Module RAD
Qt / QML : Pow211

Nom :

Prénom :

Pour ces travaux, vous devez utiliser une machine équipée du Qt Framework version 5.9.x (LTS) ou supérieure ; l'usage de l'EDI Qt Creator est recommandé.

1. Cahier des Charges

L'objectif de ces travaux est de réaliser une application exploitable aussi bien en version *desktop* qu'en version mobile (sous Android et iOS) ; basée sur des technologies mixtes langage déclaratif / langage C++.

Le thème retenu devant être à la fois simple d'un point de vue algorithmique et suffisamment étoffé pour montrer les particularités d'une application graphique multi-stations avec ou sans clavier, nous allons développer un logiciel *2048 like*, le fameux jeu qui consiste à additionner des tuiles (dont les valeurs sont des puissances de 2) pour atteindre 2048.

2048 se joue sur une grille de 4 x 4 cases, avec des tuiles de couleurs et de valeurs variées (mais toujours des puissances de deux) qui peuvent être déplacées dans les 4 directions – gauche, droite, haut, bas – au moyen des touches flèches du clavier ou par glissement d'un doigt sur une surface tactile.

8	32	64	512
4	8	16	256
2	4	8	32
		4	8

Si deux tuiles ayant le même nombre entrent en collision durant un mouvement, elles fusionnent en une nouvelle tuile de valeur double. À chaque mouvement, une tuile portant un 2 ou un 4 apparaît dans une case vide de manière aléatoire.

La partie commence avec 2 tuiles de valeur 2 ou 4 posées aléatoirement sur le plateau. La partie est gagnée lorsqu'une tuile portant la valeur 2048 apparaît sur la grille ; elle est perdue si les 16 cases du plateau sont occupées et que plus aucune fusion n'est possible. Le score atteint est tout simplement la somme des fusions réalisées.

Ce jeu a été créé en un week-end par Gabriele Cirulli, un développeur web italien autodidacte alors âgé de 19 ans… La licence du jeu est libre et *opensource*.

Q1. À votre avis, pourquoi ces travaux portent-ils le nom « Pow211 » ?

Le TP porte le nom "Pow211" puisque 2 porté à la puissance 11 donne 2048 (le jeu)

Supposez par exemple que les nouvelles tuiles soient toutes des « 2 », y compris les deux premières, et que le joueur arrive à faire systématiquement les fusions possibles à chaque tour…

Q2. Combien de tours sont nécessaires pour obtenir une première tuile de valeur → (la première tuile de valeur 8 est obtenue au 4e tour)

8	16	32	64
4			

Pour répondre à cette question, vous pouvez simuler un début de partie de la manière ci-contre (les numéros à gauche sont le nombre de tours, les valeurs soulignées représentent les tuiles entrantes) →

1	2 + 2 → 4 + 2
2	4 + 2 + 2
3	4 + 4 + 2
4	8 + 2 + 2
5	8 + 4 + 2
6	8 + 4 + 2 + 2
7	8 + 4 + 4 + 2
8	8 + 8 + 2 + 2
9	… …

Q3. Dans ces mêmes conditions, déduisez la loi qui relie le nombre de tours (N) et la puissance de 2 (n) de la tuile à atteindre :

N = ...

Q4. Combien faut-il alors de tours pour gagner la partie ? .....

2. Création du projet

Nous allons donc comme annoncé créer une application avec un cœur applicatif en C++ et une interface QML afin de voir les liaisons entre les deux. La mise au point de l'application sera faite en mode *desktop* sous Linux, Mac OSX ou Windows ; et nous verrons ensuite comment Qt Mobile vous facilite la vie pour porter le produit sur Android et/ou iOS (le prix à payer se résumant à une simple re-compilation !).

C'est parti, il est temps de démarrer Qt Creator…

☑ Créez un nouveau projet de type « Application Qt Quick » nommé `Pow211`.
Choisissez l'ensemble de composants « Qt Quick 2.2 » et un kit de développement « Desktop ».

Vous obtenez normalement un fichier *main.cpp* qui instancie une application QML et charge le fichier *main.qml* qui lui contient l'interface utilisateur.
Le fichier QML est très lisible et compréhensible : un rectangle est dessiné et contient le texte « Hello World ». Une zone de capture des évènements de la souris est placée, la fenêtre est fermée lorsqu'on clique dedans.

☑ Vérifiez cela en exécutant par exemple la version *release* du programme.

Il faut maintenant modéliser notre plateau de jeu :

☑ Ajoutez au projet une nouvelle classe `Pow211` dérivée de `QObject`.
Placez dans l'interface *pow211.h* deux constantes `BOARDSIZE` (= 4) et `MAXVALUEOFTILE` (= 2048).

Nous allons utiliser les types Qt `quint16` pour le tableau à 2 dimensions des tuiles et `quint32` pour le score. Un autre attribut sera utilisé pour stocker la valeur de la tuile la plus élevée présente sur le jeu.

Q5. Proposez des déclarations pour les membres privés `m_board`, `m_higherTile` et `m_score` :

```
quint16 m_board[BOARDSIZE*BOARDSIZE];
quint16 tile(int index ) const ;
quint32 m_score;
int m_higherTitle;
```

☑ Enrichissez l'interface publique de la classe `Pow211` avec les sélecteurs suivants (avec `index` compris entre `0` et `(BOARDSIZE * BOARDSIZE – 1)` :

```
quint16 tile(int index ) const ;  // lecture valeur de la tuile d'indice index
quint32 score() const ;           // lecture du score courant
```

Dans `m_board`, l'absence de tuile est matérialisée par une valeur nulle ; ce qui est à priori le cas pour toutes les cases du jeu avant le début de partie.
L'initialisation des membres de la classe est naturellement confiée au constructeur.

Q6. Proposez une implémentation de ces initialisations (pensez à la fonction standard `memset()`) :

```
Pow211::Pow211(QObject *parent)
    : QObject(parent)
, m_score(0), m_higherTitle(0){
 memset(m_board, BOARDSIZE*BOARDSIZE*sizeof(quint16),0);
}
}
```

Pour nos tests préliminaires, nous allons ajouter à notre classe une méthode publique permettant de visualiser le contenu de notre plateau de jeu sur la console texte (onglet « Sortie de l'application » de Qt Creator).

Q7. Qu'est-il nécessaire d'ajouter au fichier *pow211.cpp* pour que le code ci-contre se compile correctement ?

```
#include <iostream>
#include <iomanip>
```

```
1  void Pow211::print() const
2  {
3      cout << endl ;
4      for ( int y = 0 ; y < BOARDSIZE ; ++y ) {
5          for ( int x = 0 ; x < BOARDSIZE ; ++x ) {
6              cout << setw(5) << m_board[x][y] ;
7          }
8          cout << endl ;
9      }
10     cout << endl ;
11 }
```

Q8. Que pouvez-vous insérer dans *main.cpp*, et à quelle(s) ligne(s), pour créer et visualiser un plateau de jeu ?

```
engine.load(QUrl("../Pow211/main.qml"));
Pow211 pow;
pow.print();
```

.....
.....
.....
.....

```
1  #include <QGuiApplication>
2  #include <QQmlApplicationEngine>
3
4  int main(int argc, char *argv[])
5  {
6      QGuiApplication app(argc, argv);
7      QQmlApplicationEngine engine;
8      engine.load(QUrl(...));
9
10     return app.exec();
11 }
```

Toujours pour nos tests de développement, nous allons encore ajouter une méthode privée `setBoard()` qui nous permettra de forcer un remplissage arbitraire du plateau.

