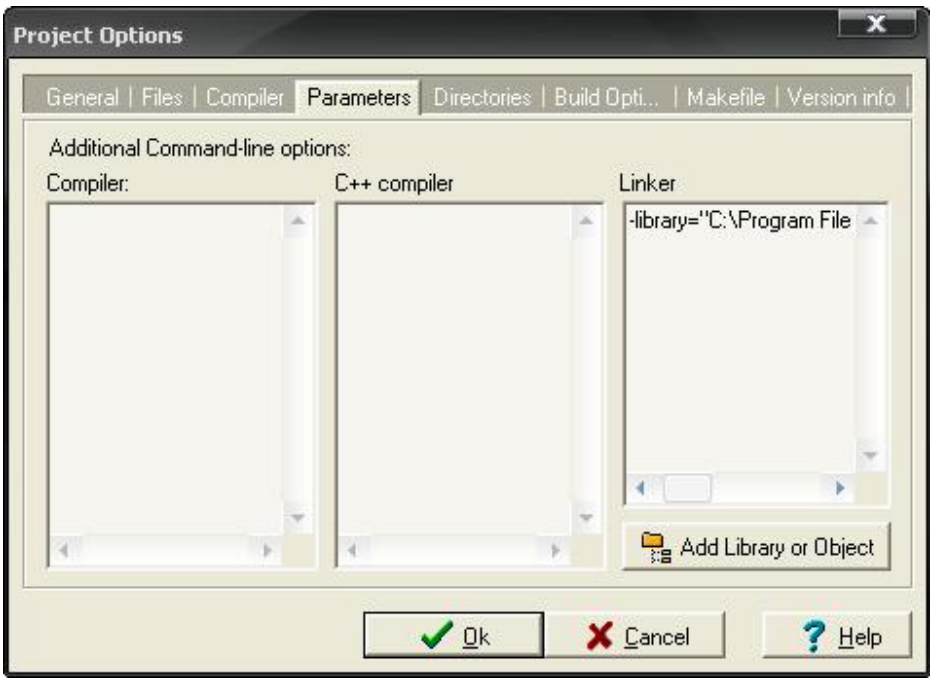
Intégrer la librairie

Déplacez tout d'abord tous les headers (.h) dans le dossier CP_Include du ClassPad 300 SDK, puis déplacez GaumerieLib.lib dans le dossier Lib.
Toutes les fonctions GaumerieLib sont contenues dans le fichier GaumerieLib.lib, il faut donc pour les utiliser, les lier à l'Add-In lors de la compilation. Pour faire un programme opérationnel utilisant cette librairie, vous devez ajouter un argument au linker indiquant qu'il doit lier GaumerieLib.lib au programme.
Pour configurer le linker sous Dev-C++, ouvrez votre projet, allez dans [Project] > [Project Options], sélectionnez l'onglet "Parameters", puis entrez dans "Linker" ceci :

```
library="C:\PROGRA~1\Casio\CLASSP~1\Lib\GaumerieLib.lib"
```

En admettant que vous avez installé le ClassPad 300 SDK dans le dossier `C:\Program Files\Casio\ClassPad 300 SDK\`.
Vous devez réduire les noms de dossier comportant des espaces à des noms courts.
Pour "Program Files", vous devez écrire les 6 premières lettres du nom en majuscules, puis le symbole "~", et généralement, le chiffre 1, ce qui donnera "PROGRA~1".
Par contre, s'il existe un autre dossier nommé "Program" dans le même dossier que "Program Files", "Program" aura pour nom court "PROGRA~1" et "Program Files" aura pour nom court "PROGRA~2", car "Program Files" vient après "Program" dans l'ordre alphabétique.

Si le dossier est différent, remplacez simplement `C:\PROGRA~1\Casio\CLASSP~1\` par votre dossier d'installation.



Créer un programme opérationnel

Tout d'abord, les fonctions de la librairie GaumerieLib ne sont pas supportés par le compilateur GCC, il est donc impossible d'exécuter ou de déboguer un Add-In sous Windows. Vous devez compiler uniquement un fichier .cpa à transférer sur le ClassPad. Pour inclure la librairie dans un programme, il suffit d'inclure le fichier GaumerieLib.h.

2 – Utilisation des différents timers

Cette section présente les différents types de timers utilisables par le biais des fonctions de GaumerieLib.

