Rapport de stage
Programmation Enyo en OCaml

Vincent Botbol

20 août 2012

Chapitre 1

Introduction

En quelques années, les appareils mobiles se sont peu à peu imposés sur le marché technologique. Grâce à leurs interfaces graphiques simplifiées et leurs fonctionnalités, ces outils ont sû se rendre indispensable auprès du grand-public. Aujourd'hui, environ un tiers de la population possède un smartphone ou une tablette.

La variété de ces appareils est telle que la tâche est parfois agaçante pour les développpeurs d'exporter leurs applications sur d'autres systèmes utilisant des langages différents. La nécessité d'une unification s'est donc imposé. Pour se faire, le choix des constructeurs quant à la technique s'est dirigé vers les nouvelles technologies web : HTML5 et JavaScript. Populaire et en constante évolution, ces deux solutions apportent un grand nombre de possibilités mais possèdent néanmoins leurs désavantages.

Le but de ce stage est de permettre au programmeur de développer des applications portables de manière sûre en passant pour cela par le langage OCaml garant d'une solidité à travers la puissance de son typage statique tout en gardant le confort de programmation qu'on lui connait. Pour se faire, on s'est intéressé à la bibliothèque JavaScript "Enyo" (2) en interfaçant celle-ci afin de satisfaire notre objectif.

La traduction d'OCaml vers JavaScript a déjà été mise-en-oeuvre dans plusieurs travaux : OBrowser et Js_of_ocaml. Fournissant des bases intéressantes pour ce projet, il a été pratique d'en faire usage pour s'abstraire du problème d'inter-opérabilité entre les deux langages.

La divergence entre JavaScript et OCaml est importante. Cela peut beaucoup jouer lors d'un tel interfaçage. Il est alors important de bien étudier la bibliothèque afin de déterminer le niveau d'utilisation des traits intrusifs. En répondant cette question, il faut pouvoir déterminer le type de modèle et sa représentation s'adaptant le mieux. Les objets Enyo étant nombreux, il peut être important de se donner les moyens de les représenter de façon efficace. Se munir d'un IDL peut-il être un atout pour réaliser ceci ?

Premièrement, nous étudierons Enyo pour nous permettre d'obtenir les

informations nécessaires pour établir une approche.

Ensuite, nous tenterons de fournir un modèle en détaillant les raisons de ce choix et les problèmes qui en ressortent.

Enfin, nous présenterons un IDL permettant de modéliser les objets Enyo.

Chapitre 2

Présentation Enyo

2.1 Introduction

Enyo[4] est un framework Javascript orienté-objet. Il privilégie un style de développement basé sur une encapsulation arborescente de composants.

Dans sa première version, Enyo fut employé pour le développement d'application des tablettes HP TouchPad fonctionnant sous webOS. Après un succès mitigé, HP décida de transformer webOS en projet *open source* devenant ainsi : Open WebOS.

Enyo passe alors en version 2.0. Sa conception en Javascript profite de l'engouement actuel pour ce langage et permet aux développeurs de déployer leurs applications sur les plateformes le supportant, ce qui aujourd'hui désigne la plupart des appareils grand public (Android, iOS, WindowsPhone, etc.)

On peut attribuer à Enyo des avantages tels que sa prise en main aisée grâce au style de programmation proche d'une vision du DOM [1], le poids relativement faible de sa bibliothèque et sa possibilité d'extension relativement aisée.

1. Document Object Model

2.2 Débuter en Enyo

Il est important de distinguer trois phases dans le développement d'une application Enyo.

- La première, dite de déclaration, consiste à définir les composants, leurs positions dans la hiérarchie de l'application ainsi que le modèle de calcul.
- La seconde est celle de la génération. Permettant à la bibliothèque Enyo de "compléter" le squelette décrit par la phase de déclaration, l'objet destiné à être instancié.
- Enfin, nous avons la phase d'instanciation où le résultat de la génération est à même d'être créé puis utlisable, par exemple, en l'affichant.

On observe ainsi de fortes similitudes avec un langage de description d'interface. Il faut commencer par *décrire* l'objet, ensuite le *générer* pour enfin l'*utiliser*.

La définition des composants de l'application consiste à déclarer des objets Javascript possédant des propriétés qui auront un sens au moment d'étendre l'application.