Ajoutez la méthode ci-contre →

```
1  void Pow211::setBoard()
2  {
3      quint16 board[] = {
4              0,    2,    4,    8,
5             16,   32,   64,  128,
6            256,  512, 1024, 2048,
7              0,    0,    0,    0
8      } ;
9
10     for ( int i = 0 ; i < BOARDSIZE * BOARDSIZE ; ++i ) {
11         m_board[ i % BOARDSIZE ][ i / BOARDSIZE ] = board[i] ;
12     }
13 }
```

Cette méthode peut être appelée dans le constructeur après l'initialisation du plateau.

3. Déplacement et fusion des tuiles

L'algorithme de déplacement et de fusion des tuiles est le cœur du jeu !
Intéressons-nous aux diverses situations possibles pour une ligne lors d'un déplacement vers la droite…

situation initiale					*décalage(s) et fusion(s)*					*déplacement(s)*			
		8	4	→			8	4	→			8	4
4		16		→		4		16	→			4	16
	2	2		→			4		→				4
8	8			→		16			→				16
4		4	8	→			8	8	→			8	8
4		2	2	→		4		4	→			4	4
2	2	4	4	→		4		8	→			4	8
4	4	4		→		4	8		→			4	8
8			8	→				16	→				16
4	4	4	4	→		8		8	→			8	8

On constate qu'il peut y avoir de 0 à 2 fusions sur la même ligne.

Supposons que l'on parcourt la ligne de droite à gauche ; pour chaque tuile, il faut commencer par la déplacer à droite tant que cela est possible (autrement dit tant que l'emplacement à sa droite est libre).
Ensuite, si une tuile et sa voisine de droite sont égales, il faut ranger la somme à droite et mettre à 0 la tuile en cours ; pour ensuite poursuivre le traitement avant celle mise à 0.
Pour finir, il suffit de justifier les tuiles restantes à droite.

La méthode privée ci-contre fait glisser la tuile d'abscisse x vers la droite si cela est possible →

ligne 6 : seul le déplacement d'une tuile non nulle retourne un résultat vrai.

```
1  bool Pow211::slideTile(int line, int x )
2  {
3      if ( m_board[x+1][line] == 0 ) {
4          m_board[x+1][line] = m_board[x][line] ;
5          m_board[x][line] = 0 ;
6          return ( m_board[x+1][line] != 0 ) ;
7      }
8      return false ;
9  }
```

Et cette autre méthode privée gère les déplacements et les fusions d'une ligne →

lignes 5 à 8 : les tuiles de la ligne sont glissées vers la droite.

lignes 10 à 21 : si deux tuiles adjacentes ont la même valeur non nulle, la fusion sur celle de droite est réalisée.

lignes 23 à 26 : une fois les fusions réalisées, les tuiles restantes sont à nouveau glissées vers la droite.

```
1  bool Pow211::slideLine(int line )
2  {
3      bool modified = false ;
4
5      for ( int x = BOARDSIZE - 2 ; x >= 0 ; x-- ) {
6          for ( int xr = x ; xr < BOARDSIZE - 1 ; xr++ )
7              modified |= slideTile(line, xr ) ;
8      }
9
10     int x = BOARDSIZE - 2 ;
11     while ( x >= 0 ) {
12         if ( ( m_board[x][line] )
13            &&( m_board[x][line] == m_board[x+1][line] ) ) {
14             m_board[x][line] = 0 ;
15             m_board[x+1][line] *= 2 ;
16
17             x-- ;
18             modified = true ;
19         }
20         x-- ;
21     }
22
23     for ( int x = BOARDSIZE - 2 ; x >= 0 ; x-- ) {
24         for ( int xr = x ; xr < BOARDSIZE - 1 ; xr++ )
25             slideTile(line, xr ) ;
26     }
27
28     return modified ;
29 }
```

Q9. Quelles instructions devez-vous insérer ligne 16 pour mettre à jour correctement les membres `m_score` et `m_higherTile` ?

m_score + = m_board[x+1][line];
if(m_higherTile < m_board[x+1][line]) m_higherTile = m_board[x+1][line];

Q10. Pourquoi avoir utilisé l'opérateur |= en ligne 7 ?

.....

Testez l'algorithme en ajoutant par exemple dans le constructeur un appel à `setBoard()` pour reproduire les diverses situations de la page précédente, et une commande de glissement à droite du style :

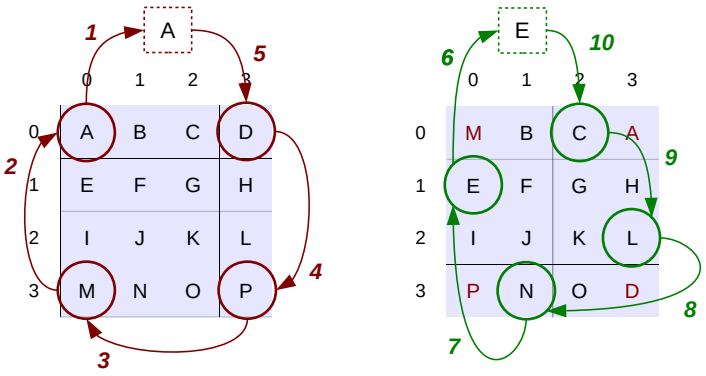
```
for ( int i = 0 ; i < BOARDSIZE ; ++i )   slideLine(i) ;
```

Il serait tentant d'écrire le même genre de traitement pour les déplacements vers la gauche, le haut et le bas ; mais nous allons mettre en oeuvre une solution beaucoup plus rapide et plus élégante…

Cette solution consiste à « pivoter » le plateau de jeu de 90, 180 ou 270 degrés avant d'appliquer notre algorithme de traitement d'un glissement vers la droite ; puis à finir la rotation pour ramener le plateau dans son orientation initiale !

La rotation de tableau est un grand classique en algorithmique, étudions donc une procédure de rotation d'un quart de tour vers la droite.

