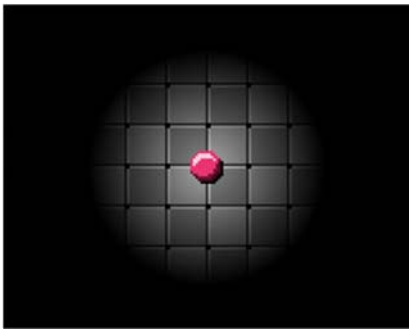
Éclairage Tamisé en Haute Résolution

Contourner gb.display avec gb.tft



Dernière modification le 23 novembre 2018 à 16:31 GMT+4

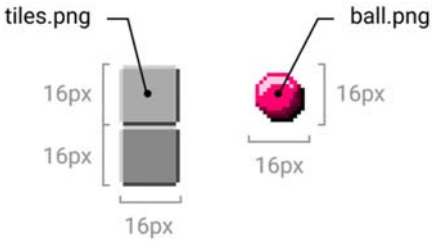
Table des matières

Quel est notre objectif ?

Pour cet exercice, nous allons réaliser une petite scène de jeu très simple, dans laquelle il s'agira de déplacer une petite bille sur un dallage régulier semblable à un échiquier *infini*. Les commandes de mouvement de notre bille seront appliquées à l'aide du `PAD` de la console. Je vous laisse découvrir dans cette petite vidéo le résultat que nous chercherons *grosso modo* à atteindre tout au long de ce tutoriel :
L'objet de ce tutoriel consistera à détailler comment réaliser l'*éclairage* de la scène de jeu. Le dallage sera plongé dans le noir et partiellement éclairé à l'endroit où se trouve la bille (le centre de l'écran ici). L'intensité lumineuse étant inversement proportionnelle au carré de la distance quand on s'éloigne de la bille.

Les éléments graphiques

Nous n'aurons besoin que de deux petits éléments graphiques pour réaliser le rendu de la scène de jeu. Je les ai dessinés avec l'éditeur en ligne Piskel, qui est un outil très bien fait et gratuit :



Vous pouvez télécharger ces éléments graphiques… vous n'aurez pas à les dessiner !

Lecture préliminaire

La technique d'affichage que nous utiliserons tout au long de ce tutoriel n'est probablement pas celle que vous avez l'habitude d'appliquer… En effet, pour pouvoir gérer l'affichage en haute résolution (160x128 pixels), nous ne pourrons malheureusement pas nous appuyer sur l'API `gb.display` standard qui ne permet pas de gérer un affichage en RGB565 (codage sur 16 bits) avec seulement 32 Ko de RAM. Nous ne pourrons donc pas bénéficier des fonctions très utiles que vous avez l'habitude d'utiliser, comme `drawPixel()`, `drawCircle()`, `fillRect()` ou encore `drawImage()`. Nous devrons recoder entièrement l'affichage de nos éléments graphiques… bits à bits… pour pouvoir attaquer directement le contrôleur DMA (*Direct Memory Access*) en passant par l'API `gb.tft`. C'est la seule solution pour gérer un affichage en haute résolution en 65 536 couleurs (et avec des performances raisonnables). Notre très estimé Andy nous a livré un article très intéressant sur le sujet, dans lequel il détaille comment mettre en oeuvre cette technique de *bas niveau*. Nous aurons l'occasion d'y revenir dans le prochain chapitre, mais je vous encourage vivement à passer cet article à la loupe avant d'aller plus loin ! Il vous évitera d'aller déchiffrer le fichier Display-ST7735.cpp dans la bibliothèque Gamebuino-Meta, qui met en œuvre l'interface avec l'écran TFT Adafruit 1,8" de la META, et qui vous apparaîtra sûrement indigeste *a priori*. Mais la récompense est de taille ! Cette technique vous ouvre les portes de la haute résolution en full RGB565. Un grand merci Andy pour nous avoir donné les clefs du contrôleur DMA !

Méthode d'affichage

La méthode d'Andy

La technique décrite par Andy dans son article High Resolution without gb.display est relativement simple mais terriblement astucieuse. L'idée générale consiste à effectuer le rendu graphique dans une zone tampon en mémoire, avant de la déverser vers le bus d'interfaçage de l'écran TFT. Néanmoins, il part du constat que la Gamebuino META ne dispose pas de suffisamment de mémoire (sa RAM étant limitée à 32 Ko) pour pouvoir contenir un tampon de la taille de l'écran en haute résolution avec une profondeur de couleurs de 16 bits (la mémoire nécessaire s'élève à 40 Ko). C'est la raison pour laquelle on ne peut pas utiliser `gb.display`.

Il propose donc de découper la surface de rendu à calculer en tranches suffisamment petites, de sorte qu'elles puissent être contenues en RAM à tour de rôle, sous la forme d'un petit tampon. Le calcul du rendu est ainsi effectué sur ce tampon, et une fois que le calcul est terminé, le tampon est alors recopié vers l'écran pour être affiché. Puis on passe à la tranche suivante, et ainsi de suite… Si le cycle de rendu partiel (calcul + recopie vers l'écran) de chaque tranche est suffisamment rapide, la scène complète du jeu peut alors être affichée sans que l'on puisse distinguer chacune des étapes successives.
L'idée géniale d'Andy consiste à utiliser le contrôleur DMA pour effectuer les transferts entre la mémoire et le périphérique d'affichage. L'acronyme `DMA` signifie *Direct Memory Access*, ce qui veut dire un accès direct à la mémoire. Il est en effet possible d'opérer un transfert **direct** de données via un contrôleur spécialisé entre la mémoire principale de la console et le périphérique d'affichage. Ce procédé ne fait intervenir le processeur que pour initier et conclure le transfert à travers des interruptions. Par conséquent le transfert DMA offre un moyen plus rapide pour l'échange de blocs de données entre la mémoire et l'écran. Cerise sur le gâteau : pendant que le transfert s'effectue, le processeur est au repos. On peut donc le solliciter pour effectuer d'autres tâches en parallèles ! Et c'est là que toute la magie de cette technique opère…
Il s'agit en effet de gérer **2** tampons en parallèle : pendant que le processeur s'active à calculer le rendu sur le premier tampon, le contrôleur DMA se charge de transférer le second vers l'écran. Et c'est précisément en utilisant `gb.tft` que nous allons pouvoir réaliser ces opérations dans notre code C++.

Mise en œuvre

Vous avez toute la liberté d'organiser le découpage de la surface complète de rendu, en délimitant les tranches comme bon vous semble. Il vous suffit de déterminer l'agencement le plus approprié au rendu que vous devez réaliser. Les méthodes les plus simples (mais pas nécessairement les plus performantes selon le cas) consistent à découper la surface de rendu en tranches régulières horizontales ou verticales.

Tout au long de ce tutoriel, nous nous appuierons sur un découpage en tranches horizontales. Chaque tranche couvrira la largeur totale de l'écran, sur une hauteur de 8 pixels. Nous pouvons donc calculer la taille nécessaire à allouer en mémoire pour accueillir un tampon correspondant à chacune de ces tranches :
Taille d'un tampon : (160 x 8 pixels) x (16 bits) = 20 480 bits = 2 560 octets = 2,5 Ko

Comme nous devons gérer deux tampons en parallèle, il nous suffira donc de réserver 2 x 2,5 = 5 Ko de RAM pour pouvoir réaliser le rendu complet de notre scène de jeu avec 65 536 couleurs… C'est génial, nan ?

La mise en œuvre est donc assez simple : il suffira de parcourir l'ensemble des tranches une à une, et pour chaque nouvelle tranche, on alternera l'écriture des bits sur le tampon 1 ou le tampon 2.
Que devrons-nous écrire sur le tampon ? Et bien, pour chaque point `(x,y)` de la tranche en cours, nous devrons déterminer la couleur du pixel qui sera affiché à cette position. Nous reporterons donc sur le tampon la valeur de l'entier, codé sur 16 bits, correspondant à cette couleur.
Pour déterminer la couleur du pixel à afficher, il suffira de parcourir l'ensemble des éléments graphiques de la scène de jeu et de déterminer quels sont ceux qui ont une intersection non vide avec la tranche courante. Il ne restera plus qu'à projeter les coordonnées `(x,y)` dans le référentiel de chaque élément graphique pour y piocher le code de la couleur à recopier dans le tampon. Les éléments graphiques seront parcourus dans l'ordre de la pile d'affichage (du plus éloigné au plus proche). On pourrait même essayer d'optimiser les choses car un pixel de profondeur `n` masque nécessairement tous les pixels dont la profondeur est supérieure à `n`. Il est donc inutile d'explorer tous les pixels des éléments graphiques ayant une profondeur supérieure à `n`.

Voici une petite animation illustrant le processus de rendu pour une scène de jeu très simple où seule la bille est présente :

Remarque importante :
L'ordre dans lequel les octets sont organisés en mémoire diffère selon qu'ils sont destinés au CPU ou à l'écran TFT ! Cet ordre est désigné par le terme ***endianness****.*
Quand le CPU de la META enregistre un entier sur 16 bits en mémoire, par exemple `0xA3B7` *en notation hexadécimale, il l'enregistre dans des octets dans l'ordre inversé* `B7 A3` *pour une structure de mémoire fondée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet. Ainsi l'octet de poids le plus faible (ici* `B7`*) est enregistré à l'adresse mémoire la plus petite (le poids le plus faible en premier). L'octet de poids supérieur (ici* `A3`*) est enregistré à l'adresse mémoire suivante, et ainsi de suite. Les architectures qui respectent cette règle sont dites* ***little-endian****.*
Dans le cas de l'écran TFT, c'est exactement l'inverse : l'octet de poids le plus fort (`A3`*) est enregistré à l'adresse mémoire la plus petite. L'octet de poids inférieur (*`B7`*) est enregistré à l'adresse mémoire suivante, et ainsi de suite. Les architectures qui respectent cette règle sont dites* ***big-endian****.*

Il faudra donc penser à inverser les octets deux à deux lorsque l'on écrira dans le tampon qui est destiné à être envoyé à l'écran (par le biais du contrôleur DMA), sans quoi les couleurs affichées risquent de ne pas correspondre à ce que vous attendiez !

Moteur de rendu

Organisation de l'architecture du jeu

Après tout ce blabla, nous pouvons enfin passer à l'implémentation. Nous allons commencer par mettre en place le moteur de rendu de notre application. Mais juste auparavant, organisons un peu les choses.

Définissons les constantes globales qui nous seront utiles par la suite :

constants.h

```
#ifndef SHADING_EFFECT_CONSTANTS
#define SHADING_EFFECT_CONSTANTS


#define SCREEN_WIDTH 160
#define SCREEN_HEIGHT 128


#endif
```

Puis définissons le *croquis* de notre application `ShadingEffect.ino` :

ShadingEffect.ino

```
#include <Gamebuino-Meta.h>
#include "GameEngine.h"

void setup() {
    gb.begin();

    // nous n'utiliserons pas le tampon graphique standard
    // défini par `gb.display`, donc initialisons-le avec
    // une taille nulle pour qu'il n'occupe pas inutilement
    // de la place en mémoire :
    gb.display.init(0, 0, ColorMode::rgb565);

    // initialisation du contrôleur principal
    GameEngine::init();
}

void loop() {
    while(!gb.update());

    // exécution de la boucle de contrôle principale
    GameEngine::tick();
}
```

Vous pouvez noter que nous allons confier le contrôle global de l'application au composant `GameEngine` :

GameEngine.h

```
#ifndef SHADING_EFFECT_GAME_ENGINE
#define SHADING_EFFECT_GAME_ENGINE

class GameEngine
{
    public:

        static void init();
        static void tick();
};
#endif
```

Que nous allons définir de la façon suivante à ce stade :

GameEngine.cpp

```
#include "GameEngine.h"

// le moteur de rendu
#include "Renderer.h"

void GameEngine::init() {
    // rien de spécial à faire pour le moment
}

void GameEngine::tick() {
    // exécution du rendu de la scène de jeu
    Renderer::draw();
}
```

Nous voyons ici que la boucle principale est simplement chargée d'exécuter le rendu de la scène de jeu, qui est confié au moteur de rendu `Renderer`.