Par exemple, définissons un simple morceau de texte contenant la chaine "Bonjour monde !" :

```
1  var monApplication = {kind: "Control",
2                        name: "MonApp",
3                        content: "Bonjour monde!"}
```

Ici, nous avons simplement déclaré un objet Javascript [2] muni de propriétés suffisantes pour renseigner Enyo sur le type d'objet à créer.
Explicitons ces propriétés :

- *kind* définit le type d'objet Enyo que nous souhaitons créer.
- *name* nomme l'objet que nous sommes en train de créer. C'est par la valeur de cette propriété que nous pourrons accéder à un composant ou encore l'instancier. [3]
- *content* attribut un texte au composant. Cette propriété concerne les objets Enyo de type "Control" ou un de ses descendants héritant de celle-ci.

Désormais défini, il faut passer le composant à la méthode *kind* d'Enyo pour générer la nouvelle classe de l'application.

```
1  enyo.kind(monApplication);
```

2. Pour simplifier les déclarations d'objets, nous utilisons la notation Json de JavaScript.

3. Il est important de noter que si placée dans un composant en racine de l'application, la propriété *name* déterminera le nom de la classe générée. Autrement, le nom d'un sous-composant servira de clé à une table d'association définie sur le composant racine de l'application servant à récupérer les divers sous-composants

A la suite de cet appel, on pourra constater qu'une nouvelle classe *MonApp* a été ajouté à l'environnement global de JavaScript. Il ne reste donc plus qu'à l'instancier pour pouvoir ensuite l'afficher.

```
1   var app = new MonApp();
2   app.renderInto(document.body);
```

En s'intéressant de plus près à l'objet "app", on peut constater qu'il possède un nombre important de méthodes et de propriétés que nous n'avons pas définies à la description, telle que la méthode *renderInto* que nous venons d'utiliser.

Ces méthodes ont été ajoutées par induction lors de la génération de l'application par la méthode *enyo.kind*.

Si le champ *kind* spécifie un "Control", il est donc nécessaire à Enyo de lui rajouter toutes les propriétés et méthodes que possède un Control Enyo qui n'ont pas été définies par l'utilisateur.

2.3 Ajout de sous-composants

Enyo suivant une logique d'arborescence de composants. Lorsque l'on crée son application, il est nécessaire de pouvoir lui ajouter des sous-composants. On réalise cela en définissant la propriété *components* avec en valeur le tableau des composants que nous souhaitons.

Imaginons ici que l'on souhaite créer une application consistant d'un champ de saisie et d'un bouton de validation :

```
1  enyo.kind({kind:"Control",
2             name:"MonApp",
3             components: [
4                          {kind:"Input"},
5                          {kind:"Button", content:"ok"}
6                         ]
7            });
```

La fonction kind va récursivement créer tous les sous-composants de la hiérarchie ainsi que leur méthodes associées.

Comme précédemment, pour l'afficher, on instancie la nouvelle classe générée et on lui applique une méthode d'affichage :

```
1   var app = new MonCadre();
2   app.renderInto(document.body);
```

Si l'on observe le code html après ces appels, on s'aperçoit qu'enyo a injecté dans le DOM une balise html de type dans laquelle sera placée, respectivement, deux balises : un <input type="text" ..> et un <button>.

Notons que nous n'avons pas défini la propriété *name* des sous-composants. Rappelons que le nom d'un composant sert uniquement à l'accession de celui-ci par les autres composants de la hiérarchie ou encore pour son instanciation.

Cependant, lors de l'appel à *enyo.kind*, seule la racine de l'arbre sera étendu à l'environnement.

Il faut aussi noter que si la propriété name n'est pas définie, Enyo s'en charge en lui donnant un label généré ("input0" et "button0" ici). Il n'est évidemment pas recommandé d'utiliser ces derniers pour y accéder.

2.4 Événements

Reprenons l'exemple précédent et supposons désormais que l'on souhaite, lorsque l'on clique sur le bouton, afficher un message d'alerte.

```
1  enyo.kind({kind:"Control",
2             name:"MonCadre",
3             components: [
4               {kind:"Input", value:""},
5               {kind:"Button", content:"ok",
6                handlers:{ontap:"monCallback"},
7                monCallback:function(sender, event){
8                  alert("ok");
9                  return true;
10               }
11              }
12            ]
13           });
```

Au bouton, nous avons rajouté une propriété *handlers*. Cet objet sert à définir tous les événements que l'objet va traiter. Cela peut-être des événements du DOM (onkeyup, onload, ...) comme des événements spécifiques à la plateforme sur laquelle l'application est déployée (l'inclinaison d'une tablette par exemple). Ici, nous utilisons l'événement "ontap", alias d'"onclick".