On commence par les 4 coins. L'élément d'indice (0,0) est copié dans une variable temporaire, on copie ensuite chaque coin dans le coin suivant (en tournant dans le sens horaire), et on termine par la copie de la sauvegarde temporaire dans le coin d'indice (3,0).
On applique le même principe en commençant par l'élément d'indice (0,1)… et ainsi de suite.
La figure ci-contre illustre les 10 premières copies de valeurs →



Q11. Représentez l'état de la grille ci-dessus <u>après</u> la 15e copie de valeurs →

M	I	E	A
N	F	G	B
O	J	K	C
P	L	H	D

Ajoutez à la classe `Pow211` la nouvelle <u>méthode privée</u> ci-dessous ; elle satisfait la procédure décrite précédemment.

```
1  void Pow211::rotate()
2  {
3      int n =  BOARDSIZE ;
4      for ( int x = 0 ; x < n / 2 ; ++x ) {
5          for ( int y = x ; y < n - x - 1 ; ++y ) {
6              quint16 tmp = m_board[ x ][ y ] ;
7              m_board[ x ][ y ] = m_board[ y ][ n - x - 1 ] ;
8              m_board[ y ][ n - x - 1 ] = m_board[ n - x - 1 ][ n - y - 1 ] ;
9              m_board[ n - x - 1 ][ n - y - 1 ] = m_board[ n - y - 1 ][ x ] ;
10             m_board[ n - y - 1 ][ x ] = tmp ;
11         }
12     }
13 }
```

Q12. Dans notre cas, combien de fois est exécuté le bloc d'instructions des lignes 6 à 10 ? . 4 .....

Testez l'algorithme en adaptant provisoirement le constructeur de `Pow211`.

Pour en terminer avec les déplacements et fusions de tuiles, il faut fournir maintenant une <u>interface publique</u> constituée d'une méthode pour chacune des directions :

`bool slideUp() ;` `bool slideDown() ;` `bool slideLeft() ;` `bool slideRight() ;`

Implémentez cette interface en vous inspirant de l'exemple ci-contre →

```
1  bool Pow211::slideUp()
2  {
3      bool modified = false ;
4
5      rotate() ;
6      for ( int i = 0 ; i < BOARDSIZE ; ++i ) modified |= slideLine(i) ;
7      rotate() ;
8      rotate() ;
9      rotate() ;
10
11     return modified ;
12 }
```

4. Ajout automatique de tuiles

Après chaque glissement ayant entrainé une modification du plateau, une nouvelle tuile doit être ajoutée sur un emplacement libre. Nous devons donc ajouter à notre classe la capacité de recenser le nombre et les positions des cases libres.

Ajoutez les 2 membres privés suivants :

```
int m_freeCount ;
int m_freeIndex[ BOARDSIZE * BOARDSIZE ] ;
```

Préparez une nouvelle méthode privée de prototype : `int findFreeCells() ;`

Cette méthode doit remplir `m_freeIndex` avec les indices de `0` à `(BOARDSIZE * BOARDSIZE – 1)` des cases libres, mettre à jour le compteur `m_freeCount` et retourner la valeur de ce compteur.

Q13. Complétez le code proposé ci-contre →

```
1  int Pow211::findFreeCells()
2  {
3      m_freeCount = 0 ;
4
5      for ( int i = 0 ; i < BOARDSIZE * BOARDSIZE ; ++i ) {
6          m_freeIndex[i] = 0 ;
7          if ( m_board[ i % BOARDSIZE ][ i / BOARDSIZE ] == 0 ) {
8              m_freeIndex[i]= i;
9              m_freeCount++;
10         }
11     }
12     return m_freeCount;
13 }
```

La bibliothèque Qt propose un générateur pseudo-aléatoire nommé `qrand()`, comme pour le générateur standard de la lib. C, celui-ci doit être initialisé par une « graine » de départ basée sur le temps pour donner l'illusion d'un vrai tirage aléatoire ; cette post-initialisation est réalisée par la fonction `qsrand()`.

Placez l'inclusion de `QTime` dans *pow211.cpp* puis ajoutez dans le constructeur de la classe l'instruction d'initialisation : `qsrand( (uint) QTime::currentTime().msec() ) ;`

Nous avons maintenant tous les éléments nécessaires au développement d'une méthode privée d'ajout d'une tuile de valeur 2 ou 4.

Implémentez cette méthode telle que proposée ci-dessous :

```
1  void Pow211::addRandomTile()
2  {
3      if ( !findFreeCells() )      return ;
4
5      int i = m_freeIndex[ qrand() % m_freeCount ] ;
6      m_board[ i % BOARDSIZE ][ i / BOARDSIZE ] = ( (qrand() % 10 ) / 8 + 1 ) * 2 ;
7  }
```

Q14. Quel est le rôle de la ligne 5 ?

Le rôle de la ligne 5 est de choisir aléatoirement une case dans le tableau m_freeIndex

Q15. Quel est le pourcentage de chance d'obtenir une tuile de valeur 4 ?

Il y a 8 cas où la valeur sera 2.
Il y a 2 cas où la valeur sera 4.
Il y a 10 cas au totals, il y a donc 20% de chance d'avoir un 4.

Ajoutez 2 appels successifs à `addRandomTile()` dans votre constructeur et d'autres placés judicieusement quand il y a lieu, puis testez le programme.

5. Évaluation de fin de partie

La partie est considérée comme terminée et gagnée si une tuile de valeur `MAXVALUEOFTILE` apparaît sur le plateau. La partie est perdue si toutes les cellules du plateau sont occupées et qu'il n'y a plus aucune fusion possible…

Nous devons donc être capables de vérifier s'il existe au moins une paire de tuiles adjacentes de même valeur, aussi bien horizontalement que verticalement.

Commençons par tester cela sur une ligne…

Ajoutez la nouvelle méthode privée ci-dessous :

```
1  int Pow211::findPairInLine(int line )
2  {
3      int num = 0 ;
4      for ( int x = 0 ; x < BOARDSIZE - 1 ; ++x ) {
5          if ( m_board[x][line] == m_board[x + 1][line] ) num++ ;
6      }
7      return num ;
8  }
```

Q16. Proposez maintenant une autre méthode privée `findPair()` assurant le comptage des fusions possibles sur les 2 axes et retournant le nombre trouvé :

```
int Pow211::findPair()
{
    int num = 0 ;
    for ( int i = 0 ; i < BOARDSIZE ; i++ ) num += findPairInLine(i) ;
    for(int i = 0; i < BOARDSIZE; i++)
    {
      for(int j = 0; j < BOARDSIZE; j++)
      {
          if ( m_board[i][j] == m_board[i][j + 1] ) num++ ;
      }
    }
    return num ;
}
```

L'interface publique de la classe peut maintenant fournir deux méthodes de diagnostic. La première signale si la partie est gagnée, la deuxième indique que le jeu est bloqué et donc la partie perdue.

Q17. Complétez ces deux méthodes :

```
bool Pow211::win()
{
    if(m_higherTile == 2048) return 1;
    else return 0;
}

bool Pow211::gameOver()
{
    if(!findPair()) return 1;
    else return 0;
}
```

Nous en avons (enfin !?) fini avec l'algorithmique, place à la partie visualisation graphique et à la gestion des interactions avec l'utilisateur…

6. Grille de jeu

La première approche consiste à définir dans l'interface graphique un élément de type `Grid` et à fournir une instance de `Pow211` à QML afin qu'il puisse récupérer les valeurs des tuiles.