Timers à haute fréquence (SH3Timers)

Les SH3Timers sont des registres à 32 bits du processeur qui se décrémentent à une vitesse fixée. Une fois la valeur d'un timer arrivé à zéro, ce timer reprend une valeur fixée et se décrémente à nouveau, et indique son passage à zéro dans un autre registre dont on peut lire la valeur et l'effacer. Il y a en tout 3 SH3Timers, identifié par les valeurs `SH3Timer::TIMER_0`, `SH3Timer::TIMER_1` et `SH3Timer::TIMER_2`.

Contrôle des SH3Timers

Pour initialiser un SH3Timer, il faut tout d'abord régler sa vitesse de décrémentation, et la valeur qu'il prend après chaque passage à zéro. Pour cela, il faut utiliser ces deux fonctions :

```
void SH3Timer::SetClockFreq(TIMER timer, FREQUENCE freq);
// Règle la fréquence d'horloge d'un timer
// timer doit être soit SH3Timer::TIMER_0 soit SH3Timer::TIMER_1 soit SH3Timer::TIMER_2
// freq doit être soit SH3Timer::FRQ_3MHZ soit SH3Timer::FRQ_920KHZ soit SH3Timer::FRQ_230KHZ
// soit SH3Timer::FRQ_57KHZ
// SH3Timer::FRQ_3MHZ = fréquence de 3 682 419 Hz
// SH3Timer::FRQ_920KHZ = fréquence de 920 385 Hz
```

```
// SH3Timer::FRQ_230KHZ = fréquence de 230 147 Hz
// SH3Timer::FRQ_57KHZ = fréquence de 57 538 Hz

void SH3Timer::SetTimerFreq(TIMER timer, unsigned int freq)
// Règle la valeur initiale que prend le timer après un passage à zero
// timer doit être soit SH3Timer::TIMER_0 soit SH3Timer::TIMER_1 soit SH3Timer::TIMER_2
// freq est un entier positif

// Exemple:A une fréquence de 57 538 Hz, le timer se décrémente 57 538 fois toutes les secondes.
// Avec une valeur initiale de 57 538, le timer passera à zéro toutes les secondes
// Avec une valeur initiale de 141 076 = 57 538 * 2, le timer passera à zéro toutes les deux
// secondes.

// Pour régler le timer 0 à 57 538 Hz avec une valeur initiale de 141 076, ils suffit d'écrire
// SH3Timer::SetClockFreq(SH3Timer::TIMER_0, SH3Timer::FRQ_57KHZ);
// SH3Timer::SetTimerFreq(SH3Timer::TIMER_0, 141 076);

// Pour obtenir la fréquence ou la valeur initiale d'un timer, ils suffit d'utiliser ces
// fonctions:
FREQUENCE SH3Timer::GetClockFreq(TIMER timer);
unsigned int SH3Timer::GetTimerFreq(TIMER timer);
```

Il est aussi possible de régler et d'obtenir la valeur actuelle du timer à n'importe quel moment :

```
void SH3Timer::SetCount(TIMER timer, unsigned int count);
// Règle la valeur actuelle d'un timer
// count doit être un entier positif

unsigned int SH3Timer::GetCount(TIMER timer)
// Retourne la valeur actuelle d'un timer
```

Pour démarrer et arrêter les timers, quatre fonctions sont à disposition:

```
void SH3Timer::Start(TIMER timer);
// Démarre un timer

void  SH3Timer::Stop(TIMER timer)
// Arrête un timer

void SH3Timer::StartAll();
// Démarre tous les timers

void  SH3Timer::StopAll()
// Arrête tous les timers
```

Danger ! N'oubliez jamais d'arrêter tous les timers à la fin du programme. Quand le programme se termine, si les timers ne sont pas arrêtés, ils continueront à se décrémenter en boucle, même lorsque le ClassPad est éteint.

Pour savoir si un timer est déjà passé à zéro et effacer l'indicateur de passage à zéro, il suffit d'appeler cette fonctionà n'importe quel moment dans le programme:

```
bool SH3Timer::HasUnderflowed(TIMER timer);
// Retourne true si le timer est passé à zéro, et false dans le cas contraire
// Permet d'effectuer des opérations à intervalles réguliers durant une boucle
```

Note : SH3Timer n'est pas une classe, c'est simplement un espace de nommage.