A chaque propriété-événement est attachée une chaine contenant le nom de la fonction à appeler.

Nous définissons donc la fonction "click" qui prend en premier argument l'émetteur de l'événement et l'événement envoyé contenant ses propriétés.

La valeur de retour de cette fonction spécifie si l'événement doit s'arrêter ou remonter ("bubble") aux composants parents. On retourne vrai pour l'empêcher de remonter et faux pour le laisser "buller" sur ses ancêtres.

On notera qu'en général, les fonctions de traitement ne pas censées devoir remonter l'arbre sont définies à la racine qui possède un accès à tous ses sous-composants grâce à sa table de hash "$".

Il existe quelques spécificités pour la manipulation d'événements :

- Si la fonction de traitement n'est pas trouvée, l'événement est remonté dans l'arbre de composants jusqu'à trouver une fonction dont le nom correspond.
- Il n'est nécessaire de déclarer le bloc *handlers* que lorsque la fonction de traitement se situe dans le même objet que celui-ci. Par exemple,

cette définition est parfaitement valable :

```
enyo.kind({kind:"Control",
           components:[{kind:"Bouton",
                        ontap:"traiteClique"}
                      ],
           traiteClique:function(sender, event){
             alert("Click recu"); return true
           }
});
```

– Il est possible de propager un événement manuellement en utilisant la fonction *send(nomEvenement, evenement)* de l'objet *enyo.Signals*. Cet événement est remis à tous les objets de type *kind :"Signals"*. Il suffit alors à l'objet-récepteur de posséder un sous-composant de ce type et d'implémenter la fonction de traitement lié au handler du sous-composant. (voir exemple)

2.5 Propriétés published

Les propriétés placées dans l'objet "published" du composant sont en général utilisées pour la logique de l'application. Certains "published" sont déjà présents dans les objets Enyo. Par exemple, la propriété "content" de l'objet "Control" est elle-même publiée.

Enyo traite ces propriétés comme étant des variables publiques et génère automatiquement : getter, setter ainsi qu'une méthode valueChanged appelée à chaque appel du set avec une nouvelle valeur.

En résumé, un composant déclaré ainsi :

```
{kind:"app", published:{maVariable:0}}
```

est équivalent à :

```
{kind:"app", maVariable:0,
  getMaVariable:function(){..},
  setMaVariable:function(nouvelleValeur){..},
  maVariableChanged:function(ancienneValeur){}}
```

Cependant il reste possible d'initialiser les valeurs de propriétés déjà publiées hors de l'objet *published* de la même façon dont nous procédions avec le champ *content* jusqu'à présent. Cela aura uniquement pour effet de masquer la première initialisation que le constructeur Enyo aura effectué.

Note : la propriété *monObjet.published.maVariable* ne sert qu'à indiquer au constructeur d'objet Enyo qu'il doit générer la variable et sa barrière d'abstraction. Des effets de bords sur cette variable après instanciation n'auront aucun effet.

2.6 Copie, partage et réutilisation de composants

Il est possible de réutiliser des composants déjà créés. Il faut cependant faire attention à ne pas lui attribuer de nom. Dans le cas échéant, le constructeur considèrera que l'objet existe déjà et tous les traitements visant une des occurrences de ce nom seront appliqués au premier objet défini.

```
var unBouton = {kind:"Button", name:"bouton1",
                handlers:{ontap:"clique"},
                content:"un bouton",
                clique:function(){
                  this.setContent("Reception du click")
                }
};

enyo.kind({name:"App", components:[unBouton,unBouton]});
```

Dans cet exemple, au moment de l'instanciation un message d'avertissement concernant une collision de nom est émis mais les deux boutons s'afficheront correctement. Cependant, on constate bien que les clicks émis sur l'un ou l'autre des boutons n'auront pour effet de changer que le premier des deux boutons.