Ajouter l'inclusion de `QtQml` dans *main.cpp*.
En supposant que votre fonction `main()` contienne une instruction du style `pow211.print()`, où `pow211` est une instance de classe `Pow211`, vous pouvez remplacer cette instruction par la suivante :

```
engine.rootContext()->setContextProperty("board", &pow211 ) ;
```

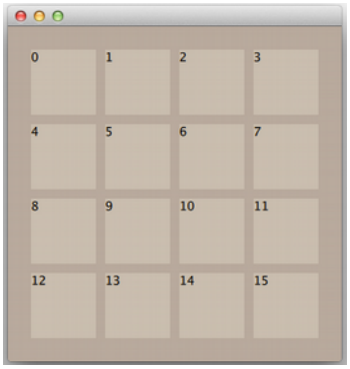
Cette instruction va assurer la reconnaissance de l'objet de classe `Pow211` au sein de l'interface QML sous le nom `board`.

Éditez maintenant le fichier *main.qml* →

lignes 9 à 12 : l'élément de base est un rectangle avec une couleur de fond qui occupe tout l'espace de son parent.

lignes 14 à 18 : cet élément contient un autre élément de type grille 4 x 4 avec une marge de 10 pixels entre les cases.

lignes 20 à 30 : le modèle répétitif de remplissage est simplement spécifié par le nombre de cases à remplir. Le remplissage proprement dit est délégué à un objet de type rectangle 70 x 70 avec une couleur de fond et l'indice de la case en tant que texte.




```
1  import QtQuick 2.2
2  import QtQuick.Window 2.1
3
4  Window {
5      visible: true
6      width: 360
7      height: 360
8
9      Rectangle {
10         id: main
11         color: "#baaa9e"
12         anchors.fill: parent
13
14         Grid {
15             id: grid
16             rows: 4
17             columns: 4
18             spacing: 10
19
20             Repeater {
21                 model: 16
22                 delegate: Rectangle {
23                     width: 70
24                     height: 70
25                     color: "#cbbeb1"
26                     Text {
27                         text: index
28                     }
29                 }
30             }
31         }
32     }
33 }
```

Q18. Recherchez dans la documentation la ou les propriétés à ajouter à l'élément `Grid` pour le centrer sur l'espace de son parent et obtenir une IHM telle que le *screen shot* ci-dessus :

.. anchors.centerIn: parent .....
.....

Dans *pow211.h*, ajoutez `Q_INVOKABLE` devant le prototype de la méthode publique `tile()`.
Dans *main.qml*, spécifiez comme texte de remplissage l'invocation `board.tile(index)` (en ligne 27).

Vous obtenez normalement une grille avec toutes les tuiles nulles sauf deux qui valent 2 ou 4 ! Relancez plusieurs fois votre application pour voir si votre générateur aléatoire de début de partie fonctionne correctement.

Dans *main.qml*, en ligne 21, le modèle de grille fixé de manière absolue à 16 sera par la suite initialisé avec l'objet `board` afin de s'adapter à d'éventuelles autres dimensions (cf. p.11).

7. Style des tuiles

Vous allez voir ici comment personnaliser l'aspect visuel des tuiles en définissant un délégataire sur mesure sous la forme d'un fichier QML indépendant.

☑ Ajoutez au projet un nouveau fichier de type « Fichier QML (Qt Quick 2) » nommé *Tile.qml* (avec une majuscule). Éditez le fichier et ajustez l'importation pour disposer de la version 2.2 de Qt Quick.

Pour faire simple, nous pouvons ensuite choisir d'effectuer le dessin des tuiles avec un rectangle à coins arrondis et de doter cet élément de propriétés modifiables depuis notre application C++…

☑ Remplissez *Tile.qml* de cette manière →

lignes 5 à 7 : propriétés partageables.

lignes 9 à 13 : fonction Javascript de calcul de la taille du texte en fonction du nombre de caractères ; cette fonction est invoquée ligne 21.

```
1  import QtQuick 2.2
2
3  Rectangle {
4      id: tile
5      property int    tileWidth: 70
6      property string tileValue: ""
7      property string tileColor: "lightsteelblue"
8
9      function fontSize(value) {
10         var v = parseInt(value, 10);
11         if (v < 1024) return tileWidth / 2.5;
12         return tileWidth / 3;
13     }
14
15     width:  tileWidth
16     height: tileWidth
17     radius: tileWidth / 10
18     color:  tileColor
19     Text {
20         text: tileValue
21         font.pointSize: fontSize(tileValue)
22         anchors.centerIn: parent
23         color: "black"
24     }
25 }
```

☑ Adaptez maintenant le `Repeater` de *main.qml* pour sélectionner le nouveau `delegate` et affecter correctement la propriété `tileValue` :

```
delegate: Tile
tileValue: board.tile(index)
```

Testez à nouveau le programme.

Pour améliorer le rendu, nous pouvons choisir de ne pas mettre de texte lorsque la valeur d'une tuile est nulle.

☑ Dans *Tile.qml*, remplacez `text: tileValue` par `text: valueToText(tileValue)`

Q19. En vous inspirant de la fonction `fontSize()`, écrivez la fonction Javascript requise de manière à retourner un texte vide si la valeur est nulle :

```
function valueToText(value) {
    ..........
    ..........
    ..........
    ..........
}
```

D'un point de vue visuel, il est aussi souhaitable d'affecter des couleurs différentes aux tuiles en fonction de leur valeur. On peut pour cela doter la classe `Pow211` d'une nouvelle méthode publique de prototype :

```
Q_INVOKABLE QString color(int index ) const ;
```

qui retourne un code couleur sous la forme `"#rrggbb"` fonction de la valeur de la tuile d'indice `index`.

☑ En prévision, ajoutez dans main*.qml* l'affectation de la propriété `tileColor: board.color(index)`

Quant à implémentation de la méthode C++, vous pouvez vous baser sur l'embryon ci-dessous (le choix des couleurs est de votre ressort !).


```
1  QString Pow211::color(int index ) const
2  {
3      quint16 value = tile( index ) ;
4
5      switch ( value ) {
6          case    2: return "#eee4da" ;
7          case    4: return "#eae0c8" ;
8          // TODO
9          default:   return "#cbbeb1" ;
10     }
11 }
```

L'illustration a été obtenue en plaçant un appel provisoire à `setBoard()` dans le constructeur (avec le jeu de valeurs proposé page 3)… L'effet dégradé a été obtenu en remplaçant l'attribut `color` de `Tile` par un attribut `gradient` variant du blanc à la couleur de la tuile.