Définition du moteur de rendu

Nous allons ici appliquer strictement la méthode d'Andy décrite au chapitre précédent pour définir notre moteur de rendu :

Renderer.h

```
#ifndef SHADING_EFFECT_RENDERER
#define SHADING_EFFECT_RENDERER

#include <Gamebuino-Meta.h>
#include "constants.h"

// définition de la hauteur des tranches
#define SLICE_HEIGHT 8

class Renderer
{
    private:
        // déclaration des deux tampons mémoires dans lesquels
        // seront réalisés les calculs de rendu
        static uint16_t buffer1[SCREEN_WIDTH * SLICE_HEIGHT];
        static uint16_t buffer2[SCREEN_WIDTH * SLICE_HEIGHT];
        // témoin qui nous indiquera si un transfert de mémoire
        // vers le contôleur DMA est en cours ou pas
        static bool drawPending;

        // méthode chargée d'initier le transfert mémoire vers le contrôleur DMA
        static void customDrawBuffer(int16_t x, int16_t y, uint16_t* buffer, uint16_t
w, uint16_t h);
        // méthode chargée d'attendre que le transfert soit terminé
        // et de clôturer la transaction avec le contrôleur DMA
        static void waitForPreviousDraw();

    public:

        // exécution du rendu de la scène de jeu
        static void draw();
};

#endif
```

Et voici la définition des méthodes magiques :

Renderer.cpp

```
#include "Renderer.h"

// définition des deux tampons mémoire
uint16_t Renderer::buffer1[SCREEN_WIDTH * SLICE_HEIGHT];
uint16_t Renderer::buffer2[SCREEN_WIDTH * SLICE_HEIGHT];

// pour le moment, aucun transfert de mémoire n'est en cours
bool Renderer::drawPending = false;

// la routine magique relative au contrôleur DMA...
// jetez un coup d'oeil dans la bibliothèque officielle
// si cela attise votre curiosité :
// Gamebuino-META/src/utility/Display-ST7735/Display-ST7735.cpp
namespace Gamebuino_Meta {
  #define DMA_DESC_COUNT (3)
  extern volatile uint32_t dma_desc_free_count;

  static inline void wait_for_transfers_done(void) {
    while (dma_desc_free_count < DMA_DESC_COUNT);
  }

  static SPISettings tftSPISettings = SPISettings(24000000, MSBFIRST, SPI_MODE0);
};

// méthode chargée d'initier le transfert mémoire vers le contrôleur DMA...
// ces opérations sont également définies dans :
// Gamebuino-META/src/utility/Display-ST7735/Display-ST7735.cpp
void Renderer::customDrawBuffer(int16_t x, int16_t y, uint16_t* buffer, uint16_t w,
uint16_t h) {
  drawPending = true;
  gb.tft.setAddrWindow(x, y, x + w - 1, y + h - 1);
  SPI.beginTransaction(Gamebuino_Meta::tftSPISettings);
  gb.tft.dataMode();
  gb.tft.sendBuffer(buffer, w*h);
}

// méthode chargée d'attendre que le transfert soit terminé
// et de clôturer la transaction avec le contrôleur DMA...
// idem, tout est dans :
// Gamebuino-META/src/utility/Display-ST7735/Display-ST7735.cpp
void Renderer::waitForPreviousDraw() {
  if (drawPending) {
    // la petite routine magique définie plus haut
    Gamebuino_Meta::wait_for_transfers_done();
    gb.tft.idleMode();
    SPI.endTransaction();
    drawPending = false;
  }
}
```

```
// exécution du rendu de la scène de jeu
void Renderer::draw() {
  // on calcule le nombre de tranches horizontales à découper
  uint8_t slices = SCREEN_HEIGHT / SLICE_HEIGHT;
  // puis on parcourt chaque tranche une à une
  for (uint8_t sliceIndex = 0; sliceIndex < slices; sliceIndex++) {

    // on définit un pointeur qui alternera entre les deux tampons mémoire
    uint16_t* buffer = sliceIndex % 2 == 0 ? buffer1 : buffer2;
    // on calcule l'ordonnée de la première frange horizontale de la tranche
    uint8_t sliceY = sliceIndex * SLICE_HEIGHT;

    // ---------------------------------------------------------------------
    // ici nous effectuerons les calculs de rendu de nos éléments graphiques
    // à partir des 3 paramètres suivants :
    // - sliceY       : ordonnée de la tranche courante (frange supérieure)
    // - SLICE_HEIGHT : hauteur des tranches
    // - buffer       : pointeur vers le tampon courant
    // ---------------------------------------------------------------------

    // puis on s'assure que l'envoi du tampon précédent
    // vers le contrôleur DMA a bien eu lieu
    if (sliceIndex != 0) waitForPreviousDraw();
    // après quoi on peut alors envoyer le tampon courant
    customDrawBuffer(0, sliceY, buffer, SCREEN_WIDTH, SLICE_HEIGHT);
  }

  // toujours attendre que le transfert DMA soit bien terminé
  // pour la dernière tranche avant de sortir de la méthode !
  waitForPreviousDraw();
}
```

Il nous reste maintenant à définir précisément le calcul du rendu de chaque élément graphique présent sur la scène de jeu. Nous ne les avons pas encore définis, mais nous allons le faire bientôt. Toutefois, auparavant, pour introduire un peu de généricité au niveau de notre moteur de rendu, nous allons définir un mécanisme bien pratique : le *patron* **Observateur**.

Observateurs et observables

Le patron de conception Observateur

Le *Design Pattern* **Observateur** (ou ***Observer***) est utilisé pour envoyer un signal à des objets qui jouent le rôle d'*observateurs*. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les objets qu'ils observent (les *observables*). Les notions d'observateur et d'observable permettent de limiter le couplage entre les objets aux seuls phénomènes à observer. Elles permettent également une gestion simplifiée d'observateurs multiples sur un même objet observable.

Dans ce modèle, le sujet observable se voit attribuer une collection d'observateurs qu'il notifie lors de changements d'états. Chaque observateur est chargé de faire les mises à jour adéquates en fonction des changements notifiés. Ainsi, l'observé n'est pas responsable des changements qu'il impacte sur les observateurs.
Nous allons appliquer ce patron dans le cadre de notre application. L'observable sera bien évidemment le moteur de rendu `Renderer` et les observateurs seront tout simplement les objets affichables sur la scène de jeu : la bille et le dallage sur lequel se déplace la bille.
Comme nous l'avons mentionné plus haut, l'observable se voit attribuer une collection d'observateurs. Nous aurons donc besoin d'une structure de données pour gérer cette collection. Et pour garder une certaine généricité, c'est-à-dire conserver une relative indépendance des types de données des observateurs, nous allons définir une **interface** pour qualifier nos observateurs de manière générique. Une interface est une sorte de contrat auquel doivent se plier tous les objets qui y souscrivent.
Nous pourrions ainsi définir une interface `Renderable` qui garantisse que tout objet ayant souscrit à ce contrat est capable de se dessiner (et donc d'effectuer un rendu de son apparence graphique). De cette façon, le moteur de rendu aurait juste à gérer une collection d'objets `Renderable`, sans avoir véritablement besoin de connaître la nature exacte des observateurs. En effet, il a juste besoin de s'assurer qu'ils sont capables de se dessiner.

Définition de l'interface Renderable

Cette interface peut se définir très simplement :

Renderable.h

```
#ifndef SHADING_EFFECT_RENDERABLE
#define SHADING_EFFECT_RENDERABLE

#include <Gamebuino-Meta.h>

class Renderable
{
    public:

        // méthode virtuelle pure
        virtual void draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) =
0;
};

#endif
```

La notion d'interface n'existe pas au sens strict du terme en C++, comme on peut la trouver en Java ou en Ada par exemple. Mais il existe une possibilité pour l'émuler en s'appuyant sur la notion de classe abstraite et de méthode virtuelle *pure*. Une méthode virtuelle pure est une méthode qui est déclarée mais non définie dans une classe. Elle est définie dans une des classes dérivées. Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de « `= 0` ». La méthode doit également être déclarée avec le mot-clef `virtual`.

Donc, chaque objet répondant à l'*interface* `Renderable` devra implémenter la méthode `draw()` pour se dessiner lorsqu'il recevra une notification du moteur de rendu `Renderer` pour le prévenir que c'est à son tour de se dessiner. Et souvenez-vous de ce que nous avions mentionné dans la définition de la classe `Renderer` :

Renderer.cpp

```
// exécution du rendu de la scène de jeu
void Renderer::draw() {
    // ...
    // ------------------------------------------------------------------
    // ici nous effectuerons les calculs de rendu de nos éléments graphiques
    // à partir des 3 paramètres suivants :
    // - sliceY       : ordonnée de la tranche courante (frange supérieure)
    // - SLICE_HEIGHT : hauteur des tranches
    // - buffer       : pointeur vers le tampon courant
    // ------------------------------------------------------------------
    // ...
}
```

Nous retrouvons donc ces 3 paramètres comme arguments de la fonction `draw()`.
Bien, maintenant que le *contrat* est établi, il faut ajouter à notre moteur de rendu la capacité de gérer une collection d'observateurs `Renderable`. Nous pourrions gérer cela avec un simple tableau… l'ennui, c'est qu'en C++ un tableau doit nécessairement être dimensionné de manière à ce que l'espace mémoire nécessaire à le contenir soit réservé. Or, dans un jeu, le nombre d'objets présents sur la scène de jeu et devant être affichés peut être très variable. Il est donc dangereux ou limitatif de définir la taille d'un tel tableau *a priori*. On peut même avoir tendance à vouloir allouer plus d'espace que nécessaire pour éviter d'atteindre les limites du nombre d'objets pouvant être référencés dans ce tableau. C'est donc définitivement une très mauvaise idée !

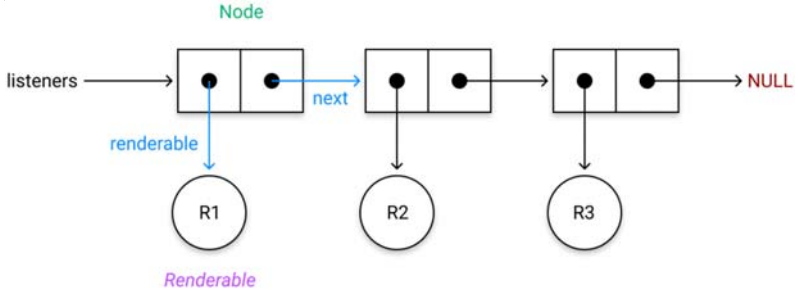
On préfèrera dans ce cas gérer une structure de données dynamique, dont la taille peut varier selon les besoins au cours de l'exécution du programme. L'une des structures de données dynamiques bien connues parmi les plus simples en algorithmique est la *liste chaînée*.

Collection dynamique d'observateurs

Nous allons donc attribuer une liste d'observateurs à notre moteur de rendu. Cette liste sera désignée par la variable statique `listeners` (les observers sont également appelés des *écouteurs*). Alors, faudra-t-il déclarer cette variable de cette façon ?

```
static Renderable* listeners;
```

NON ! C'est un peu plus compliqué que ça.
Les observateurs devront être rangés dans des boîtes que nous allons enchaîner les unes aux autres avec des pointeurs. Chacune de ces boîtes sera considérée comme un chaînon de la liste. Ces chaînons constitueront les *nœuds* (`Node`) de la liste. Et voici comment va se présenter la structure de cette liste chaînée :



Chaque `Node` est constitué de 2 pointeurs :

1. `renderable` qui est un pointeur vers un objet de type `Renderable`
2. `next` qui est un pointeur vers un objet de type `Node`

Le pointeur `renderable` permet donc d'accéder à l'objet de type `Renderable` qui est encapsulé dans le nœud, tandis que le pointeur `next` permet d'accéder au nœud suivant de la liste.
La déclaration correcte de notre variable statique `listeners` est donc :

```
static Node* listeners;
```

Et le dernier nœud de la liste aura son pointeur `next` qui pointe sur NULL (rien).

Voyons maintenant comment implémenter cette liste chaînée. Pour cela nous allons avoir besoin de déclarer la classe `Node` :

Node.h

```
#ifndef SHADING_EFFECT_NODE
#define SHADING_EFFECT_NODE

#include "Renderable.h"

class Node
{
    private :

        // pointeur vers l'objet de type `Renderable`
        Renderable* renderable;
        // pointeur vers le noeud suivant de la liste
        Node* next;

    public:

        // constructeur du noeud
        Node(Renderable* renderable);
        // destructeur du noeud
        ~Node();

        // méthode d'accès à l'objet `Renderable` encapsulé
        Renderable* getRenderable();
        // méthode d'accès au noeud suivant de la liste
        Node* getNext();

        // méthode de recherche d'un objet `Renderable` en particulier
        Node* search(Renderable* renderable);
        // méthode qui permet d'ajouter un nouveau noeud à la liste
        void add(Renderable* renderable);
        // méthode qui permet de supprimer un noeud de la liste
        void del(Renderable* renderable);
};

#endif
```

Dont voici la définition :

Node.cpp

```
#include "Node.h"

// le constructeur initialise les attributs du noeud
Node::Node(Renderable* renderable) : renderable(renderable), next(NULL) {};

// le destructeur "oublie" les objets qu'il référençait
Node::~Node() {
    this->renderable = NULL;
    this->next = NULL;
}

// permet d'accéder à l'objet `Renderable` encapsulé
Renderable* Node::getRenderable() {
    return this->renderable;
}

// permet d'accéder au noeud suivant de la liste
Node* Node::getNext() {
    return this->next;
}

// recherche un objet `Renderable` en particulier
// et retourne un pointeur sur le noeud détenteur de cet objet
Node* Node::search(Renderable* renderable) {
    if (this->renderable == renderable) {
        return this;
    }

    // si on arrive en fin de liste, c'est que
    // l'objet `Renderable` recherché n'a pas été trouvé
    if (this->next == NULL) {
        return NULL;
    }

    // si le noeud courant n'en est pas le détenteur,
    // alors on effectue la recherche dans le noeud suivant,
    // et ainsi de suite...
    return this->next->search(renderable);
}

// ajoute un noeud en FIN de liste
void Node::add(Renderable* renderable) {
    if (this->next == NULL) {
        this->next = new Node(renderable);
    } else {
        this->next->add(renderable);
    }
}

// supprime le noeud détenteur de l'objet `Renderable`
// que l'on cherche à indentifier...
void Node::del(Renderable* renderable) {
    if (this->next != NULL) {
        if (this->next->getRenderable() == renderable) {
            Node* toDelete = this->next;
            this->next = toDelete->getNext();
            delete toDelete;
        } else {
            this->next->del(renderable);
        }
    }
}
```

Vous voyez que dans le cas de la suppression d'un nœud, on part du principe ici que le nœud courant n'encapsule pas l'objet `Renderable` que l'on recherche. En effet, la recherche s'effectue à partir du nœud suivant. C'est normal, ne vous inquiétez pas, la suppression du nœud courant ne peut pas se faire ici. En effet, si l'on supprimait le nœud courant… alors le nœud précédent perdrait sa référence vers son suivant (qui est justement le nœud courant)… et perdrait du même coup toute la liste ! On ne peut donc pas traiter ce cas ici. Nous le traiterons dans la classe `Renderer` qui détient la référence à la tête de la liste.