Horloge en temps réel (RTCClock)

L'horloge en temps réel est une horloge interne qui fonctionne en permanence et qui s'incrémente toutes les secondes, permettant ainsi d'obtenir la date et l'heure.

Paramétrer l'horloge

Pour paramétrer la date et l'heure, les fonctions de l'horloge en temps réel utilisent une structure de données DateTime définie comme ceci:

```
struct DateTime
{
    unsigned char uc_Seconde;   //  Secondes
    unsigned char uc_Minute;    //  Minutes
    unsigned char uc_Hour;      //  Heures
    unsigned char uc_WeekDay;   //  Jour de la semaine
                                //  Les valeurs possibles sont:
                                //  RTC_WD_SUNDAY
                                //  RTC_WD_MONDAY
                                //  RTC_WD_TUESDAY
                                //  RTC_WD_WEDNESDAY
                                //  RTC_WD_THURSDAY
                                //  RTC_WD_FRIDAY
                                //  RTC_WD_SATURDAY
    unsigned char uc_Day;       //  Jour
    unsigned char uc_Month;     //  Mois
    unsigned char uc_Year;      //  2 derniers chiffres de l'année
    unsigned char uc_MaskEnb;   //  Masque des données à prendre en compte
        /*  bit 0 : uc_Seconde          bit 4 : uc_Day
            bit 1 : uc_Minutes          bit 5 : uc_Month
            bit 2 : uc_Hour             bit 6 : uc_Year
            bit 3 : uc_WeekDay

            0 : pas pris en compte      1 : pris en compte  */
        // Une valeur de 0xFF (11111111 en binaire) prend tout en compte
        // Il est possible de définir simplement ce qu'on veut prendre en compte avec ceci:
        // DT_ENB_SECONDE = 0x01 = 00000001
        // DT_ENB_MINUTE  = 0x02 = 00000010
        // DT_ENB_HOUR    = 0x04 = 00000100
        // DT_ENB_WEEKDAY = 0x08 = 00001000
        // DT_ENB_DAY     = 0x10 = 00010000
        // DT_ENB_MONTH   = 0x20 = 00100000
        // DT_ENB_YEAR    = 0x40 = 01000000
        // Ainsi DT_ENB_SECONDE|DT_ENB_MINUTE|DT_ENB_HOUR prend en compte uniquement les
        // secondes, les heures et les minutes


    //  vérifie si les données sont valides et cohérents
    bool IsValid()
    {
        return !((uc_MaskEnb&0x01 && (uc_Seconde>0x59     ||  (uc_Seconde&0x0F)>9    ))|
                 (uc_MaskEnb&0x02 && (uc_Minute>0x59      ||  (uc_Minute&0x0F)>9     ))|
                 (uc_MaskEnb&0x04 && (uc_Hour>0x23        ||  (uc_Hour&0x0F)>9       ))|
                 (uc_MaskEnb&0x08 && (uc_WeekDay>6                                   ))|
                 (uc_MaskEnb&0x10 && (uc_Day>0x31         ||  (uc_Day&0x0F)>9        ))|
                 (uc_MaskEnb&0x20 && (uc_Month>0x12       ||  (uc_Month&0x0F)>9      ))|
                 (uc_MaskEnb&0x40 && (uc_Year>0x99        ||  (uc_Year&0x0F)>9       )));
    }
};
```

Pour créer un objet DateTime contenant l'heure et la date 10:35:20 Vendredi 17 novembre 2006, et prenant tout en compte, il suffit d'écrire :

```
DateTime dt = {20, 35, 10, RTC_WD_FRIDAY, 17, 11, 06, 0xFF};
```

Un objet DateTime vide se définit comme ceci:

```
DateTime dt;
```

Pour régler l'horloge à partir d'un tel objet, ou obtenir l'heure, il faut utiliser les fonctions suivantes:

```
void RTCClock::SetDateTime(const DateTime& dt);
// Règle l'horloge à partir de l'objet DateTime passé en argument

void RTCClock::AdjustSec();
// Arrondit les secondes à la minute inférieure (<=30 sec) ou à la minute supérieure (>30 sec)

void RTCClock::GetDateTime(DateTime& dt);
// Stocke les données de date et d'heure dans l'objet DateTime passé en argument
```

Enfin, pour modifier l'état de l'horloge (marche, arrêt), il suffit d'utiliser ces fonctions:

```
void RTCClock::Run(bool state);
// Démarre (state=true) ou arrête (state=false) l'horloge

bool RTCClock::IsRunning();
// Retourne true si l'horloge est en marche, et false si elle est à l'arrêt
```

Note : RTCClock n'est pas une classe, c'est simplement un espace de nommage.