8. Interactions C++/QML

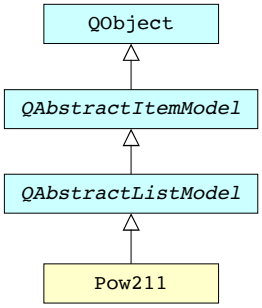
Vous savez déjà que le modèle du `Repeater` de la grille peut être une simple valeur numérique ou un objet dérivé de la classe `QObject` (ce qui est le cas de notre objet de classe `Pow211` actuellement utilisé).

Vous allez découvrir maintenant que ce modèle peut aussi être un objet beaucoup plus élaboré qui va faciliter grandement la communication entre nos deux couches logicielles ; Qt propose en effet un ensemble de classes virtuelles propres à gérer des ensembles tels que celui des tuiles de notre jeu.

La classe `QAbstractListModel` sera ici utilisée comme modèle C++ pour présenter nos données à QML, non pas comme un tableau mais en tant que liste d'éléments (il s'agit d'une sérialisation, ce qui ne devrait pas poser de problème puisque nous manipulons déjà nos tuiles à partir de leur indice).

L'avantage de cette classe est qu'elle est capable d'informer automatiquement QML lorsque les données sont modifiées ; elle possède par défaut 2 rôles `Qt::DisplayRole` et `Qt::DecorationRole` qui seront reconnus par QML sous les noms `display` et `decoration`. Nous n'avons donc pas nécessité de définir d'autres rôle : le premier va nous permettre de mettre à jour la valeur des tuiles et le deuxième leur couleur…

Commençons par modifier la parenté de Pow211 :



Surclassez `Pow211` en la dérivant maintenant de `QAbstractListModel`. N'oubliez pas pour cela de modifier le constructeur en plus de la déclaration de la classe.

`QAbstractListModel`, comme son nom l'indique, est une classe abstraite ; son modèle de développement est constitué de deux méthodes virtuelles pures issues de `QAbstractItemModel` :

```
int QAbstractItemModel::rowCount(const QModelIndex& parent ) const ;
```

→ qui doit simplement retourner le nombre d'éléments de la liste (l'argument `parent` n'est pas utilisé) ;

```
QVariant QAbstractItemModel::data(const QModelIndex& index, int role ) const ;
```

→ qui doit retourner une valeur en fonction de l'indice et du rôle qu'elle reçoit.

Q20. Complétez l'implémentation de la méthode `rowCount()` :

```
int Pow211::rowCount(const QModelIndex& parent ) const
{
    Q_UNUSED( parent ) ;
    return ..... ;
}
```

Ajoutez la méthode `data()` telle que proposée ci-dessous. La variable locale `value` récupère la valeur de la tuile concernée et l'argument `role` spécifie la caractéristique à renvoyer.

note : la classe `QVariant` agit comme une union des types de données les plus courants de Qt ; elle est utilisée ici en tant que `QString`.

```
1  QVariant Pow211::data(const QModelIndex& index, int role ) const
2  {
3      if ( role == Qt::DisplayRole ) {
4          quint16 value = tile( index.row() ) ;
5          QString str = "" ;
6          if ( value ) str = QString::number( value ) ;
7          return str ;
8      }
9      if ( role == Qt::DecorationRole ) {
10         return color( index.row() ) ;
11     }
12     return QVariant() ;
13 }
```

Reprenez *main.qml* et modifiez les propriétés du délégataire pour leur associer les rôles qui nous intéressent →

```
model: board
… …
tileValue: display
tileColor: decoration
```

Testez l'application, les tuiles présentes, leur valeur et leur couleur doivent toujours être conforme à la configuration de départ que vous avez dans le constructeur de `Pow211`.

Il est temps de faire vivre notre plateau de jeu !

Les mécanismes à mettre en place sont relativement simples :

- l'interface QML doit détecter les appuis sur les touches flèches du clavier et invoquer les méthodes `slide...()` de la classe `Pow211` ;
- la classe `Pow211` doit informer l'interface de tous les changements de position et de valeurs des tuiles.

Appliquez le modificateur `Q_INVOKABLE` aux quatre méthodes `slideUp()` … `slideRight()` de `Pow211`.

Dans l'interface QML, ajoutez à l'item rectangle `main` la capacité de détection d'appui sur les touches flèches et l'appel aux méthodes idoines →

note : La propriété `focus` permet de focaliser les entrées sur l'item ; comme il n'est pas possible de le faire au niveau de l'objet `Window`, le rectangle a été introduit en lui spécifiant de remplir tout le parent via `anchors.fill: parent`.

```
1  focus: true
2  Keys.onPressed: {
3    switch (event.key) {
4      case Qt.Key_Up:
5          board.slideUp();
6          break;
7      case Qt.Key_Right:
8          board.slideRight();
9          break;
10     case Qt.Key_Down:
11         board.slideDown();
12         break;
13     case Qt.Key_Left:
14         board.slideLeft();
15         break;
16   }
17 }
```

Et pour signaler les modifications du plateau, nous allons utiliser l'émission du signal `dataChanged()` hérité lui aussi de `QAbstractItemModel` de la manière suivante :

```
emit dataChanged(createIndex(0, 0), createIndex(rowCount() - 1, 0 ) ) ;
```

Cet usage n'est pas sélectif, il signale que tous les rôles de tous les éléments ont changé… Il est probablement possible d'optimiser cela !

Q21. Où devez-vous placer cette émission de signal, et à quelle condition ?

.....

.....

.....

Q22. Testez le programme, est-il jouable (déplacements, fusions, nouvelles tuiles) ? .....
Si non, qu'avez-vous oublié ?
.....
.....
.....
.....

Vous avez le droit de prendre 5 à 10 minutes pour essayer votre nouveau jouet 

9. Un peu plus d'ergonomie ?

Dans la perspective de portage de notre application sur tablettes tactiles, le pilotage par le clavier n'est bien évidemment pas suffisant. Heureusement, grâce aux kits du SDK Qt, les mouvements des doigts sur l'écran se traitent exactement de la même manière que ceux de la souris pour une application *Desktop*.

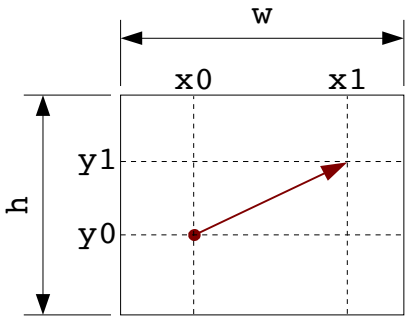
- appui sur le bouton gauche de la souris <=> pose d'un doigt sur l'écran
- déplacement de la souris bouton enfoncé <=> glissement (*swipe*) du doigt sur l'écran
- relâchement du bouton de la souris <=> relèvement du doigt

Il convient donc de :

- détecter l'appui ;
- déterminer le vecteur de déplacement jusqu'au relâchement ;
- invoquer une des méthodes `slide...()` de `Pow211`.