Enregistrement des observateurs

Rappelez-vous qu'en définitive, la gestion de cette liste chaînée doit nous permettre d'assurer la gestion des observateurs du `Renderer`. Et donc, le fait d'ajouter ou de supprimer un observateur dans cette liste se traduit, du point de vue du `Renderer`, comme le fait de permettre à un objet de type `Renderable` de pouvoir s'abonner ou se désabonner aux notifications du `Renderer`. Voilà donc comment compléter la déclaration de la classe `Renderer` :

Renderer.h

```
#ifndef SHADING_EFFECT_RENDERER
#define SHADING_EFFECT_RENDERER

#include <Gamebuino-Meta.h>
#include "Node.h"
#include "Renderable.h"
#include "constants.h"

#define SLICE_HEIGHT 8

class Renderer
{
    private:

        // pointeur vers la liste d'observateurs
        static Node* listeners;

    public:

        // abonnement d'un observateur
        static void subscribe(Renderable* renderable);
        // désabonnement d'un observateur
        static void unsubscribe(Renderable* renderable);
        // permet de savoir si un objet de type `Renderable` est déjà abonné
        static bool hasSubscribed(Renderable* renderable);

    // toutes les autres déclaration restent inchangées
};

#endif
```

Voilà comment définir ces nouvelles méthodes :

Renderer.cpp

```
// par défaut, la liste est vide
Node* Renderer::listeners = NULL;

// abonnement d'un observateur
void Renderer::subscribe(Renderable* renderable) {
  // si la liste est vide, on l'initialise avec le nouvel observateur ;-)
  if (listeners == NULL) {
    listeners = new Node(renderable);
  } else {
    // sinon on délègue l'ajout au noeud suivant
    // puisque l'ajout doit se faire en FIN de liste
    listeners->add(renderable);
  }
}

// désabonnement d'un observateur
void Renderer::unsubscribe(Renderable* renderable) {
  // si la liste est vide, il n'y a rien à faire !
  // par contre, si elle ne l'est pas...
  if (listeners != NULL) {
    // si le noeud de tête de la liste est bien celui
    // qui encapsule l'observateur que l'on cherche à identifier
    if (listeners->getRenderable() == renderable) {
      // on récupère la référence vers le noeud suivant
      // qui va devenir la nouvelle tête de liste
      Node* next = listeners->getNext();
      // et on peut donc définitivement supprimer le premier noeud
      delete listeners;
      // on raccroche le nouveau noeud de tête de liste
      listeners = next;
    } else {
      // dans le cas contraire, on délègue la suppression
      // au noeud suivant dans la liste...
      listeners->del(renderable);
    }
  }
}

// pour savoir si un objet de type `Renderable` est déjà abonné
bool Renderer::hasSubscribed(Renderable* renderable) {
  // il suffit d'identifer le noeud qui en est le détenteur
  return listeners->search(renderable) != NULL;
}
```

Vous voyez que notre problème de suppression de nœud est désormais résolu.

Envoi des notifications

Il nous reste une dernière chose à faire ici ! Mettre en place le mécanisme de notification à l'ensemble des observateurs. Et oui, maintenant nous pouvons enfin demander à tous nos observateurs de se dessiner chacun leur tour dans l'ordre de la pile d'affichage, c'est-à-dire précisément dans l'ordre où ils sont rangés dans la liste d'abonnement. Le noeud de tête se dessine, puis le noeud suivant, et ainsi de suite, jusqu'à la fin de la liste :

Renderer.cpp

```
void Renderer::draw() {
  if (listeners != NULL) {
    uint8_t slices = SCREEN_HEIGHT / SLICE_HEIGHT;
    for (uint8_t sliceIndex = 0; sliceIndex < slices; sliceIndex++) {
      uint16_t* buffer = sliceIndex % 2 == 0 ? buffer1 : buffer2;
      uint8_t sliceY = sliceIndex * SLICE_HEIGHT;

      // il suffit d'envoyer le signal au noeud de tête
      // qui le relaiera à son tour au noeud suivant,
      // et ainsi de suite, jusqu'à la fin de la liste
      listeners->draw(sliceY, SLICE_HEIGHT, buffer);

      if (sliceIndex != 0) waitForPreviousDraw();
      customDrawBuffer(0, sliceY, buffer, SCREEN_WIDTH, SLICE_HEIGHT);
    }
    waitForPreviousDraw();
  }
}
```

Pour mettre en place ce mécanisme, il nous reste donc à ajouter dans la classe `Node` la prise en compte du signal de notification pour le relayer à l'objet de type `Renderable` qu'il encapsule. Donc, dans la classe `Node` nous allons déclarer une méthode `draw()` qui va se charger de transmettre cette notification :

Node.h

```
class Node
{
    public:

        // relaie la notification provenant du `Renderer`
        // qu'il est temps de se dessiner
        void draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer);
};
```

Node.cpp

```
void Node::draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) {
    // l'observateur se dessine
    this->renderable->draw(sliceY, sliceHeight, buffer);
    // et on passe le relais au noeud suivant s'il existe
    if (this->next != NULL) {
        this->next->draw(sliceY, sliceHeight, buffer);
    }
}
```

Et la boucle est bouclée.
Vous voyez que notre interface `Renderable` est bien utile ici. On s'appuie sur le seul fait que le contrat décrit par cette interface nous garantit que l'objet est capable de se dessiner. Le moteur de rendu n'a ici aucune connaissance de la nature exacte de ses observateurs. Et pourtant tout fonctionne à merveille.

Bien, il est temps maintenant de nous intéresser aux objets que nous allons dessiner sur la scène de jeu : la bille et le dallage sur lequel elle se déplace.

La Bille

Ajout de la bille sur la scène

Maintenant que la mécanique est bien huilée, nous pouvons enfin ajouter des objets sur notre scène de jeu et admirer tout le travail de préparation que nous venons d'accomplir !
Commençons donc par notre petite bille. Rappelez-vous que le jeu est gouverné par un contrôleur principal : `GameEngine`. C'est donc lui qui va se charger de placer la bille sur la scène de jeu :

GameEngine.h

```
#ifndef SHADING_EFFECT_GAME_ENGINE
#define SHADING_EFFECT_GAME_ENGINE

// nous allons définir la classe `Ball` juste après...
#include "Ball.h"

class GameEngine
{
    private:

        // un pointeur vers l'instance de la bille
        static Ball* ball;

    public:

        static void init();
        static void tick();
};

#endif
```

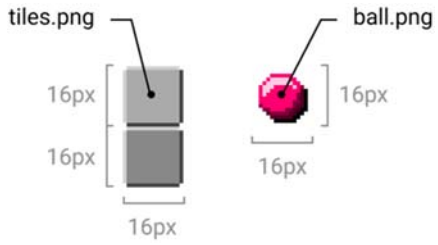
GameEngine.cpp

```
#include "GameEngine.h"
#include "Renderer.h"

// toujours initialiser un pointeur à NULL
Ball* GameEngine::ball = NULL;

void GameEngine::init() {
    // instantiation de la bille
    ball = new Ball();

    // enregistrement de la bille comme observateur
    // au sein du moteur de rendu
    Renderer::subscribe(ball);
}

void GameEngine::tick() {
    // effectue le rendu de la scène
    Renderer::draw();
}
```

Vous voyez que c'est tout simple ! Y'a rien de plus à faire en-dehors de la définition de la bille elle-même…

Préparation des éléments graphiques

Si vous ne l'avez pas déjà fait, téléchargez l'archive ZIP contenant les éléments graphiques du tutoriel et jetez un œil sur le fichier `ball.png` :



Voilà venu le moment d'intégrer cette image à notre code C++. Comment allons-nous procéder ? Rappelez-vous que nous devrons écrire les codes couleurs de tous les pixels qui composent cette bille dans le tampon `buffer` du moteur de rendu. Attention, le référentiel colorimétrique du fichier `ball.png` est le `RGB888`… or il faudra convertir ces couleurs en `RGB565`. Et de surcroît, rappelez-vous également que nous devrons faire en sorte que ces codes couleurs soient écrits selon l'ordre **big-endian** avant d'être envoyés au contrôleur DMA. Ouhlala… ça ne va pas être simple tout ça !

Et ben si ! J'ai publié un petit outil, il y a 2 semaines, qui permet justement de faire tout ça pour nous :

Image Transcoder for HD & gb.tft

Il vous suffit de paramétrer l'outil comme suit, et de glisser-déposer le fichier `ball.png` dans l'encart **Drag and drop your image here** (sur la page de l'outil bien sûr… pas ici !) :

Vous obtiendrez alors le code suivant :

```
// you can declare your sprites like this:

struct Sprite {
    int x, y;
    uint8_t width, height;
    uint8_t frames;
    uint16_t* data;
    uint16_t transparent;
};

// and after spritedata has been set
// you can initialize your sprite like that:
//
// Sprite mySprite = {
//     0, // choose an initial x value
//     0, // choose an initial y value
//     16,
//     16,
//     1,
//     (uint16_t*) &spritedata,
//     0xffff
```

```
// };

const uint16_t spritedata[] = {
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe,
0x0ef8, 0x0ef8, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe,
0x5cfe, 0x0ef8, 0x0ef8, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x5cfe, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x5cfe, 0x0ef8, 0x0780, 0xffff, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0ef8, 0x0ef8, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0ef8, 0x0780, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0ef8, 0x0780, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0780, 0x0780, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0780, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0780, 0x0450, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0xffff, 0x0780, 0x0780, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0780,
0x0780, 0x0450, 0x0450, 0x0000, 0x0000, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0x0450, 0x0780, 0x0780, 0x0780, 0x0780, 0x0780,
0x0450, 0x0450, 0x0000, 0x0000, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x0450, 0x0450, 0x0450, 0x0450, 0x0450,
0x0450, 0x0000, 0x0000, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff
};
```

La partie qui nous intéresse ici est la définition de la constante `spritedata`, que nous allons reporter dans notre classe `Ball`. J'ai choisi une couleur de transparence arbitraire qui ne fait pas partie des couleurs que j'ai utilisées pour dessiner la bille. J'ai choisi le blanc (`#ffff`), mais vous pouvez tout à fait choisir celle que vous voulez… en dehors de celles utilisées pour dessiner la bille !
Arrêtons-nous juste quelques instants sur ces codes couleurs. Prenons par exemple, le rose *flashy* de notre bille. Son code couleur est `#FF0071` en RGB888 et `#F80E` en RGB565. Vous pouvez remarquer que ce code couleur est remplacé par l'entier `0x0ef8` dans la définition de `spritedata`… ce qui signifie qu'il a été réécrit dans l'ordre **little-endian**…

*Heeeeuuuuuu…???…???… oui mais tu nous as dit qu'on devait l'écrire en **big-endian**parce-que l'écran TFT il bouffe du **big-endian** !!!???…*

Ha ha ha … moi aussi ça m'a déconcerté au départ !… Mais Soru m'a donné l'explication : dans la mesure où le processeur mange du **little-endian**, le compilateur C++ effectue justement la conversion des constantes que vous définissez dans votre code (qui sont écrites en big-endian qui est l'ordre naturel dans lequel nous écrivons les choses) pour les transformer en little-endian… donc si vous définissez une constante avec une écriture little-endian… et bien elle sera renversée et donc convertie en **big-endian**. Et c'est exactement ce qu'on veut puisque c'est ça que le processeur va balancer au contrôleur DMA. Voilà pourquoi les codes couleurs obtenus sont bien écrits en little-endian. Fallait y penser hein !

Allez, on peut continuer…

Définition de la bille

Commençons par déclarer la class `Ball` :

Ball.h

```
#ifndef SHADING_EFFECT_BALL
#define SHADING_EFFECT_BALL

#include "Renderable.h"

// voilà comment déclarer le fait que la classe `Ball` remplit le contrat
// défini par l'interface `Renderable` (qui est réalité une classe)
class Ball : public Renderable
{
    private:

        // les paramètres descriptifs du sprite
        static const uint8_t FRAME_WIDTH;
        static const uint8_t FRAME_HEIGHT;
        static const uint16_t TRANSPARENT_COLOR;
        // la carte des pixels que l'on a obtenue avec l'outil de transcodage
        static const uint16_t BITMAP[];

        // les coordonnées de la bille, qui sont constantes
        // puisque la bille est fixée au centre de l'écran
        static const uint8_t X_POS;
        static const uint8_t Y_POS;

    public:

        // la fameuse méthode qui permet de remplir le contrat `Renderable`
        void draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) override;
};

#endif
```

La déclaration qui consiste à dire que la classe `Ball` remplit le contrat défini par l'interface `Renderable` se fait par le biais de l'héritage en C++. Rappelez-vous que la notion d'interface au sens strict du terme n'existe pas ici. Et c'est en utilisant l'héritage multiple du C++ que vous pouvez faire en sorte que vos classes implémentent plusieurs interfaces, en dérivant de plusieurs classes qui définissent différents contrats.