Alarme (RTCAlarm)

L'alarme est basée sur l'horloge en temps réel, mais elle n'a pour l'instant aucune utilité car il n'est pas encore possible de faire sonner le buzzer interne du ClassPad en fonction de l'état de l'alarme. De plus, seuls les premiers modèles possèdent ce buzzer.

3 – Test de touches rapide

Cette courte section présente une fonction permettant d'effectuer un test de touches à n'importe quel moment dans le programme. De plus, elle permet de détecter l'appui sur plusieurs touches en même temps, ce qui n'est pas possible avec les PegMessages.

Utilisation de GetKey

L'espace de nommage GetKey porte mal son nom, puisque la fonction `GetKey::GetKey` ne retourne pas le code de la touche actuellement enfoncée, mais retourne l'état d'un touche (enfoncée, relâchée).

Les codes de touches sont définis par:

```
GetKey::K_ONOFF    = 0x10  // ON/Off
GetKey::K_EXE      = 0x21  // EXE
GetKey::K_EXP      = 0x22  // EXP
GetKey::K_DOT      = 0x24  // .
GetKey::K_0        = 0x25  // 0
GetKey::K_OPPOSIT  = 0x26  // (-)
GetKey::K_PLUS     = 0x31  // +
GetKey::K_3        = 0x32  // 3
GetKey::K_2        = 0x34  // 2
GetKey::K_1        = 0x35  // 1
GetKey::K_MINUS    = 0x41  // -
GetKey::K_COMMA    = 0x36  // ,
GetKey::K_6        = 0x42  // 6
GetKey::K_5        = 0x44  // 5
GetKey::K_4        = 0x45  // 4
GetKey::K_RPAR     = 0x46  // )
GetKey::K_TIMES    = 0x51  // * (multiplication)
GetKey::K_9        = 0x52  // 9
GetKey::K_Z        = 0x53  // z
GetKey::K_8        = 0x54  // 8
GetKey::K_7        = 0x55  // 7
GetKey::K_LPAR     = 0x56  // (
GetKey::K_DIV      = 0x61  // / (division)
GetKey::K_POWER    = 0x62  // ^
GetKey::K_LEFT     = 0x63  // Gauche
GetKey::K_DOWN     = 0x64  // Bas
```

```
GetKey::K_Y        = 0x65  // y
GetKey::K_X        = 0x66  // x
GetKey::K_BACK     = 0x71  // <-
GetKey::K_CLEAR    = 0x72  // Clear
GetKey::K_RIGHT    = 0x73  // Droite
GetKey::K_UP       = 0x74  // Haut
GetKey::K_KEYBOARD = 0x75  // Keyboard
GetKey::K_EQUAL    = 0x76  // =
```

Voici la syntaxe de la fonction `GetKey::GetKey` :

```
bool GetKey::GetKey(KEY_CODE code);
// Retourne true si la touche est enfoncée, et false si elle est relâchée
// code doit être un des codes écrits plus haut
// Attention, GetKey::GetKey(0x71) n'est pas accepté
// Il faut écrire GetKey::GetKey(GetKey::K_BACK)
```

4 – Fonctions d'affichage

Cette section est la plus longue, elle présente les fonctions les plus intéressantes de GaumerieLib : les fonctions d'affichage de bitmaps en niveaux de gris.

Objets bitmaps

Il existe quatre types de bitmaps dans GaumerieLib : `GBitmap`, `GBitmap2`, `GImage` et `GImage2`.
Leurs définitions sont similaires : deux entiers représentant la taille en pixels du bitmap, et un certain nombre de buffers (listes d'octets).