Commençons par étudier une fonction Javascript qui calcule une direction à partir d'un point de départ, d'un point d'arrivée et d'une aire de déplacement.

ajoutez à l'item primaire de `main.qml` les propriétés et la fonction ci-contre →




```
1  property int gestureLeft:  0;
2  property int gestureDown:  1;
3  property int gestureRight: 2;
4  property int gestureUp:    3;
5
6  function getGesture(x0, y0, x1, y1, w, h )
7  {
8      var deltaX = ( x1 - x0 ) / w ;
9      var deltaY = ( y1 - y0 ) / h ;
10     var minFactor = 0.20 ;
11
12     if (  Math.abs(deltaX) < minFactor
13        && Math.abs(deltaY) < minFactor ) return ;
14
15     if ( Math.abs(deltaX) > Math.abs(deltaY) ) {
16         if ( deltaX > 0 )   return gestureRight ;
17         return gestureLeft ;
18     }
19     else {
20         if ( deltaY > 0 )   return gestureDown ;
21         return gestureUp ;
22     }
23 }
```

Les variables `deltaX` et `deltaY` sont les déplacements signés relatifs à la zone de la grille de jeu.

Q23. Quel est le rôle des lignes 12 et 13 ?
.....
.....
.....

Il faut ensuite intercepter les événements en provenance de la souris quand celle-ci est sur l'aire de jeu.

Ajoutez l'item `MouseArea` avec la mémorisation du point de départ (*onPressed*) et un appel à la fonction `getGesture()` lors du relâchement (*onReleased*).

Complétez l'étude de cas.

```
1  MouseArea {
2      id: mouseArea
3      anchors.fill: parent
4
5      property int startX: 0
6      property int startY: 0
7
8      onPressed: {
9         startX = mouseX;
10        startY = mouseY;
11     }
12     onReleased: {
13         var gesture = getGesture(
14               startX, startY,
15               mouseX, mouseY,
16               mouseArea.width, mouseArea.height );
17
18         switch( gesture ) {
19         // TODO
20         }
21     }
22 }
```

Complétez l'étude de cas ligne 19.

Testez à nouveau le programme. Les tuiles doivent pouvoir être déplacées au moyen de la souris !

Magique, non ?

10. Affichage du score et fin de partie

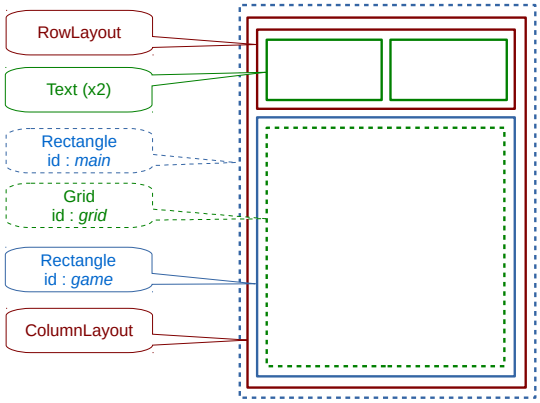
Pour mémoire, vous avez déjà développé au §5 les méthodes de détection de fin de partie. En Q9, vous avez fait le nécessaire pour que le score soit tenu à jour au fur et à mesure des fusions de tuiles et vous disposez, depuis le début du projet, d'un sélecteur pour ce score…

Améliorons légèrement notre IHM pour l'affichage de ces informations.

Nous pouvons par exemple réserver un bandeau dans la partie haute de la fenêtre d'application pour y placer deux champs texte ; un pour le score et un autre pour afficher des messages tels que « *game over* » ou « *you win!* ».

L'arrangement (*layout*) des différents éléments se fait au moyen de conteneurs verticaux et/ou horizontaux. La figure ci-contre est une proposition de solution (les éléments en pointillés sont ceux déjà existants) →

<u>note</u> : le rectangle *game* n'est pas indispensable pour le moment, il ne sera exploité qu'au paragraphe suivant.



Ajoutez la clause `import QtQuick.Layouts 1.1` au début du fichier *main.qml* et commencez la mise en place des arrangements :

```
Window {
  visible: true
  width: 360
  height: 360

  Rectangle {
    id: main
    color: "#baaa9e"
    anchors.fill: parent

    focus: true
    /* … */
  }

  /* … */
}
```

```
ColumnLayout {
  anchors.fill: parent
  RowLayout {

  }
  Rectangle {
    id: game
    Layout.fillHeight: true
    Layout.fillWidth: true
    color: "#baaa9e"

    Grid {
      /* … */
    }
  }
}
```

```
Text {
  id: message
  horizontalAlignment: Text.AlignHCenter
  Layout.fillWidth: true
  text: {
    return "Pow211 - v1.0"
  }
}

Text {
  id: score
  horizontalAlignment: Text.AlignHCenter
  Layout.minimumWidth: 100
  text: board.score
  font.pointSize: 24
}
```

Vous avez sans doute remarqué que le texte provisoire de l'élément *message* est la version de notre application tandis que celui de l'élément *score* fait visiblement référence à une méthode de l'objet *board*, autrement dit de la classe `Pow211`…

Qualifier la méthode `Pow211::score()` avec `Q_INVOKABLE` ne serait malgré tout pas suffisant ici. En effet, il ne suffit pas d'initialiser l'élément au démarrage, il faut encore que celui-ci soit mis à jour à chaque changement du score.

La solution ? Déclarer une propriété associée à un sélecteur et un signal.

Ajoutez à votre classe `Pow211` avec la ligne suivante (à placer sous la clause `Q_OBJECT`) :

```
Q_PROPERTY(quint32 score READ score NOTIFY scoreChanged )
```

Déclarez le signal :

```
void scoreChanged() ;
```

Q24. Où allez-vous placer l'instruction d'émission du signal `scoreChanged()` ?

.....
.....

Testez le programme ; le score doit évoluer au fur et à mesure des fusions des tuiles.

Vous pouvez utiliser exactement la même stratégie pour prendre en compte les méthodes `Pow211::win()` et `Pow211::gameOver()` et mettre à jour l'élément message.

Ajoutez les nouvelles propriétés et un nouveau signal (notez que le même signal est utilisé pour toutes les notifications liées aux tests de fin de partie) :

```
Q_PROPERTY(bool gameOver READ gameOver NOTIFY gameStatusChanged)
Q_PROPERTY(bool win READ win NOTIFY gameStatusChanged)
```

Q25. Où allez-vous placer l'instruction d'émission du signal `gameStatusChanged()` ?

.....
.....

Adaptez pour finir l'élément texte *message* en introduisant les deux lignes suivantes :

```
if (board.win)      return "YOU WIN!"
if (board.gameOver) return "GAME OVER"
```

Testez à nouveau le programme ; et faites en sorte de perdre et de gagner pour voir les messages…
Si vous n'arrivez pas au but, diminuer la valeur de la tuile max. à atteindre (*little player*…)
Si vous y arrivez trop vite, augmentez-la (*champion*…)

La figure ci-contre donne un aperçu du résultat après avoir modifié quelques propriétés pour enjoliver les champs de texte

(`color`, `opacity`, `font.pointSize`, `font.family`, …) →

Notre produit semble « livrable »…

Il reste cependant un petit problème à régler.