Bien, passons maintenant à la définition de notre classe `Ball` :

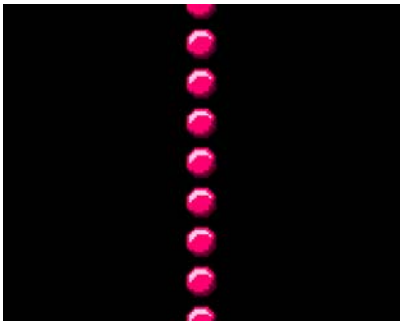
Ball.cpp

```
#include <Gamebuino-Meta.h>
#include "Ball.h"
#include "constants.h"

// on reporte les paramètres descriptifs de notre sprite
const uint8_t Ball::FRAME_WIDTH = 16;
const uint8_t Ball::FRAME_HEIGHT = 16;
const uint16_t Ball::TRANSPARENT_COLOR = 0xffff;
// on positionne la bille au centre de l'écran...
// notez que ces coordonnées correspondent au coin
// en haut à gauche de notre sprite
const uint8_t Ball::X_POS = (SCREEN_WIDTH - FRAME_WIDTH) / 2;
const uint8_t Ball::Y_POS = (SCREEN_HEIGHT - FRAME_HEIGHT) / 2;

// on reporte ici la valeur de la variable `spritedata`
// que nous a fourni l'outil de transcodage
const uint16_t Ball::BITMAP[] = {
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe,
0x0ef8, 0x0ef8, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe, 0x5cfe,
0x5cfe, 0x0ef8, 0x0ef8, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x5cfe, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x5cfe, 0x0ef8, 0x0780, 0xffff, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x5cfe, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0ef8, 0x0ef8, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x5cfe, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0ef8, 0x0780, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0ef8, 0x0780, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0780, 0x0780, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0x0780, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8,
0x0ef8, 0x0780, 0x0450, 0x0450, 0x0000, 0xffff,
    0xffff, 0xffff, 0xffff, 0x0780, 0x0780, 0x0ef8, 0x0ef8, 0x0ef8, 0x0ef8, 0x0780,
0x0780, 0x0450, 0x0450, 0x0000, 0x0000, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0x0450, 0x0780, 0x0780, 0x0780, 0x0780, 0x0780,
0x0450, 0x0450, 0x0000, 0x0000, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x0450, 0x0450, 0x0450, 0x0450, 0x0450,
0x0450, 0x0000, 0x0000, 0xffff, 0xffff, 0xffff,
```

```
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff
};

// et on définit la méthode de calcul du rendu de la bille
void Ball::draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) {
    // on détermine les bornes du sprite selon l'axe Y qui sont
    // situées à l'intérieur de la tranche courante
    int8_t startIndex = Y_POS <= sliceY ? 0 : Y_POS - sliceY;
    int8_t endIndex = Y_POS + FRAME_HEIGHT >= sliceY + sliceHeight ? sliceHeight - 1
: Y_POS + FRAME_HEIGHT - sliceY - 1;
    // on prépare une variable qui recevra tour à tour les codes couleurs
    // de chacun des pixels qui composent le sprite (dans la tranche courante)
    uint16_t value;
    // coordonnées du pixel traité
    uint8_t x,y;
    // et on balaie le sprite entre les bornes Y définies plus haut
    for (y = startIndex; y <= endIndex; y++) {
        // ainsi qu'entre les bornes selon l'axe X
        for (x = X_POS; x < X_POS + FRAME_WIDTH; x++) {
            // on va piocher la couleur du pixel correspondant
            value = BITMAP[(sliceY - Y_POS + y) * FRAME_WIDTH - X_POS + x];
            // et s'il ne s'agit pas de la couleur de transparence
            if (value != TRANSPARENT_COLOR) {
                // on la recopie dans le tampon
                buffer[x + y * SCREEN_WIDTH] = value;
            }
        }
    }
}
```

Premier rendu de la scène

Et ben c'est nickel ! Y a plus qu'à compiler et déverser tout ça sur la console pour enfin voir quelque-chose apparaître à l'écran…



C'est quoi ce bordel ???!!!…

Ha ha ha… c'est à peu près la tronche que j'ai tirée à la première exécution…

Mais l'explication est simple… une idée ?
En fait, regardez bien le code de la méthode de rendu `draw()` :

- de la bille
- et du moteur de rendu

À quel endroit voyez-vous que l'on écrit quelque-chose dans le buffer ? Nous le faisons exclusivement dans la méthode de rendu de la bille… et uniquement lorsqu'on est en présence de pixels dont la couleur n'est pas la couleur de transparence…

Bah oui ! Vous ne voyez toujours pas ? Regardons d'un peu plus près ce qui se passe réellement…
La surface de rendu est découpée en `128 / 8 = 16` tranches :

tranche	buffer	écriture	état résultant du buffer
n°1	`buffer1`	on n'écrit rien	que des zéros
n°2	`buffer2`	on n'écrit rien	que des zéros
n°3	`buffer1`	on n'écrit rien	que des zéros
n°4	`buffer2`	on n'écrit rien	que des zéros
n°5	`buffer1`	on n'écrit rien	que des zéros
n°6	`buffer2`	on n'écrit rien	que des zéros
n°7	`buffer1`	on n'écrit rien	que des zéros
n°8	`buffer2`	moitié supérieure de la bille	moitié supérieure de la bille
n°9	`buffer1`	moitié inférieure de la bille	moitié inférieure de la bille
n°10	`buffer2`	on n'écrit rien	moitié supérieure de la bille
n°11	`buffer1`	on n'écrit rien	moitié inférieure de la bille
n°12	`buffer2`	on n'écrit rien	moitié supérieure de la bille
n°13	`buffer1`	on n'écrit rien	moitié inférieure de la bille
n°14	`buffer2`	on n'écrit rien	moitié supérieure de la bille
n°15	`buffer1`	on n'écrit rien	moitié inférieure de la bille
n°16	`buffer2`	on n'écrit rien	moitié supérieure de la bille

Capito?

Bah voilà ce qui se passe… et au deuxième passage, le buffer n'est plus vide, donc les moitiés de bille sont dessinées sur toutes les tranches… alternativement… indéfiniment… Et quand on repasse sur les tranches où la bille doit effectivement être dessinée, ben on réécrit exactement la même chose dans les buffers… Donc rien ne change à l'affichage…
Le premier réflexe est de se dire : « *oh ben y a qu'à écrire des zéros au lieu de rien du tout* » … moé … si la bille était seule sur la scène de jeu, c'est effectivement ce que l'on pourrait faire… mais le hic, c'est qu'elle n'est pas toute seule ! Et en faisant cela, vous effaceriez tous les pixels des objets qui auraient été dessinés auparavant.

Par contre, la chance qu'on a dans cette histoire, c'est qu'il nous reste à effectuer le rendu du dallage… qui, lui, recouvre TOUTE la surface de l'écran… donc le problème sera réglé. En effet, le dallage se situe à une profondeur supérieure sur la pile d'affichage, il sera donc dessiné avant la bille… Par conséquent, il recouvrira toute la surface noire, et les tranches seront toujours remplies avec ses pixels… que l'on dessine la bille ou pas d'ailleurs.

Allez, rendez-vous au prochain chapitre pour attaquer le rendu du dallage !

Le dallage

Ajout du dallage sur la scène

Comme dans le cas de la bille, c'est au niveau du contrôleur principal `GameEngine` que l'on va introduire le dallage sur la scène de jeu :

GameEngine.h

```
#ifndef SHADING_EFFECT_GAME_ENGINE
#define SHADING_EFFECT_GAME_ENGINE

// nous allons définir la classe `Tiling` juste après...
#include "Tiling.h"
#include "Ball.h"

class GameEngine
{
    private:

        // un pointeur vers l'instance du dallage
        static Tiling* tiling;
        // un pointeur vers l'instance de la bille
        static Ball* ball;

    public:

        static void init();
        static void tick();
};

#endif
```

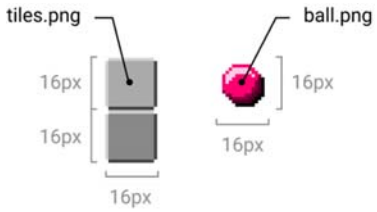
GameEngine.cpp

```
#include "GameEngine.h"
#include "Renderer.h"

// toujours initialiser un pointeur à NULL
Tiling* GameEngine::tiling = NULL;
Ball* GameEngine::ball = NULL;

void GameEngine::init() {
    // instantiation du dallage
    tiling = new Tiling();
    // instantiation de la bille
    ball = new Ball();

    // enregistrement des observateurs
    // au sein du moteur de rendu
    Renderer::subscribe(tiling);
    Renderer::subscribe(ball);
}

void GameEngine::tick() {
    Renderer::draw();
}
```

Vous noterez ici que l'enregistrement des observateurs auprès du moteur de rendu se fait selon l'ordre décroissant de la profondeur de l'élément sur la pile d'affichage. Ceci de manière à ce que le dallage soit dessiné en premier (le plus profond dans la pile) et la bille ensuite (rappelez-vous que l'ajout d'un élément dans la liste chaînée des observateurs se fait en queue de liste).

Préparation des éléments graphiques

Vous trouverez le fichier `tiles.png` dans la petite archive ZIP que vous avez téléchargée tout à l'heure. Vous voyez que cette fois, il s'agit d'une petite *spritesheet* comportant deux sprites : une dalle claire et une dalle sombre.



Voilà comment paramétrer l'outil de transcodage :

Faîtes glisser le fichier `tiles.png` sur l'encart de dépôt, et vous devriez obtenir ceci :

```
// you can declare your sprites like this:

struct Sprite {
    int x, y;
    uint8_t width, height;
    uint8_t frames;
    uint16_t* data;
    uint16_t transparent;
};

// and after spritedata has been set
```

```
// you can initialize your sprite like that:
//
// Sprite mySprite = {
//      0, // choose an initial x value
//      0, // choose an initial y value
//      16,
//      16,
//      2,
//      (uint16_t*) &spritedata,
//      0xffff
// };

const uint16_t spritedata[] = {
    0xffff, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce,
0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0xffff,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0xffff, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842,
0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0xffff,
    0xffff, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce,
0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0xffff,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
```

```
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0xffff, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842,
0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0xffff
};
```

Définition du dallage

Commençons par déclarer la classe `Tiling` :

Tiling.h

```
#ifndef SHADING_EFFECT_TILING
#define SHADING_EFFECT_TILING

#include "Renderable.h"

// la classe Tiling remplit le contrat défini par l'interface `Renderable`
class Tiling : public Renderable
{
    private:

        // les paramètres descriptifs du sprite
        static const uint8_t TILE_WIDTH;
        static const uint8_t TILE_HEIGHT;
        static const uint16_t TRANSPARENT_COLOR;
        // la carte des pixels que l'on a obtenue avec l'outil de transcodage
        static const uint16_t BITMAP[];

        // nous allons déplacer le dallage pour donner une impression de mouvement
        // à la bille, donc on définit les coordonnées du vecteur de déplacement
        // pour le prendre tout de suite en compte dans le calcul du rendu
        int8_t offsetX;
        int8_t offsetY;

    public:

        // on définit un constructeur dans lequel
        // nous initialiserons le vecteur de déplacement
        Tiling();
```

```
        // la méthode imposée par le contrat `Renderable`
        void draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) override;
};

#endif
```

Passons maintenant à la définition de notre classe `Tiling`. Je vais détailler et commenter chacune des étapes de calcul du rendu directement dans le code :

Tiling.cpp

```
#include <Gamebuino-Meta.h>
#include "Tiling.h"
#include "constants.h"

// on reporte les paramètres descriptifs de notre sprite
const uint8_t Tiling::TILE_WIDTH = 16;
const uint8_t Tiling::TILE_HEIGHT = 16;
const uint16_t Tiling::TRANSPARENT_COLOR = 0xffff;

// on reporte ici la valeur de la variable `spritedata`
// que nous a fourni l'outil de transcodage
const uint16_t Tiling::BITMAP[] = {
    // 1ère dalle
    0xffff, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce,
0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0xffff,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0x79ce, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad,
0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x55ad, 0x2842,
    0xffff, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842,
0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0xffff,
    // 2ème dalle
```

```
    0xffff, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce,
0x79ce, 0x79ce, 0x79ce, 0x79ce, 0x79ce, 0xffff,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0x79ce, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x518c,
0x518c, 0x518c, 0x518c, 0x518c, 0x518c, 0x2842,
    0xffff, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0x2842,
0x2842, 0x2842, 0x2842, 0x2842, 0x2842, 0xffff
};

// vient ensuite le constructeur, qui initialise le vecteur de déplacement
Tiling::Tiling() : offsetX(0), offsetY(0) {}

// et on définit la méthode de calcul du rendu du dallage
void Tiling::draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) {
    // nous allons précalculer certains paramètres
    // pour optimiser le temps de traitement...

    // on va chercher à numéroter les dalles selon les deux axes X et Y
    // pour leut associer des indices qui pourrait être nommés `tx` et `ty`
    // et on va alors calculer un témoin de parité sur ces deux indices :
    //   txodd = true lorsque `tx` est impair et false sinon
    //   tyodd = true lorsque `ty` est impair et false sinon
    bool txodd,tyodd;

    // il suffira alors de tester conjointement la parité de `tx` et `ty`
    // pour savoir si on doit afficher une dalle claire ou une dalle sombre
    // autrement dit, si l'on doit effectuer un "saut" dans la spritesheet
    // pour atteindre la dalle sombre... c'est le rôle du témoin `jump`
    bool jump;

    // et voici le décalage à appliquer dans la spritesheet
    // pour accéder aux couleurs de la dalle sombre
    uint16_t nfo = TILE_WIDTH * TILE_HEIGHT;
```