`GBitmap` est le bitmap monochrome de base à un buffer. Pour convertir une image Windows en GBitmap, il suffit d'utiliser l'utilitaire de conversion du ClassPad SDK pour générer un PegBitmap, puis il faut prendre uniquement la liste de type `ROMDATA UCHAR[]`, remplacer ce type par `unsigned char[]`, puis créer un GBitmap à partir de ce buffer et de la taille du bitmap. Par exemple :

```
// Code obtenu à partir d'un bitmap Windows avec le convertisseur de ClassPad SDK

ROMDATA UCHAR uctest[8] = {
0x3c,0x7a,0xf1,0xfb,0xff,0xff,0x7e,0x3c,};
PegBitmap test = { 0x00, 1, 8, 8, 0x000000ff, (UCHAR *) uctest};

// Code adapté à un GBitmap
// Pour créer un GBitmap, on peut utiliser l'expression
// GBitmap bitmap = {largeur, hauteur, buffer};

unsigned char uctest[8] = {
0x3c,0x7a,0xf1,0xfb,0xff,0xff,0x7e,0x3c,};
GBitmap test = {8, 8, uctest};
```

`GBitmap2` est un bitmap à un niveau de gris, il comporte deux buffers. La règle de ces buffers est :
Noir + Noir = Noir
Noir + Blanc = Gris
Blanc + Noir = Gris
Blanc + Blanc = Blanc

Voici un exemple simple :

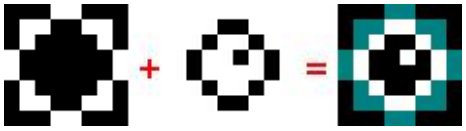
Pour créer un tel bitmap, il faut obtenir les deux buffers, toujours par le biais du convertisseur. Voici un exemple :

```
// Pour créer un GBitmap2, on peut utiliser l'expression
// GBitmap2 bitmap = {largeur, hauteur, buffer1, buffer2};

unsigned char uctest1[8] = {
0x00, 0x70, 0xe0, 0xe0, 0xf2, 0xfe, 0x7e, 0x3c,};
unsigned char uctest2[8] = {
0x3c, 0x7a, 0xf1, 0xfb, 0xff, 0xff, 0x7e, 0x3c,};
GBitmap2 test = {8, 8, uctest1, uctest2};
```

GImage est un bitmap monochrome supportant la transparence, il comporte deux buffers. La règle de ces buffers est :
Noir + Noir = Blanc
Noir + Blanc = Noir
Blanc + Noir = Inversé
Blanc + Blanc = Transparent

Voici un exemple simple (la couleur bleu-vert représente la transparence) :



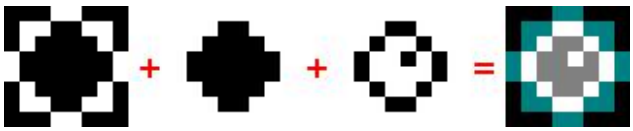
Le principe pour créer un tel bitmap est le même que pour créer un GBitmap2 :

```
// Pour créer un GImage, on peut utiliser l'expression
// GImage image = {largeur, hauteur, buffer1, buffer2};

unsigned char uctest1[8] = {
0x00, 0x70, 0xe0, 0xe0, 0xf2, 0xfe, 0x7e, 0x3c,};
unsigned char uctest2[8] = {
0x3c, 0x7a, 0xf1, 0xfb, 0xff, 0xff, 0x7e, 0x3c,};
GImage test = {8, 8, uctest1, uctest2};
```

GImage2 est un bitmap à un niveau de gris supportant la transparence, il comporte 3 buffers. La règle de ces buffers est :
Blanc + Blanc + Blanc = Transparent
Blanc + Noir + Noir = Inversé
Noir + Blanc + Blanc = Noir
Noir + Noir + Blanc = Gris
Noir + Blanc + Noir = Gris
Noir + Noir + Noir = Blanc

Voici un exemple simple (la couleur bleu-vert représente la transparence) :



Pour créer un GImage2 il suffit d'utiliser l'expression

```
GImage2 image = {largeur, hauteur, buffer1, buffer2, buffer3};
```

Dessiner sur l'écran en monochrome

La taille de l'écran est de 160 pixels en largeur, et 240 pixels en hauteur.
Il faut savoir que l'écran n'est autre qu'un buffer, c'est à dire une image sur laquelle on dessine. GaumerieLib fournit un type élémentaire BITMAP qui n'est en fait qu'un octet non signé unsigned char. Une liste d'octets (buffer) peut représenter une image

en noir et blanc, en effet, la représentation binaire de chaque octet est une ligne de 8 pixels sur l'écran (0 = pixel éteint, 1 = pixel allumé). Ainsi, l'écran comportant 160x240 = 38400, le buffer le représentant comportera 38400/8 = 4800 octets.
Voici un exemple simple pour comprendre :

Représentation d'un buffer en hexadécimal :
```
0xFF,0xFF,0xC0,0x03,0x3F,0xFC
```
Représentation de ce buffer en binaire :
```
11111111,11111111,
11000000,00000011,
00111111,11111100
```
Résultat pour une image de 16x3 :

Résultat pour une image de 14x2 : On observe que les pixels qui "débordent" sont simplement ignorés

Il existe deux moyens d'obtenir une référence à l'écran.