Pour vous en convaincre, essayez de retailler la fenêtre d'application…

11. Auto-adaptation de la taille des tuiles

Les tuiles doivent toujours rester carrées mais le plateau (*grid*) doit occuper au mieux l'espace attribué par son conteneur (*game*).

Par défaut, nous avons actuellement un espacement entre tuiles fixé à 10 pixels et des tuiles de 70 pixels de côté.

Si l'on raisonne en unités de 10 pixels, le côté de notre plateau vaut : 1 + 7 + 1 + 7 + 1 + 7 + 1 + 7 + 1 = 33.

Ajoutez à l'élément *game* une fonction qui calcule cette unité par rapport ses propres dimensions →

```
1 function step() {
2   var s = Math.min(game.width, game.height) / 33 ;
3   if ( s == 0 ) return 1 ;
4   return s ;
5 }
```

note : la ligne 3 évite un avertissement au démarrage lié à l'affectation d'une valeur nulle à la propriété `fontSize` du texte des tuiles.

Utilisez ensuite ce calcul pour définir l'espacement des éléments de la grille : `spacing: game.step()`
Et fixez la taille des tuiles via la propriété adéquate de `Tile` : `tileWidth: game.step() * 7`

Les tuiles et leur texte s'adaptent maintenant à la taille de la fenêtre d'application. À vous d'améliorer encore le code QML afin de faire suivre de la même manière la taille des textes du bandeau haut.

Notre taille de grille est définie par la constante `BOARDSIZE`. Il peut être intéressant de tester le jeu avec un plateau différent ; par exemple 5 x 5 ou 6 x 6…

Enrichissez la classe `Pow211` d'un sélecteur public

```
Q_INVOKABLE int size() const ;
```

qui retourne simplement la valeur de la constante `BOARDSIZE`.

Q26. Quelles modifications devez-vous apporter au fichier *main.qml* pour que le programme fonctionne correctement quelle que soit la taille de grille imposée ?

.....
.....
.....
.....

12. Déploiement sur mobile Android

On suppose ici que la chaine de développement pour Android est correctement installée et que Qt Creator est configuré en conséquence (cf. tutoriel *Qt-5.x-mobiles*) ; vous devez par ailleurs disposer d'au moins un androphone, qu'il soit virtuel (AVD) ou réel.

À partir du mode **Projet** de Qt Creator, déroulez la liste **Ajouter un kit** et sélectionnez le kit « Android pour armeabi-v7a ».
La rubrique **Général** de l'onglet **Compiler** permet d'activer ou non le *shadow build*. Activez-le de manière à ne pas écraser les fichiers de construction de l'application Desktop (la construction de la cible pour le nouveau kit sera faite dans un répertoire indépendant situé par défaut au même niveau que le répertoire de base du projet).

L'onglet **Exécuter** du kit Android donne accès à la rubrique **Configurations de déploiement**. Le détail de cette rubrique propose un bouton « Créer le fichier AndroidManifest.xml ».

Provoquez la création du manifeste. Vous devez obtenir à la base du projet un sous-répertoire nommé `android` qui contient le nouveau fichier.

Le manifeste apparaît en mode **Édition** de Qt Creator dans **Autres fichiers | android**, il peut être édité en mode texte grâce au bouton « Source XML » situé dans le bandeau supérieur.

Vous pouvez éventuellement personnaliser votre programme par une icône d'application :

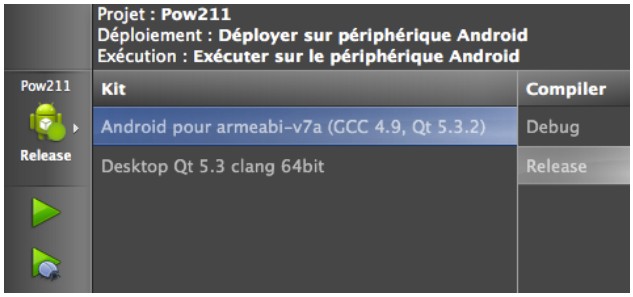
- ajoutez dans le répertoire `android` un sous-répertoire `res`, lui-même devant contenir les divers sous-répertoires `drawable-…` avec le même PNG sous différents formats (fichiers nommés par exemples *ic_launcher.png*) ;
- adaptez le manifeste en ajoutant la propriété `android:icon="@drawable/ic_launcher"` à la balise `<application>`.

Vous pouvez aussi personnaliser l'identité du paquetage en remplaçant la chaine par défaut « `org.qtproject.example` » ; respectez néanmoins les conventions de nommage qui sont celles des paquets Java, par exemple pour un usage privé (non publié) :

`org.`*`author_or_company_name`*`.`*`product_name`*

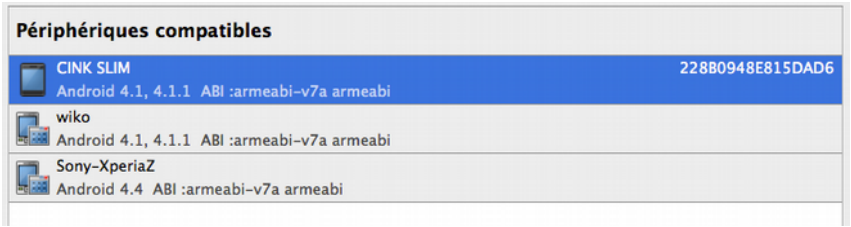
Sélectionnez la cible à fabriquer →

Et cliquez sur **Run** !



Qt Creator fait appel à un outil nommé `androiddeployqt` pour fabriquer le paquetage Android.

Le résultat `.apk` est placé dans le sous-répertoire `android-build/bin` de la base du *shadow build*.

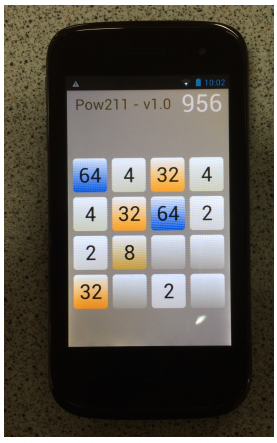


Qt Creator vous demande ensuite de choisir l'appareil destinataire du paquetage…

← ici un exemple avec un appareil réel relié en USB et deux AVD.

Exemples de résultats sur émulateur de tablette et sur appareil réel :

Notez que la rotation est gérée automatiquement…



13. Déploiement sur mobile iOS

On suppose ici que la chaine de développement pour iOS (Xcode) est correctement installée et que Qt Creator est configuré en conséquence ; les appareils mobiles iOS utilisés doivent bénéficier d'un profil d'approvisionnement de développement valide (cf. tutoriel *Qt-5.x-mobiles*).