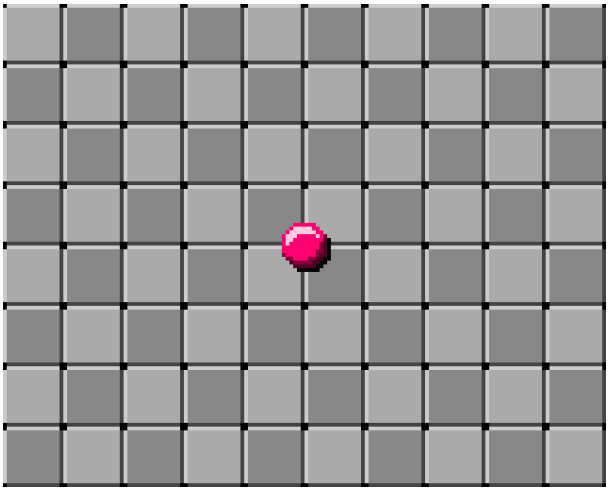
```
// tous les pixels (sx,sy) de la tranche courante vont être examinés...
// il faudra alors passer du système de coordonnées locales à la tranche
// (sx,sy) au système de coordonnées globales de l'écran (x,y)...
// et sx est en fait équivalent à x puisque la tranche couvre la largeur
// totale de l'écran :
//   x = sx
//   y = sy + sliceY
uint8_t sy,x,y;

// n'oublions pas que nous devons tenir compte du vecteur de déplacement
// (offsetX, offsetY) qui sera appliqué au dallage pour donner une impression
// de mouvement à la bille...
// il nous faudra donc transposer notre système de coordonnées globales (x,y) :
//   xo = x + offsetX
//   yo = y + offsetY
uint8_t xo,yo;

// lorsque nous devrons écrire dans le tampon, nous aurons affaire à
// un tableau unidimensionnel... il nous faudra donc projeter les
// coordonnées globales (x,y) sur l'indice correspondant dans le tampon :
//   buffer_index = x + (sy * SCREEN_WIDTH)
//                      -------syw-------
uint16_t syw;

// lorsque nous devrons aller piocher la couleur `value` dans la spritesheet
// qu'il faudra ensuite recopier dans le tampon, nous devrons calculer à
// quel indice `index` aller chercher cette couleur dans le tableau BITMAP
uint16_t index, value;

// le calcul de cet indice peut se décomposer en deux parties :
//   index = index_x + index_y, où :
//   index_x = (xo % TILE_WIDTH) + (jump * nfo)
//   index_y = (yo % TILE_HEIGHT) * TILE_WIDTH
uint16_t index_y;

// balayage de chaque pixel de la tranche (ici la composante Y)
for (sy = 0; sy < sliceHeight; sy++) {

    // passage du système bidimensionnel de la tranche
    // au système unidimensionnel du tampon
    syw = sy * SCREEN_WIDTH;

    // passage du système de coordonnées locales à la tranche
    // au système de coordonnées globales de l'écran
    y = sliceY + sy;

    // on applique la composante Y du vecteur de déplacement
    yo = y + this->offsetY;

    // on calcule le témoin de parité de la dalle selon l'axe Y
    tyodd = (yo / TILE_HEIGHT) % 2;

    // puis la composante Y de l'indice de lecture dans la spritesheet
    index_y = (yo % TILE_HEIGHT) * TILE_WIDTH;

    // balayage de chaque pixel de la tranche (ici la composante X)
    for (x = 0; x < SCREEN_WIDTH; x++) {
```

```
            // on applique la composante X du vecteur de déplacement
            xo = x + this->offsetX;

            // on calcule le témoin de parité de la dalle selon l'axe X
            txodd = (xo / TILE_WIDTH) % 2;

            // puis on détermine si on doit piocher le code couleur
            // dans une dalle claire ou une dalle sombre
            jump = txodd ^ tyodd;

            // l'indice de lecture dans la spritesheet peut maintenant
            // être entièrement déterminé
            index = index_y + (xo % TILE_WIDTH) + (jump * nfo);

            // il ne reste plus qu'à piocher le code couleur du sprite
            value = BITMAP[index];

            // et à le recopier dans le tampon s'il s'agit pas d'un
            // pixel transparent, sinon on fixe la couleur noire
            buffer[x + syw] = value != TRANSPARENT_COLOR ? value : 0;
        }
    }
}
```

Voilà, j'espère avoir été suffisamment clair sur le détail de toutes les étapes de calculs nécessaires à effectuer le rendu du dallage… C'était pas de la tarte !

Nouveau rendu de la scène

Y a plus qu'à admirer le résultat… on compile… on déverse le code machine sur la META… et voilà ce que ça donne désormais :



Bingo ! On a enfin une scène de jeu qui ressemble à quelque-chose. Autant vous avouer tout de suite qu'on a fait le plus dur !… Si, si, je vous l'assure !… le reste c'est de la crème à côté ! Et pourtant on n'a pas encore attaqué véritablement le sujet ciblé de ce tutoriel, à savoir le *shading*… Pourtant tous ces éléments étaient nécessaires. Il fallait mettre tout ça en place pour que les choses soient plus simples à digérer ensuite. Vous allez voir que ça n'est pas compliqué finalement…

Mais gardons encore un peu de suspense avant de tout dévoiler ! Nous allons d'abord ajouter un peu d'interactivité et de mouvement dans tout ça. Allez ! Direction le prochain chapitre…

Dynamique du mouvement

Gestion des événements utilisateur

Nous devons intercepter les événements lorsque l'utilisateur appuiera sur les boutons de la console de manière à impulser un mouvement à la bille. Et dans la mesure où la bille reste au centre de l'écran comme si elle était suivie par une caméra, l'illusion du mouvement sera donnée par le déplacement relatif de son environnement, c'est-à-dire le dallage. Rappelez-vous que nous avons déjà prévu dans le calcul de rendu du dallage un vecteur de déplacement `(offsetX, offsetY)`. Et bien, c'est justement ce vecteur que nous allons nous appliquer à calculer maintenant pour gérer l'impression de mouvement.
Commençons par mettre en place les éléments nécessaires à l'interception des événements utilisateurs. Pour simplifier les choses, nous allons le faire directement dans le moteur du jeu `GameEngine` :

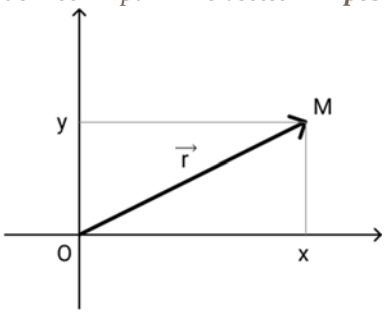
GameEngine.cpp

```
void GameEngine::tick() {

    // interception des événements utilisateur

    if (gb.buttons.repeat(BUTTON_LEFT, 1)) {
        tiling->left();
    }

    if (gb.buttons.repeat(BUTTON_RIGHT, 1)) {
        tiling->right();
    }

    if (gb.buttons.repeat(BUTTON_UP, 1)) {
        tiling->up();
    }

    if (gb.buttons.repeat(BUTTON_DOWN, 1)) {
        tiling->down();
    }

    // on délègue ensuite le calcul du mouvement au dallage
    // nous allons donc lui ajouter une boucle de contrôle
    // pour réaliser ces calculs
    tiling->tick();

    // et on effectue ensuite le rendu de la scène de jeu
    Renderer::draw();
}
```

Cinématique du point matériel

La gestion du mouvement est relativement simple. Nous allons nous appuyer sur la *cinématique du point matériel*. Je vous renvoie à vos cours de physique pour les détails théoriques… mais, en substance, nous retiendrons simplement ici les principes suivants :

- *Si l'on considère un référentiel spatial ayant pour origine le point* O *de coordonnées* $(0,0)$*, et par-rapport auquel on étudie le mouvement d'un point matériel* M*. La position du point* M *à l'instant* t *est donnée par le* ***vecteur position*** $\overrightarrow{OM}$ *(ou* $\overrightarrow{r}$*) de coordonnées* (x,y).



- *La vitesse moyenne entre deux positions successives* M *et* M' *du point matériel se définit comme le rapport entre la distance* MM' *parcourue et la durée* $\Delta t = t' - t$ *entre ces deux instants. Il s'agit d'une grandeur scalaire. En considérant des instants de plus en plus rapprochés, et donc en passant à la limite* $\Delta t \to 0$*, on peut définir une grandeur vectorielle instantanée appelée* ***vecteur vitesse*** *et définie par la dérivée temporelle du vecteur position :*

$$\overrightarrow{v} = \lim_{\Delta t \to 0} \overrightarrow{MM'} \Delta t = d\overrightarrow{r} dt v = \lim \Delta t \to 0 \overrightarrow{MM'} \Delta t = d\overrightarrow{r} dt$$

- *De la même manière on peut définir le* ***vecteur accélération*** *défini par la dérivée temporelle du vecteur vitesse :*

$$\overrightarrow{a} = d\overrightarrow{v} dt = d^2 \overrightarrow{r} dt^2 a = dv dt = d2r dt2$$

Ces équations sont très utiles… mais pour qu'il y ait mouvement, il faut qu'on introduise également la notion de dynamique qui s'appuie sur les *forces* mécaniques…

Principe fondamental de la dynamique

Ce principe désigne une loi physique mettant en relation la masse d'un objet et l'accélération qu'il reçoit si des forces lui sont appliquées. On l'appelle aussi la *deuxième loi de Newton* ou *relation fondamentale de la dynamique*, et elle s'énonce ainsi :

Soit un corps de masse m *constante non nulle, l'accélération subie par ce corps dans un référentiel galiléen est proportionnelle à la résultante des forces qu'il subit, et inversement proportionnelle à sa masse :*

$$\overrightarrow{a} = 1m \sum_i \overrightarrow{F_i} a = 1m \sum iFi$$

Pas trop mal à la tête ?… … mais nan … ce n'est pas si méchant, allez. Vous allez voir que tout ça va se traduire de manière très simple dans notre code.

Gestion du mouvement

Nous avons désormais toutes les clefs mathématiques et physiques pour mettre en oeuvre le mouvement de notre bille, transposé sur le déplacement du dallage pour simuler le suivi d'une caméra.
Pour ne pas faire intervenir de calculs trigonométriques, qui seraient plus coûteux pour le processeur, qui est déjà bien occupé avec le calcul du rendu de la scène, nous allons décider de représenter nos vecteurs position, vitesse et accélération par leurs coordonnées respectives (c'est-à-dire en les projetant sur les axes X et Y), et nous allons simplifier (drastiquement) la relation fondamentale de la dynamique. Pour cela, on va agglomérer la masse de la bille ainsi que la somme des forces qui lui sont appliquées sous la forme d'une simple impulsion motrice constante `PULSE=1` que l'on affectera telle quelle aux coordonnées `ax` et `ay` du vecteur accélération. Vous voyez qu'on ne va pas faire dans la dentelle… mais ça simplifiera énormément les calculs et soulagera le processeur.

Injectons maintenant toutes ces idées dans le code de la classe `Tiling` :

Tiling.h

```
#ifndef SHADING_EFFECT_TILING
#define SHADING_EFFECT_TILING

#include "Renderable.h"

// l'impulsion motrice constante
#define PULSE 1

class Tiling : public Renderable
{
    private:

        static const uint8_t TILE_WIDTH;
        static const uint8_t TILE_HEIGHT;
        static const uint16_t TRANSPARENT_COLOR;
        static const uint8_t BITMAP[];

        // les coordonnées du vecteur accélération
        float ax,ay;

        // les coordonnées du vecteur vitess
        float vx,vy;

        // les coordonnées du vecteur position ne sont autres
        // que celles de notre vecteur de déplacement
        int8_t offsetX,offsetY;

    public:

        // le constructeur devra désormais initialiser tous les vecteurs
        Tiling();

        // les commandes de déplacement invoquées par `GameEngine`
        void left();
        void right();
        void up();
        void down();

        // le point de raccordement de la boucle de contrôle
        void tick();

        void draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) override;
};

#endif
```

Voyons maintenant comment définir les nouveaux éléments que nous avons introduits :

Tiling.cpp

```
Tiling::Tiling() {
    this->ax = 0;
    this->ay = 0;
    this->vx = 0;
    this->vy = 0;
    this->offsetX = 0;
    this->offsetY = 0;
}

// la relation fondamentale de la dynamique s'applique
// ici très simplement :-)

void Tiling::left() {
    this->ax = -PULSE;
}

void Tiling::right() {
    this->ax = PULSE;
}

void Tiling::up() {
    this->ay = -PULSE;
}

void Tiling::down() {
    this->ay = PULSE;
}

// le raccordement à la boucle de contrôle
// est l'occasion d'appliquer nos équations
// de la dynamique du mouvement

void Tiling::tick() {
    // le vecteur vitesse est directement dérivé
    // du vecteur accélération
    this->vx += this->ax;
    this->vy += this->ay;

    // le vecteur de déplacement est directement dérivé
    // du vecteur vitesse
    this->offsetX += this->vx;
    this->offsetY += this->vy;

    // ! attention ici !
    // n'oubliez pas que l'accélération est une grandeur INSTANTANÉE
    // donc une fois qu'elle a été appliquée au vecteur vitesse,
    // vous devez impérativement la remettre à zéro !
    this->ax = 0;
    this->ay = 0;
}
```

Ok ! Allez-y ! Compilez… déversez le code machine sur la console pour l'exécuter… et testez !… Que remarquez-vous ? Le mouvement de la bille ne vous paraît pas bizarre ? Testez bien… par exemple donnez de petites impulsions toujours dans la même direction… vous voyez que la vitesse de la bille augmente et ne fait qu'augmenter, sauf si vous lui appliquez une force contraire en lui donnant une impulsion dans le sens opposé, ce qui aura pour effet de la freiner… jusqu'à inverser son mouvement. Et ensuite il ne cessera d'augmenter à nouveau dans cette direction.
Que se passe-t-il ? Et bien tout se déroule normalement ! Si, si !… Mais je ne vous ai pas encore parlé d'un autre principe fondamental de la physique Newtonienne : l'**inertie**.