Tout d'abord, si vous utilisez toujours la librairie PEG en association avec la librairie GaumerieLib, vous pouvez obtenir la référence au buffer temporaire de l'écran. Il s'agit de l'image sur laquelle vous dessinez après un `BeginDraw()`, et qui est affichée sur l'écran après `EndDraw()`.

```
BITMAP* screen = (BITMAP*)VRAM_ADRS;
```

Si vous ne voulez pas utiliser PEG, vous pouvez créer une image vierge de la taille de l'écran :

```
BITMAP buffer[4800] = {0x00};
```

Il suffit d'effectuer vos opérations sur cette image, puis d'appeler la fonction `CopyToScreen()` avec pour paramètre cette image pour la copier directement sur l'écran. Entre autres, GaumerieLib met à disposition la fonction `CopyFromScreen()` qui permet d'effectuer une capture d'écran.

```
BITMAP buffer[4800] = {0x00};
CopyFromScreen(buffer); // Capture l'écran dans buffer

// On effectue quelques opérations sur "buffer"
// ...

CopyToScreen(buffer);   // Puis on affiche le tout à nouveau sur l'écran
```

Mais quelles sortes d'opérations peut on effectuer sur une image ?
Pour l'instant, GaumerieLib permet uniquement d'afficher des images, il n'y a pas de fonction pour tracer des lignes, des cercles, des rectangles, ou du texte.
Il existe plusieurs fonctions similaires pour afficher des images :

```
// Affichage de bitmap monochrome sans transparence
// tuc_Src : Bitmap à afficher (bitmap source)
// i_SrcWidth, i_SrcHeight : Largeur et hauteur du bitmap à afficher
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// tuc_Dest : Bitmap sur lequel afficher (bitmap de destination)
// i_DstWidth, i_DstHeight : Largeur et hauteur du bitmap de destination
// uc_type : Type d'affichage : DRAWBITMAP_COPY > Affiche le bitmap tel quel
//                              DRAWBITMAP_OR > Affiche uniquement les pixels noirs
//                              DRAWBITMAP_XOR > Inverse les pixels correspondant aux pixels
//                                               noirs du bitmap
//                              DRAWBITMAP_NOT > Force au blanc les pixels correspondant au
//                                               pixels noirs du bitmap
// Par défaut :  DRAWBITMAP_COPY
inline void DrawBitmap(const BITMAP* tuc_Src, int i_SrcWidth, int i_SrcHeight,
    int i_x, int i_y,
    BITMAP* tuc_Dest, int i_DstWidth, int i_DstHeight,
    unsigned char uc_type = DRAWBITMAP_COPY);
```

```
// Affichage de bitmap monochrome avec transparence
// tuc_Src1 : Premier buffer du bitmap à afficher (bitmap source)
// tuc_Src2 : Second buffer du bitmap à afficher (bitmap source)
//   Les deux buffers doivent avoir la même taille
// i_SrcWidth, i_SrcHeight : Largeur et hauteur du bitmap à afficher
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// tuc_Dest : Bitmap sur lequel afficher (bitmap de destination)
// i_DstWidth, i_DstHeight : Largeur et hauteur du bitmap de destination
// uc_type : Type d'affichage : DRAWIMAGE_ORXOR > Affiche l'image par la méthode OR - XOR
//                              DRAWIMAGE_NOTOR > Affiche l'image par la méthode AND - OR
// Par défaut : DRAWIMAGE_ORXOR
inline void DrawImage(const unsigned char* tuc_Src1, const unsigned char* tuc_Src2,
    int i_SrcWidth, int i_SrcHeight,
    int i_x, int i_y,
    unsigned char* tuc_Dest, int i_DstWidth, int i_DstHeight,
    unsigned char uc_type = DRAWIMAGE_ORXOR);
```