☑ À partir du mode **Projet** de Qt Creator, déroulez la liste **Ajouter un kit** et sélectionnez le kit « iphonesimulator-clang Qt 5.x for iOS ».
La rubrique **Général** de l'onglet **Compiler** permet d'activer ou non le *shadow build*. Activez-le de manière à ne pas écraser les fichiers de construction de l'application Desktop (la construction de la cible pour le nouveau kit sera faite dans un répertoire indépendant situé par défaut au même niveau que le répertoire de base du projet).

L'onglet **Exécuter** permet de choisir le type de *device* à simuler.

☑ Sélectionnez la cible à fabriquer →

Et cliquez sur **Run** !

Qt Creator fait appel à Xcode pour fabriquer la cible *Pow211.app*. Le résultat est localisé pour dans le sous-répertoire `Release-iphonesimulator` de la base du *shadow build*.

Pour des tests sur des appareils de type iPhone 5 (4") et iPad (9,7"), nous devons préparer notre projet afin de fournir des icônes d'application dans des résolutions au moins égales à, respectivement, 120 x 120 et 152 x 152 pixels (cf. tutoriel *Qt-5.x-mobiles*).

Les ressources des *bundles* iOS sont décrites par un fichier nommé *Info.plist*. Xcode s'est chargé de créer un tel fichier à la racine du *shadow build*.

☑ Créez dans le répertoire du projet un nouveau sous-répertoire nommé par exemple `ios` ; puis copiez dans ce répertoire le fichier *Info.plist* produit par Xcode. Ce fichier va nous servir de base pour ajouter nos images personnelles…

☑ Créez encore un répertoire `ios/bundle_data` destiné à contenir au moins une version d'icône PNG de dimensions minimales 152 x 152 pixels.
Dans ce qui suit, on suppose disposer de 2 fichiers : *icon_128x128.png* et *icon_192x192.png*.

☑ Éditez le fichier *Pow211.pro* de manière à prendre en charge les nouvelles images et le fichier *Info.plist* personnel ; ajoutez pour cela les lignes ci-contre →

```
1 mac:!macx {
2 BUNDLE_DATA.files = \
3         $$PWD/ios/bundle_data/icon_128x128.png \
4         $$PWD/ios/bundle_data/icon_192x192.png
5
6 QMAKE_BUNDLE_DATA += BUNDLE_DATA
7 QMAKE_INFO_PLIST = $$PWD/ios/Info.plist
8 }
```

note : la clause `mac` est vraie pour Mac OS et pour iOS ; `macx` ne l'est que pour Mac OS.

☑ Une fois le fichier projet sauvegardé, notre fichier *Info.plist* apparaît dans la rubrique **Autres fichiers** de notre projet. Il peut être donc directement édité au sein de Qt Creator. Éditez le fichier en remplaçant la balise `CFBundleIconFile` par →

(*les images sont recensées sans leur extension*)

```
1 <key>CFBundleIconFiles</key>
2 <array>
3     <string>icon_128x128</string>
4     <string>icon_192x192</string>
5 </array>
```

Après essais sur le simulateur iOS, vous remarquez que l'icône choisie apparaît bien lorsque vous quitter le programme (iOS Simulator, commande **Hardware | Home**) ; il est cependant possible que le changement d'orientation ne soit pas géré correctement (commandes **Hardware | Rotate Left** et **Rotate Right**).

Cet inconvénient peut être résolu en interdisant la rotation de notre interface.
Ajoutez au fichier *Info.plist* la balise ci-contre →

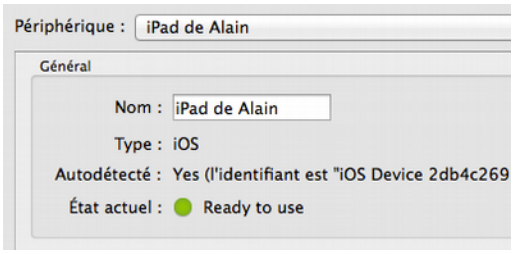
```
1 <key>UISupportedInterfaceOrientations</key>
2 <array>
3     <string>UIInterfaceOrientationPortrait</string>
4 </array>
```

Et avec un appareil iOS réel ?

À partir du mode **Projet** de Qt Creator, déroulez la liste **Ajouter un kit** et sélectionnez le kit « iphoneos-clang Qt 5.x for iOS ».
La rubrique **Général** de l'onglet **Compiler** permet d'activer ou non le *shadow build*. Activez-le de manière à ne pas écraser les fichiers de construction de l'application Desktop (la construction de la cible pour le nouveau kit sera faite dans un répertoire indépendant situé par défaut au même niveau que le répertoire de base du projet).

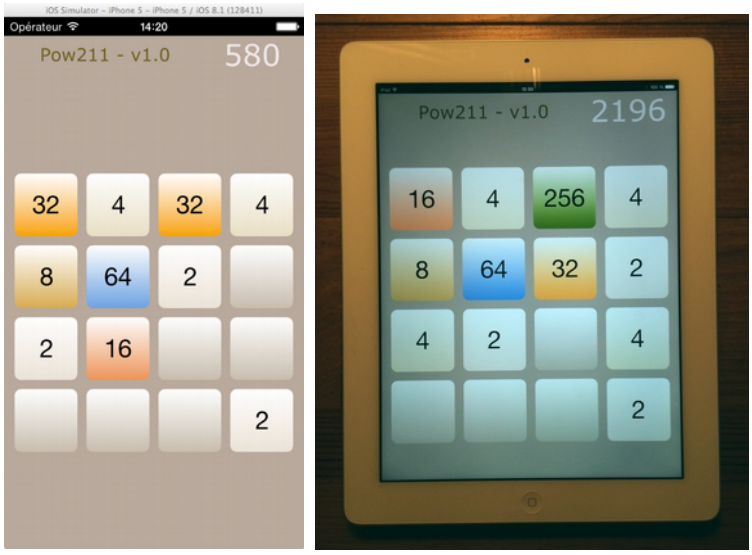
La commande Qt Creator **Préférences… | Compiler & Exécuter** permet, pour le kit concerné, de sélectionner l'appareil à utiliser.

Préférences… | Appareils mobiles permet aussi de vérifier que l'appareil est bien connecté et *Ready to use* (un appareil marqué seulement *Connected* signifie un problème de profil de développement).



Il ne reste qu'à cliquer sur **Run** !

Exemples de résultats sur émulateur iOS de smartphone et sur tablette iPad réelle →



Remarque : l'icône de l'application *Desktop* pour Mac OS est spécifiée différemment par une clause particulière dans le fichier projet ; l'image doit être au format ICNS (*Apple Icon Image*).

Supposons par exemple la présence d'un fichier *images/pow211.icns* dans le répertoire du projet ; il suffit d'ajouter la ligne suivante dans le fichier projet :

```
macx:ICON = images/pow211.icns
```

Bon développement à tous !

Code less.
Create more.
Deploy everywhere.