Principe d'inertie

En physique, l'inertie d'un corps, dans un référentiel galiléen (dit inertiel), est sa tendance à conserver sa vitesse : en l'absence d'influence extérieure, tout corps ponctuel perdure dans un mouvement rectiligne uniforme. L'inertie est aussi appelée principe d'inertie, ou loi d'inertie, et, depuis Newton, première loi de Newton.

La loi d'inertie exprime le fait que si la vitesse $\rightarrow v \rightarrow$ *du corps ponctuel par rapport au repère galiléen est constante, « la somme des forces* $\rightarrow F_i F_i \rightarrow$ *s'exerçant sur le corps est nulle » :*

$$\rightarrow v = cte \Leftrightarrow \sum_i \rightarrow F_i = \rightarrow 0 v \rightarrow = cte \Leftrightarrow \sum iFi \rightarrow = 0 \rightarrow$$

Vous saisissez maintenant ? Voilà la raison pour laquelle la vitesse de la bille ne ralentit jamais (sauf si on lui applique une force opposée à la direction de son mouvement). C'est tout simplement le principe d'inertie qui est à l'œuvre dans notre modélisation. Aussi, pour ajouter plus de réalisme à notre scène de jeu, nous pourrions considérer que le dallage sur lequel se déplace la bille induit des frictions (des forces de frottement) sur la bille. En physique, le frottement (ou friction) est une interaction qui s'oppose au mouvement relatif entre deux systèmes en contact. Autrement dit, le dallage (qui est fixe) interagit avec la bille de manière à s'opposer à son mouvement. Ce qui veut tout simplement dire que cette interaction peut être grossièrement modélisée par un affaiblissement du vecteur vitesse de la bille. Si on considère maintenant que la bille est fixe (puisqu'elle est suivie par la caméra), c'est donc le dallage qui se déplace par-rapport à elle. Par conséquent, les frictions entraînent un affaiblissement de la vitesse du dallage par-rapport à la bille.
Comment pouvons-nous affaiblir simplement cette vitesse ? Et bien il suffit de la multiplier (à chaque cycle de la boucle de contrôle) par un facteur plus petit que `1`. Et oui… tout simplement. Et plus ce facteur sera proche de zéro, plus le ralentissement sera brutal. Nous déciderons (arbitrairement) ici d'affecter la valeur `0.9` à notre facteur de friction.

Intégrons maintenant ce nouveau phénomène dans nos équations :

Tiling.cpp

```
void Tiling::tick() {
    this->vx += this->ax;
    this->vy += this->ay;
    this->offsetX += this->vx;
    this->offsetY += this->vy;
    this->ax = 0;
    this->ay = 0;

    // application du phénomène de friction :
    this->vx *= .9;
    this->vy *= .9;

    // et nous pouvons également décider que lorsque les composantes
    // de la vitesse deviennent inférieures à 0.5 (en valeurs absolues),
    // alors on peut considérer qu'elles s'annullent tout simplement
    if (this->vx > -.5 && this->vx < .5) { this->vx = 0; }
    if (this->vy > -.5 && this->vy < .5) { this->vy = 0; }
}
```

Et voilà… vous pouvez maintenant compiler et tester l'application… vous voyez que c'est quand même vachement mieux ! On obtient désormais un mouvement beaucoup plus réaliste, n'est-ce pas ?

Même si nous avons abordé quelques principes théoriques physiques et mathématiques qui vous permettent maintenant de comprendre POURQUOI et COMMENT mettre en œuvre un mouvement réaliste dans vos applications, vous voyez que l'application de ces principes dans le code n'est pas si compliquée.

Allez, on va ENFIN pouvoir passer à la technique tant attendue qui consiste à appliquer des effets de *shading* sur vos scènes de jeu. Direction le prochain chapitre…

Eclairage tamisé

Et la lumière fut

Nous voilà enfin dans le vif du sujet de ce tutoriel. Notre objectif, au travers de ce chapitre sera de plonger la scène de jeu dans la pénombre et d'éclairer la bille par un halo de lumière pour ne distinguer que son voisinage proche. Cet effet d'éclairage peut donner une ambiance très sympa à vos jeux. Ce qui est remarquable ici, c'est que nous disposons de toute la palette de couleurs offerte par le référentiel `RGB565`. Il est clair que si nous ne disposions que des 16 couleurs proposées par le mode d'affichage `DISPLAY_MODE_INDEX`prévu par défaut lorsque l'on veut développer une application en haute résolution avec la META, nous serions très frustrés, car un tel effet n'aurait certainement pas le même rendu. Il pourrait même être impossible à mettre en oeuvre si vos sprites mobilisent déjà ne serait-ce que quelques couleurs…

La problématique qui s'impose à nous ici est de trouver le moyen de créer de l'ombre en partant d'une scène pleinement éclairée comme celle que nous avons développée jusqu'ici. La question est donc « comment *faire* de l'ombre avec de simples pixels » ? Et bien il suffit de les « assombrir » ! C'est-à-dire de jouer sur le niveau de luminosité de la couleur du pixel.
Le modèle colorimétrique que nous avons à notre disposition est le modèle `RGB565`. Autrement dit, les couleurs sont fabriquées à partir de trois couleurs primaires : le rouge (sur 5 bits), le vert (sur 6 bits) et le bleu (sur 5 bits), dont on peut fixer les niveaux d'intensité indépendamment les uns des autres. Si je vous donne par exemple la couleur `ORANGE` définie dans la palette officielle de la bibliothèque et que je vous demande de l'assombrir en diminuant sa luminosité de `50%`… Comment faire ? Regardons d'un peu plus près le codage de cette couleur dans les deux référentiels `RGB888` et `RGB565` :

Couleur	R8	G8	B8	RGB888	RGB565	R5	G6	B5
ORANGE	255	168	17	`#FFA811`	`0xFD42`	31	42	2

Diminuer la luminosité de cette couleur de `50%` revient simplement à diviser par `2` chacun des niveaux d'intensité de ses 3 couleurs primaires (on arrondira à l'entier le plus proche lorsque la division n'est pas entière) :

Couleur	R8	G8	B8	RGB888	RGB565	R5	G6	B5
ORANGE	255	168	17	`#FFA811`	`0xFD42`	31	42	2
ORANGE sombre	128	84	9	`#805409`	`0x82A1`	16	21	1

On obtient ainsi 2 niveaux de luminosité régulièrement répartis entre la teinte de référence et la couleur noire (où la luminosité est nulle).

En effet, si l'on soustrait à la teinte intermédiaire l'équivalent de 50% de la luminosité de la teinte de référence, on retombe bien sur le noir.

De manière générale, pour obtenir n niveaux de luminosité $\{R_i,G_i,B_i\}$ pour $i \in [1,n]$ d'une teinte de référence $\{R,G,B\}$, on peut définir les relations suivantes :

$$\forall i \in [1,n] \begin{cases} R_i = \frac{n-i+1}{n} R \\ G_i = \frac{n-i+1}{n} G \\ B_i = \frac{n-i+1}{n} B \end{cases}$$

Vous avez compris le principe ? Ok, donc si l'on souhaite maintenant générer 8 niveaux de luminosité, il suffit d'appliquer ces jolies petites formules pour obtenir rapidement la palette suivante (en incluant le noir comme 9ème couleur) :

Je vous laisse faire le calcul

Vous voyez que c'est relativement simple, n'est-ce pas ? Bien… Comment allons-nous appliquer ce principe à notre scène de jeu, de manière à créer ce halo de lumière autour de notre petite bille ?

Et bien, il suffira simplement d'aller piocher le code couleur du pixel dans la spritesheet du dallage, exactement comme nous l'avons fait jusqu'ici, et en fonction de la distance qui sépare ce pixel du centre de la bille, c'est-à-dire du centre de l'écran, nous lui affecterons un niveau de luminosité. C'est aussi simple que ça.

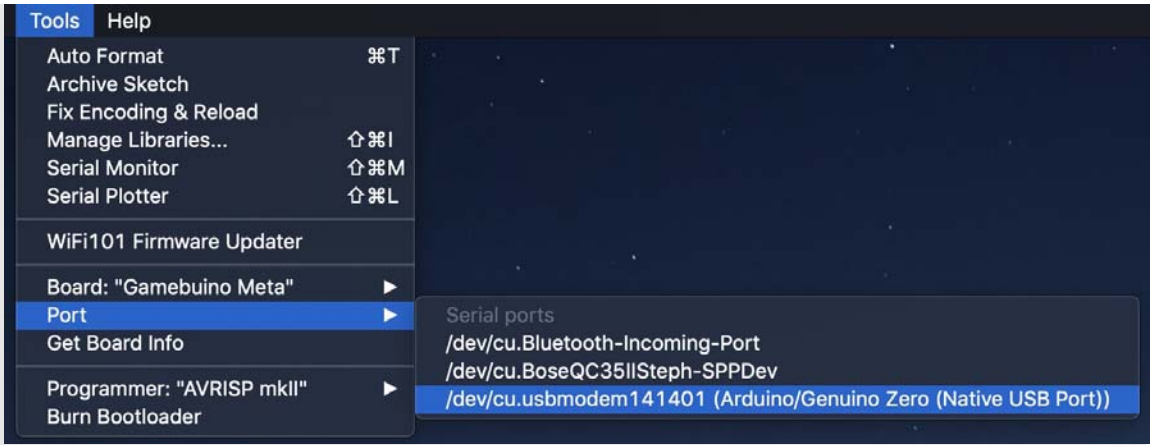
Préparatifs pour la mesure de performances

Mais auparavant, nous allons ajouter quelques lignes dans le croquis de notre application de manière à mesurer la charge du CPU et l'espace mémoire disponible, pour observer l'impact du calcul de luminosité sur les performances à l'exécution.

ShadingEffect.ino

```
#include <Gamebuino-Meta.h>
#include "GameEngine.h"

void setup() {
    gb.begin();

    // initialisation du port série
    SerialUSB.begin(9600);

    gb.display.init(0, 0, ColorMode::rgb565);
    GameEngine::init();
}

void loop() {
    while(!gb.update());

    // mesure de la charge du CPU à chaque seconde (on est à 25 fps par défaut)
    // et envoi des données vers le port série
    if (gb.frameCount % 25 == 0) {
        SerialUSB.printf("CPU: %i, RAM: %i\n", gb.getCpuLoad(), gb.getFreeRam());
    }

    GameEngine::tick();
}
```

Connectez votre META sur un port USB de votre ordinateur, puis ouvrez le port série dans **Arduino IDE** (notez que l'identifiant du port sera probablement différent chez vous) :



Puis lancez l'exécution pour observer comment se déroulent les choses tant que nous n'avons rien fait concernant le calcul de luminosité… Vous devriez obtenir quelque-chose dans ce goût-là :

```
[Starting] Opening the serial port - /dev/cu.usbmodem141401

[Info] Opened the serial port - /dev/cu.usbmodem141401

CPU: 82, RAM: 9755

CPU: 82, RAM: 9755

CPU: 82, RAM: 9755
```

Le CPU est donc déjà à `82%` de charge ! Forcément… on dessine énormément de pixels… et pour chaque pixel, on effectue pas mal de calculs finalement. Pour mémoire, voici le code de notre méthode `draw()` débarassé de tous ses commentaires :

Tiling.cpp

```
void Tiling::draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) {
    bool txodd,tyodd,jump;
    uint8_t sy,x,y, xo,yo;
    uint16_t syw,index,value,index_y;
    uint16_t nfo = TILE_WIDTH * TILE_HEIGHT;
    for (sy = 0; sy < sliceHeight; sy++) {
        syw = sy * SCREEN_WIDTH;
        y = sliceY + sy;
        yo = y + this->offsetY;
        tyodd = (yo / TILE_HEIGHT) % 2;
        index_y = (yo % TILE_HEIGHT) * TILE_WIDTH;
        for (x = 0; x < SCREEN_WIDTH; x++) {
            xo = x + this->offsetX;
            txodd = (xo / TILE_WIDTH) % 2;
            jump = txodd ^ tyodd;
            index = index_y + (xo % TILE_WIDTH) + (jump * nfo);
            value = BITMAP[index];
            buffer[x + syw] = value != TRANSPARENT_COLOR ? value : 0;
        }
    }
}
```

Première mise en œuvre

Commençons par déclarer deux constantes dans `Tiling.h` :

Tiling.h

```
class Tiling
{
    private:

        // le carré du rayon du halo
        static const uint16_t HALO_RADIUS2;

        // le nombre de niveaux de luminosité
        static const uint8_t BRIGHTNESS_LEVELS;

    // le reste des déclarations reste inchangé
};
```

Vous voyez que nous allons caractériser le halo de lumière par **le carré** de son rayon. En effet, dans la mesure où nous allons devoir calculer des distances, pour éviter de surcharger inutilement le CPU nous travaillerons avec le carré de ces distances, de manière à éviter d'avoir à utiliser la fonction `sqrt()` qui permet de calculer la racine carrée d'un nombre, mais qui ferait s'effondrer les performances. Nous préférerons travailler directement sur les carrés.