Cette écriture peut s'avérer parfois longue et peu lisible, c'est là qu'interviennent les classes `GBitmap` et `GImage`. GaumerieLib fournit une deuxième version de ces fonctions permettant de raccourcir un peu la liste des paramètres.

```
// Affichage de bitmap monochrome sans transparence (version avec GBitmap)
// pGBmp_Src : Bitmap à afficher (bitmap source)
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// pGBmp_Dest : Bitmap sur lequel afficher (bitmap de destination)
// uc_type : Type d'affichage
inline void DrawBitmap(const GBitmap* pGBmp_Src,
    int i_x, int i_y,
    GBitmap* pGBmp_Dest,
    unsigned char uc_type = DRAWBITMAP_COPY);
```

```
// Affichage de bitmap monochrome avec transparence (version avec plusieurs GBitmaps)
// pGBmp_Src1 : Premier buffer du bitmap à afficher (bitmap source)
// pGBmp_Src2 : Second buffer du bitmap à afficher (bitmap source)
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// pGBmp_Dest : Bitmap sur lequel afficher (bitmap de destination)
// uc_type : Type d'affichage
inline void DrawImage(const GBitmap* pGBmp_Src1, const GBitmap* pGBmp_Src2,
    int i_x, int i_y,
    const GBitmap* pGBmp_Dest,
    unsigned char uc_type = DRAWIMAGE_ORXOR);

// Affichage de bitmap monochrome avec transparence (version avec GImage)
// pImg_Src : Bitmap avec transparence à afficher (bitmap source)
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// tuc_Dest : Bitmap sur lequel afficher (bitmap de destination)
// i_DstWidth, i_DstHeight : Largeur et hauteur du bitmap de destination
// uc_type : Type d'affichage
inline void DrawImage(const GImage* pImg_Src,
    int i_x, int i_y,
    BITMAP* tuc_Dest, int i_DstWidth, int i_DstHeight,
    unsigned char uc_type = DRAWIMAGE_ORXOR);
```

On observe que la largeur et la hauteur des bitmaps ne sont plus nécessaires, vu que celles ci sont intégrées dans les classes `GBitmap` et `GImage`. Il est un peu dommage que la version avec `GImage` de `DrawImage()` ne prenne pas pour argument d'objet `GBitmap` pour le bitmap de destination, mais elle sera sans doute ajoutée dans une prochaine version de la librairie.

Dessiner sur l'écran en niveaux de gris

Pour l'instant, GaumerieLib ne supporte qu'un seul niveau de gris.

Le principe pour obtenir du gris sur un écran monochrome est simple, il faut simplement faire clignoter les pixels très rapidement. Pour cela, le "moteur" de niveaux de gris de GaumerieLib utilise le SH3Timer 0, dans ce cas, il ne faudra plus toucher à ce timer si vous utilisez les niveaux de gris dans un programme. L'écran est à présent double, chaque buffer étant affiché alternativement. Chacun de ces buffers a donc une dimension de 160x240, comme l'écran normal.

Tout d'abord, au démarrage du programme, il faut initialiser le "moteur" de niveaux de gris avec la fonction `GrayScales::Initialize()`, pour cela, on doit créer deux bitmaps de la taille de l'écran. Il existe deux moyens :

```
////////////////////////////////////////////////
// Méthode 1 : on crée deux buffers indépendants

BITMAP screen1[4800] = {0x00};
BITMAP screen2[4800] = {0x00};

GrayScales::Initialize(); // Initialise le timer des niveaux de gris
GrayScales::SetBuffers(screen1, screen2); // Rattache les deux buffers à l'écran
GrayScales::Start(); // Démarre le moteur de niveaux de gris

////////////////////////////////////////////////
// Méthode 2 : on crée un GBitmap2

BITMAP screen1[4800] = {0x00};
BITMAP screen2[4800] = {0x00};
GBitmap2 screen = {160, 240, screen1, screen2};

GrayScales::Initialize();
// tuc_Bmp et tuc_Bmp2 sont les deux buffers constituant un GBitmap
GrayScales::SetBuffers(screen.tuc_Bmp, screen.tuc_Bmp2);
GrayScales::Start();
```

La méthode avec `GBitmap2` est plus pratique car elle permet ensuite d'utiliser les fonctions de dessin en niveaux de gris sur un `GBitmap2`, ce qui est plus simple que sur deux buffers isolés.
Toute action effectué sur les buffers rattachés à l'écran sera immédiatement affichée, aucune fonction n'est à appeler.
De même que pour le dessin monochrome, le dessin en niveaux de gris ne permet que l'affichage de bitmaps sur l'écran. Les fonctions sont similaires à celles utilisées en dessin monochrome.

```
// Affichage de bitmap à niveaux de gris sans transparence
// pGBmp2_Src : Bitmap à afficher (bitmap source)
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// pGBmp2_Dest : Bitmap sur lequel afficher (bitmap de destination)
// uc_type : Type d'affichage (DRAWBITMAP_COPY / OR / XOR / NOT)
inline void DrawBitmap2(const GBitmap2* pGBmp2_Src,
       int i_x, int i_y,
       GBitmap2* pGBmp2_Dest,
       unsigned char uc_type = DRAWBITMAP_COPY);

// Affichage de bitmap à niveaux de gris avec transparence
// pImg2_Src : Bitmap avec transparence à afficher (bitmap source)
// i_x, i_y : Coordonnées de l'endroit où on veut afficher le bitmap
// tuc_Dest1, tuc_Dest2 : Les deux buffers du bitmap sur lequel afficher (bitmap de destination)
// i_DstWidth, i_DstHeight : Largeur et hauteur du bitmap de destination
// uc_type : Type d'affichage (DRAWBITMAP_COPY / OR / XOR / NOT)
inline void DrawImage2(const GImage2* pImg2_Src,
    int i_x, int i_y,
    unsigned char* tuc_Dest1, unsigned char* tuc_Dest2, int i_DstWidth, int i_DstHeight,
    unsigned char uc_type = DRAWIMAGE_ORXOR);
```

Pour arrêter le moteur de niveaux de gris, il suffit d'appeler les fonctions `Stop()` puis `Restore()`.

```
//Restore l'état du timer utilisé pour les niveaux de gris
void Restore();

// Arrête le timer
void Stop();
```

Avant de quitter le programme, il faut toujours arrêter le moteur de niveaux de gris, dans le cas contraire, celui-ci se comporterait comme un timer non arrêté, il fonctionnerait en permanence, même lorsque le ClassPad est éteint.

5 – Modèle de programme pour les jeux

Cette section fournit simplement un exemple de modèle de programme réutilisable facilement pour créer un jeu en niveaux de gris.

Code source

Voici le code source de chaque fichier à intégrer.

bitmaps.h

```
BITMAP scrN[4800]  = {0x00};
BITMAP scrNG[4800] = {0x00};
GBitmap2 screen = {160, 240, scrN, scrNG};
```

main.cpp

```
////////////////////////////////////////////////////////////////////
// Définitions des fonctions et variables globales

// Librairies
#include <stdlib.h>
#include <string.h>
#include "GaumerieLib.h"
#include "bitmaps.h"

class PegPresentationManager {};
typedef unsigned char byte;
typedef unsigned char ID_MESSAGE;

// Fonctions externes
extern "C" {
    void LibGetTime2(byte *hour2, byte *minute2, byte *second2);
    void Bevt_Request_LaunchMenu(void);
}

////////////////////////////////////////////////////////////////////
// Définir toutes les variables globales ici                      //


////////////////////////////////////////////////////////////////////
// Fonctions du jeu                                               //


////////////////////////////////////////////////////////////////////
// Fonction principale

void main()
{
    // Tout se déroule ici
    // Dès qu'on quitte cette fonction, le programme se termine

}

////////////////////////////////////////////////////////////////////
// Zone réservée, ne pas modifier

void PegAppInitialize(PegPresentationManager *pPresentation)
{
    byte h, m, s;
```

```
    LibGetTime2(&h, &m, &s);
    srand(3600*(int)h+60*(int)m+(int)s);
    GrayScales::Initialize();
    GrayScales::SetBuffers(screen.tuc_Bmp, screen.tuc_Bmp2);
    GrayScales::Start();
    main();
    GrayScales::Stop();
    GrayScales::Restore();
    SH3Timer::StopAll();
    Bevt_Request_LaunchMenu();
}

extern "C" char *ExtensionGetLang(ID_MESSAGE MessageNo)
{
       return "";
}
```

Utilisation

Le coeur du programme se déroule dans la fonction `main()`. Un `GBitmap2` pour l'écran a aussi été créé dans bitmaps.h, il est nommé `screen` par défaut.

6 – A propos...

GaumerieLib

Librairie C++ développée par Alexandre Rion (Gaumerie).
Tutoriel écrit par Thê-Vinh Truong (Kilburn).