Définissons maintenant ces deux constantes :

Tiling.cpp

```
const uint16_t Tiling::HALO_RADIUS2 = 0b1<<11;
const uint8_t Tiling::BRIGHTNESS_LEVELS = 4;
```

Vous remarquerez, que pour chacune d'entre elles, nous avons choisi une valeur qui est une puissance de 2 : `0b1<<11` = 2₁₁211 = `2048`, autrement dit, le rayon du halo s'étendra sur √2048 2048 = environ `45` pixels, et nous souhaitons afficher ici `4` = 2₂22 niveaux de luminosité.
Voyons maintenant comment modifier notre méthode `draw()` pour qu'elle prenne en compte ces niveaux de luminosité dans son calcul de rendu :

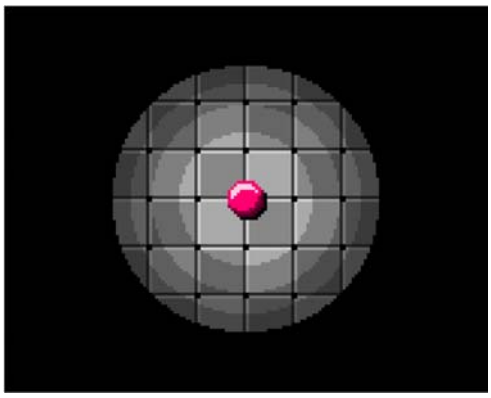
Tiling.cpp

```
void Tiling::draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) {
    bool txodd,tyodd,jump;
    uint16_t nfo = TILE_WIDTH * TILE_HEIGHT;
    uint8_t sy,x,y;
    uint8_t xo,yo;
    uint16_t syw;
    uint16_t index, value;
    uint16_t index_y;

    // nous aurons besoin de mesurer le carré de la distance `r2`
    // entre le pixel du dallage et le centre de l'écran...
    // un calcul intermédiaire permettra de déterminer
    // la composante Y de `r2` que nous nommerons `ry2`
    uint16_t r2,ry2;
    uint8_t hsw = SCREEN_WIDTH / 2;
    uint8_t hsh = SCREEN_HEIGHT / 2;

    // pour effectuer les calculs de luminosité, nous aurons
    // besoin d'isoler les composantes rouge, verte et bleue
    // du code couleur RGB565 du pixel à traiter
    uint16_t red, green, blue;

    // nous calculerons alors le niveau de luminosité du pixel
    // en fonction du rapport entre le carré de la distance `r2`
    // qui le sépare du centre de l'écran et le carré du rayon
    // du halo de lumière `HALO_RADIUS2`
    uint16_t lux;

    // cette variable nous permettra d'effectuer les calculs
    // intermédiaires sur la couleur à transformer pour qu'elle
```

```
    // intègre les données de luminosité
    uint16_t color;

    for (sy = 0; sy < sliceHeight; sy++) {
        syw = sy * SCREEN_WIDTH;
        y = sliceY + sy;
        yo = y + this->offsetY;
        tyodd = (yo / TILE_HEIGHT) % 2;
        index_y = (yo % TILE_HEIGHT) * TILE_WIDTH;

        // on calcule la composante Y du carré de la distance
        // qui sépare le pixel du centre de l'écran
        ry2 = (y - hsh) * (y - hsh);

        for (x = 0; x < SCREEN_WIDTH; x++) {
            xo = x + this->offsetX;
            txodd = (xo / TILE_WIDTH) % 2;
            jump = txodd ^ tyodd;
            index = index_y + (xo % TILE_WIDTH) + (jump * nfo);

            // on calcule le carré de la distance qui sépare
            // le pixel du centre de l'écran
            r2 = (x - hsw) * (x - hsw) + ry2;

            // on plonge la dalle dans le noir par défaut
            value = 0;

            // et lorsque l'on est à l'intérieur du halo
            if (r2 <= HALO_RADIUS2) {
                // on récupère le code couleur du sprite de la dalle
                color = BITMAP[index];

                // s'il ne s'agit pas de la couleur de transparence
                if (color != TRANSPARENT_COLOR) {
                    // on calcule le niveau de luminosité à cette distance
                    lux = BRIGHTNESS_LEVELS - BRIGHTNESS_LEVELS * r2 / HALO_RADIUS2;

                    // on réécrit le code couleur en big-endian
                    color = ((color & 0xff) << 8) | (color >> 8);

                    // puis on récupère les niveaux d'intensité
                    // des 3 couleurs primaires
                    red   = color >> 11;
                    green = (color >> 5) & 0b111111;
                    blue  = color & 0b11111;

                    // on applique le niveau de luminosité
                    // à chacune des couleurs primaires
                    red   = red   * lux / BRIGHTNESS_LEVELS;
                    green = green * lux / BRIGHTNESS_LEVELS;
                    blue  = blue  * lux / BRIGHTNESS_LEVELS;

                    // puis on recompose le code couleur RGB565
                    value = (red << 11) | (green << 5) | blue;

                    // et on le réécrit en little-endian
                    value = ((value & 0xff) << 8) | (value >> 8);
                }
            }

            // il ne reste plus qu'à recopier
            // le code couleur dans le tampon
            buffer[x + syw] = value;
        }
    }
}
```

Allez-y, vous pouvez compiler et tester !

Vous devriez obtenir un truc dans ce goût là :



C'est sûr qu'avec seulement 4 niveaux de luminosité, c'est assez moche… mais ça permet de bien visualiser le découpage quadratique des couronnes d'ombre : la luminosité y est inversement proportionnelle aux carrés de leurs rayons. Voyons un peu ce que donnent les mesures de performances :

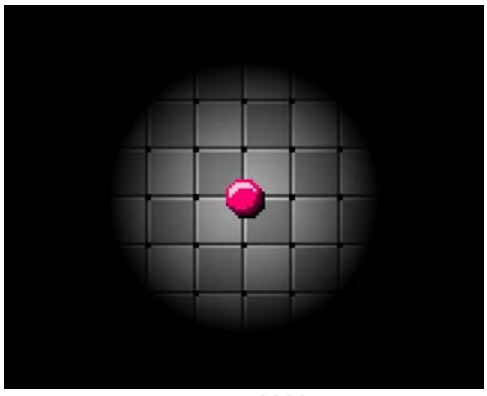
```
[Starting] Opening the serial port - /dev/cu.usbmodem141401
[Info] Opened the serial port - /dev/cu.usbmodem141401
CPU: 98, RAM: 9755
CPU: 98, RAM: 9755
CPU: 98, RAM: 9755
```

Ouch ! Vous voyez que ça a un coût ! Le processeur est déjà presque à donf !… Allez, même pas peur… passons à 32 niveaux de luminosité pour plus de finesse dans le rendu :

Tiling.cpp

```
const uint8_t Tiling::BRIGHTNESS_LEVELS = 0b1<<5; // 2^5 = 32
```

Recompilez et testez !



Woaaaaahhhh...

Vous remarquerez que les performances restent les mêmes. Merci les puissances de `2` ! Tiens d'ailleurs, essayez de fixer une valeur qui n'est pas une puissance de `2` à la constante `BRIGHTNESS_LEVELS`, par exmple la valeur `25`… vous allez voir l'impact sur les performances ! Ne vous fiez pas trop à ce qui est envoyé au port série… cet affichage n'a lieu qu'en dehors de la méthode `draw()`… donc si la charge du CPU est trop importante, les données seront envoyées dès qu'il sortira de la méthode `draw()` et qu'il pourra souffler un peu. Du coup les valeurs affichées ne rendent pas vraiment compte de la charge réelle… Par contre, essayez de déplacer votre bille… vous vous rendrez très vite compte du réel impact sur les performances.

Vous voyez donc que travailler avec des puissances de 2peut largement optimiser un calcul. Et c'est tout à fait logique, puisque dans ce cas, les opérations arithmétiques peuvent souvent être simplifiées par de simples décalages de bits. Mettez-ça dans une de vos cases mémoires.

On pourrait encore optimiser les choses. Par exemple, dans la mesure où le carré du rayon du halo de lumière est une puissance de 2 :

Tiling.cpp

```
// on pourrait remplacer le test suivant
if (r2 <= HALO_RADIUS2) ...

// par celui-ci
if (!(r2 >> 11)) ...
```

On pourrait également réécrire les codes couleurs en **big-endian** dans la constante `BITMAP` : ça nous éviterait 1 renversement (mais pas l'autre !)… par contre, on ne gagnerait presque rien sur cette opération (qui est très rapide). Et c'est de toutes façons inutile à ce stade, puisque je vais maintenant vous proposer une autre manière d'optimiser largement les choses…

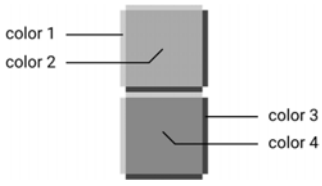
Une nouvelle idée ?

Ce qui plombe les performances ici est bien entendu le calcul sur la luminosité, qui intervient sur chaque pixel. Comment pourrions-nous alléger la charge du processeur ? Et bien tout simplement en ayant effectué les calculs au préalable, à sa place. Mais comment ? Et bien c'est tout simple : il suffit de lui fournir une spritesheet qui intègre tous les codes couleurs précalculés avec tous les niveaux de luminosité que l'on souhaite afficher :

Il suffira alors d'aller piocher le pixel dans la dalle approprié. Cette solution n'est pas mal du tout… mais elle a trois inconvénients majeurs ! S'il vous prenait l'envie de modifier le nombre de niveaux de luminosité, il vous faudrait alors entièrement redessiner la spritesheet, ce qui peut s'avérer laborieux à la longue… De même que si vous souhaitez afficher de nombreux niveaux de luminosité, la conception de la spritesheet peut là aussi s'avérer laborieux… Et de surcroît, plus vous aurez de dalles à y intégrer, plus elles occuperont de l'espace en mémoire ! Vous voyez que ça n'est pas la meilleure solution tout compte fait.

La bonne solution

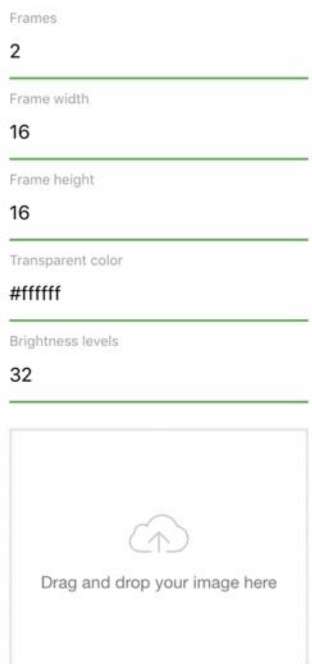
Mais il existe une autre solution. En effet, il suffit d'indexer les couleurs utilisées sur la spritesheet `tile.png` telle que je vous l'ai donnée (il y en 4). Puis de reconstruire la constante `BITMAP` en utilisant ces index plutôt que les codes couleurs `RGB565`.



Il ne restera plus ensuite qu'à construire une palette de couleurs `COLORMAP` présentant toutes les déclinaisons des couleurs de référence que nous avons indexées, en fonction des niveaux de luminosité souhaités… et à écrire les codes couleurs en **little-endian**. Bon… ça fait quand même un peu de boulot…

Mais j'ai pensé à vous : je vais publier, en même temps que ce tuto, un petit outil qui va faire tous les calculs pour vous et vous fournir *in fine* les constantes `BITMAP` et `COLORMAP` correspondantes.
Je vous mettrai le lien de cette création ici dès que je l'aurai publiée…

Dès que l'outil sera disponible, saisissez le paramétrage suivant, puis faîtes glisser l'image `tiles.png` sur l'encart **Drag and drop your image here** :



Vous devriez obtenir le code suivant :

```
// reminder for decoding:
//
// number of frames: 2
// frame width: 16
// frame height: 16
// transparent color: 0xffff
// brightness levels: 32

const uint8_t BITMAP[] = {
    // frame #1
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0,
    // frame #2
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
```

```
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0
};

const uint16_t COLORMAP[] = {
    // color #0 (transparent color)
    0xffff,
    // color #1
    0x79ce, 0x38c6, 0xf7bd, 0xd7bd, 0x96b5, 0x75ad, 0x34a5, 0xf39c, 0xd39c, 0x9294,
0x718c, 0x3084, 0x1084, 0xcf7b, 0x8e73, 0x6d6b, 0x2c63, 0x0c63, 0xcb5a, 0x8a52,
0x694a, 0x2842, 0x0842, 0xc739, 0x8631, 0x6529, 0x2421, 0x0421, 0xc318, 0x8210,
0x6108, 0x2000,
    // color #2
    0x55ad, 0x34a5, 0xf39c, 0xd39c, 0xb294, 0x718c, 0x518c, 0x3084, 0x1084, 0xcf7b,
0xae73, 0x8e73, 0x4d6b, 0x2c63, 0x0c63, 0xcb5a, 0xaa52, 0x8a52, 0x494a, 0x2842,
0x0842, 0xc739, 0xa631, 0x8631, 0x4529, 0x2421, 0x0421, 0xc318, 0xa210, 0x8210,
0x4108, 0x2000,
    // color #3
    0x2842, 0x0842, 0x0842, 0xe739, 0xe739, 0xc739, 0xa631, 0xa631, 0x8631, 0x8631,
0x6529, 0x6529, 0x4529, 0x4529, 0x2421, 0x2421, 0x0421, 0x0421, 0xe318, 0xe318,
0xc318, 0xa210, 0xa210, 0x8210, 0x8210, 0x6108, 0x6108, 0x4108, 0x4108, 0x2000,
0x2000, 0x0000,
    // color #4
    0x518c, 0x3084, 0x1084, 0xcf7b, 0xae73, 0x8e73, 0x6d6b, 0x4d6b, 0x2c63, 0x0c63,
0xeb5a, 0xcb5a, 0xaa52, 0x8a52, 0x694a, 0x494a, 0x2842, 0x0842, 0xe739, 0xa631,
0x8631, 0x6529, 0x4529, 0x2421, 0x0421, 0xe318, 0xc318, 0xa210, 0x8210, 0x6108,
0x4108, 0x2000
};
```

Les données générées permettent d’économiser beaucoup d’espace mémoire par-rapport à la solution précédente (qui consistait à redessiner tous les sprites avec différents niveaux de luminosité). Mais voyons tout de suite de quelle manière…

- mettre en œuvre ce nouveau codage de la spritesheet,
- et optimiser encore davantage notre méthode de calcul de rendu.

Reprenons la déclaration intégrale de notre classe `Tiling` pour mieux voir les changements que nous allons y apporter :

Tiling.h

```
#ifndef SHADING_EFFECT_TILING
#define SHADING_EFFECT_TILING

#include "Renderable.h"

// l'accélération transférée à chaque impulsion motrice
#define PULSE 1

// nous avons vu que les calculs de distance
// pouvaient être effectués rapidement en utilisant
// le carré du rayon du halo de lumière simplement
// exprimé par une puissance de 2
#define HALO_RADIUS2_POWER_OF_TWO 11

// et nous pouvons faire de même avec
// les niveaux de luminosité
#define BRIGHTNESS_LEVELS_POWER_OF_TWO 5

class Tiling : public Renderable
{
    private:

        static const uint8_t TILE_WIDTH;
        static const uint8_t TILE_HEIGHT;
```

```
        // notez le changement de type des valeurs
        // désormais stockées dans la spritesheet
        static const uint8_t BITMAP[];

        // la palette de couleurs associée à la spritesheet
        // incluant toutes les déclinaisons lumineuses des
        // couleurs de référence de nos sprites
        static const uint16_t COLORMAP[];

        // les propriétés cinématiques
        float ax,ay;
        float vx,vy;
        int8_t offsetX,offsetY;

    public:

        Tiling();

        // les commandes de mouvement
        void left();
        void right();
        void up();
        void down();

        // le point de raccordement de la boucle de contrôle
        void tick();

        // la méthode de calcul du rendu
        void draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) override;
};

#endif
```

Voyons maintenant les définitions des variables statiques de notre classe `Tiling` ainsi que la nouvelle définition de la méthode `draw()` :

Tiling.cpp

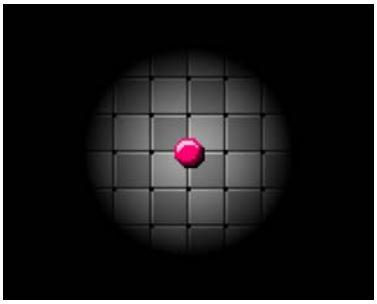
```
// les paramètres descriptifs de nos sprites (les dalles)
const uint8_t Tiling::TILE_WIDTH = 16;
const uint8_t Tiling::TILE_HEIGHT = 16;

// la nouvelle spritesheet obtenue avec l'outil de transcodage
const uint8_t Tiling::BITMAP[] = {
    // la dalle claire
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0,
    // la dalle sombre
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
```

```
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
    0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0
};

// puis la palette de couleurs que nous a fournie l'outil de transcodage
// avec leurs 32 niveaux de luminosité
const uint16_t Tiling::COLORMAP[] = {
    // couleur #0 (couleur de transparence)
    0xffff,
    // couleur #1
    0x79ce, 0x38c6, 0xf7bd, 0xd7bd, 0x96b5, 0x75ad, 0x34a5, 0xf39c, 0xd39c, 0x9294,
0x718c, 0x3084, 0x1084, 0xcf7b, 0x8e73, 0x6d6b, 0x2c63, 0x0c63, 0xcb5a, 0x8a52,
0x694a, 0x2842, 0x0842, 0xc739, 0x8631, 0x6529, 0x2421, 0x0421, 0xc318, 0x8210,
0x6108, 0x2000,
    // couleur #2
    0x55ad, 0x34a5, 0xf39c, 0xd39c, 0xb294, 0x718c, 0x518c, 0x3084, 0x1084, 0xcf7b,
0xae73, 0x8e73, 0x4d6b, 0x2c63, 0x0c63, 0xcb5a, 0xaa52, 0x8a52, 0x494a, 0x2842,
0x0842, 0xc739, 0xa631, 0x8631, 0x4529, 0x2421, 0x0421, 0xc318, 0xa210, 0x8210,
0x4108, 0x2000,
    // couleur #3
    0x2842, 0x0842, 0x0842, 0xe739, 0xe739, 0xc739, 0xa631, 0xa631, 0x8631, 0x8631,
0x6529, 0x6529, 0x4529, 0x4529, 0x2421, 0x2421, 0x0421, 0x0421, 0xe318, 0xe318,
0xc318, 0xa210, 0xa210, 0x8210, 0x8210, 0x6108, 0x6108, 0x4108, 0x4108, 0x2000,
0x2000, 0x0000,
    // couleur #4
    0x518c, 0x3084, 0x1084, 0xcf7b, 0xae73, 0x8e73, 0x6d6b, 0x4d6b, 0x2c63, 0x0c63,
0xeb5a, 0xcb5a, 0xaa52, 0x8a52, 0x694a, 0x494a, 0x2842, 0x0842, 0xe739, 0xa631,
0x8631, 0x6529, 0x4529, 0x2421, 0x0421, 0xe318, 0xc318, 0xa210, 0x8210, 0x6108,
0x4108, 0x2000
};

// vous allez voir que nous allons optimiser certaines choses et que,
// finalement, si vous virez tous les commentaires, cette méthode est
// somme toute assez concise :-)
void Tiling::draw(uint8_t sliceY, uint8_t sliceHeight, uint16_t* buffer) {
    bool txodd,tyodd;
    bool jump;
    uint16_t nfo = TILE_WIDTH * TILE_HEIGHT;
    uint8_t sy,x,y;
    uint8_t xo,yo;
    uint16_t syw;
    uint16_t index, index_y, value;

    // nous aurons besoin de mesurer le carré de la distance `r2`
    // entre le pixel du dallage et le centre de l'écran...
    // un calcul intermédiaire permettra de déterminer
    // la composante Y de `r2` que nous nommerons `ry2`
    uint16_t r2,ry2;
    uint8_t hsw = SCREEN_WIDTH / 2;  // le centre
    uint8_t hsh = SCREEN_HEIGHT / 2; // de l'écran

    // nous calculerons alors le niveau de luminosité du pixel
    // en fonction du rapport entre le carré de la distance `r2`
    // qui le sépare du centre de l'écran et le carré du rayon
    // du halo de lumière, défini sous la forme d'une
    // puissance de 2 : `HALO_RADIUS2_POWER_OF_TWO`
    uint8_t lux;

    // cette variable nous permettra de stocker l'index
    // de la couleur récupérée dans la spritesheet et
    // correspondant à l'index de la couleur de référence
    // dans la palette `COLORMAP`
    uint8_t colorIndex;

    // on plonge toute la scène dans le noir en remplissant la tranche de zéros
    // > nous utilisons des pointeurs ici pour accélérer le traitement...
    // on pourrait également déplacer cette procédure
    // dans la méthode `draw()` du `Renderer`
    uint32_t* tmp = (uint32_t*)buffer;
    uint16_t bs = (sliceHeight * SCREEN_WIDTH) >> 1;
    while (bs--) *tmp++ = 0;

    // allez... c'est parti !
    for (sy = 0; sy < sliceHeight; sy++) {
        syw = sy * SCREEN_WIDTH;
```

```
        y = sliceY + sy;
        yo = y + this->offsetY;
        tyodd = (yo / TILE_HEIGHT) % 2;
        index_y = (yo % TILE_HEIGHT) * TILE_WIDTH;

        // on calcule la composante Y du carré de la distance
        // qui sépare le pixel du centre de l'écran
        ry2 = (y - hsh) * (y - hsh);

        for (x = 0; x < SCREEN_WIDTH; x++) {
            xo = x + this->offsetX;
            txodd = (xo / TILE_WIDTH) % 2;
            jump = txodd ^ tyodd;
            index = index_y + (xo % TILE_WIDTH) + (jump * nfo);

            // on calcule le carré de la distance qui sépare
            // le pixel du centre de l'écran
            r2 = (x - hsw) * (x - hsw) + ry2;

            // si le pixel est à l'intérieur du halo de lumière
            if (!(r2 >> HALO_RADIUS2_POWER_OF_TWO)) {
                // on récupère le code couleur du pixel de la dalle
                // dans notre spritesheet
                colorIndex = BITMAP[index];

                // et s'il ne s'agit pas de la couleur de transparence
                if (colorIndex--) {
                    // remarque sur le `--`
                    // si colorIndex n'est pas l'index la couleur de transparence
                    // alors colorIndex > 0 ... donc on va décrémenter sa valeur
                    // immédiatement après l'avoir lue pour simplifier le calcul
                    // qui suit, qui consistera à piocher le code couleur dans
                    // la palette des couleurs `COLORMAP`
                    //
                    // on calcule le niveau de luminosité à cette distance
                    lux    =    (r2    <<    BRIGHTNESS_LEVELS_POWER_OF_TWO)    >>
HALO_RADIUS2_POWER_OF_TWO;

                    // on va ensuite piocher le code couleur correspondant
                    // dans la palette de couleurs...
                    // > et on n'oublie pas d'appliquer le décalage correspondant
                    // > à la couleur de transparence, qui est le premier élément
                    // > de la palette
                    // et on termine en écrivant cette valeur dans le buffer
                    buffer[x    +    syw]    =    COLORMAP[1    +    (colorIndex    <<
BRIGHTNESS_LEVELS_POWER_OF_TWO) + lux];
                }
            }
        }
    }
}

// et voilà le travail ;-)
```

Vous voyez ici que, dans la mesure où nous avons initialisé notre buffer avec des `0`, nous pouvons nous contenter de n'y écrire que lorsque nous sommes à l'intérieur du halo de lumière, et que la couleur à reporter n'est pas la couleur de transparence du sprite. Ceci nous fait gagner énormément de temps en diminuant les opérations de calcul et d'écriture dans le tampon !

Bien… nous pouvons maintenant compiler et lancer l'exécution :



Le rendu est parfait

Jetons maintenant un œil aux performances :

```
[Starting] Opening the serial port - /dev/cu.usbmodem141401
[Info] Opened the serial port - /dev/cu.usbmodem141401
CPU: 73, RAM: 9739
CPU: 73, RAM: 9739
CPU: 73, RAM: 9739
```

On a bien bossé : on a fait chuter la charge du CPU de `98%` à `73%`… Bravo !
Nous devrions pouvoir augmenter le framerate à `32` pour être à `100%` au niveau de la charge du CPU. Ce qui augmentera l'impression de fluidité au rendu.

ShadingEffect.ino --> setup()

```
gb.setFrameRate(32);
```

Et voilà ! Ce tutoriel est terminé. On a fait du bon boulot. Le rendu finalisé a quand même de la gueule, vous en conviendrez. Il me reste à ajouter le mot de la fin… donc direction le prochain et dernier chapitre…

Conclusion

Le mot de la fin

Nous arrivons au terme de ce tutoriel. J'espère que vous l'avez apprécié et que vous avez pu en retenir l'essentiel. N'hésitez pas à me communiquer vos commentaires, ou à me demander des précisions sur les éléments que vous n'avez pas bien compris, sur la page officielle de cette Création.
Je tiens tout particulièrement à adresser à Andy (aoneill) mes plus chaleureux remerciements pour ses contributions majeures sur la planète Gamebuino, et en particulier les clefs magiques qu'il nous a confiées pour ouvrir les portes de la haute résolution en `RGB565`. Il devient désormais possible de faire de très belles choses sur cette petite console, en tirant parti de toutes ses capacités.

Un énorme merci Andy !

Je tiens à remercier également toute la communauté qui s'active autour du projet d'Aurélien pour en faire la promotion, pour livrer ses contributions en Open Source et apporter sa pierre à l'édifice, de façon à bâtir petit à petit un formidable outil pédagogique pour nos jeunes codeurs en herbe, et pour nous tous finalement, puisque chacun d'entre nous continue d'apprendre à la lumière des créations qui foisonnent sur le site gamebuino.com.

Bravo à Aurélien et à toute l'équipe pour avoir su conduire ce projet et créer cette communauté formidable.