Typiquement, si l'on souhaite réutiliser un composant déjà implémenté, il est recommandé de procéder ainsi :

```
enyo.kind({kind:"Button", name:"MonBouton",
           handlers:{ontap:"clique"},
           content:"un bouton",
           clique:function(){
             this.setContent("Reception du click")
           }
});

enyo.kind({name:"App",
           components:[
             // Creation de deux MonBouton
             {kind:"MonBouton", name:"bouton1"},
             {kind:"MonBouton", name:"bouton2"}]
});
```

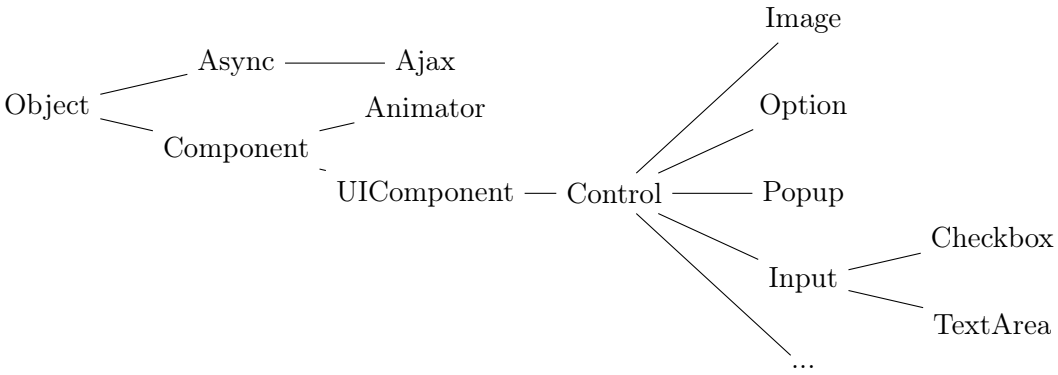
Dans ce cas-ci, on commence par décrire un nouvel objet bouton avec son fonctionnement personnalisé puis en l'étend avec la fonction *enyo.kind* et ainsi pouvoir le réutiliser dans une nouvelle description de composant.

2.7 Hiérarchie des objets

On peut considérer la plupart objets Enyo dans la hiérarchie comme des noeuds HTML, en effet, lorsque l'on construit son application, la forme arborescente de l'arbre se traduit particulièrement bien sous une forme HTML.

Ainsi, un *Button* Enyo est lui-même transformé en un noeud "<button>" à l'instanciation. Il est donc aisé de se représenter le rendu final d'une application.

Voici un aperçu de l'arbre des composants d'Enyo :



L'arbre complet peut-être trouvé en annexe (6.1).

La plupart des objets héritent de la classe *Control*. En effet, c'est à partir de celle-ci que l'on obtient des fonctions d'affichage et la génération d'un noeud html. Les objets parents de *Control*, tel que *Component* ou *Object* ne servent, en pratique, qu'à établir une logique dans le code (création de temporisateurs, calculs de valeurs, etc.). On peut donc tout à fait se passer de ceux-ci lors de l'utilisation en OCaml [4].

2.8 Bibliothèques additionnelles

Enyo possède un ensemble de bibliothèques supplémentaire apportant du contenu à l'API de base. On peut par exemple utiliser la bibliothèque Onyx pour fournir une interface graphique supérieure ou encore Layout pour organiser ses composants de la manière souhaitée.

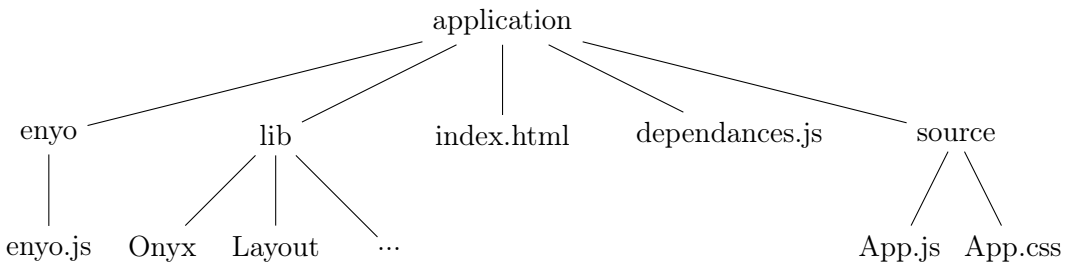
Ces objets viennent généralement se greffer à la hiérarchie existante. Par exemple, "Onyx.Input" hérite de "Control".

Pour pouvoir utiliser cette extension, il est nécessaire de l'indiquer à l'application. Il faut faire appel à *enyo.depends* sur la liste des bibliothèques à lier. Par exemple, si l'on souhaite utiliser des objets d'"Onyx" et de "Layout", il faut procéder ainsi :

```
1   enyo.depends{ "$lib/Onyx", "$lib/Layout"}
```

Il est bien sûr requis de fournir le dossier contenant ces extensions au même niveau que le dossier "enyo". Voici une arborescence classique d'une application Enyo :

4. La logique de l'application étant gérée par le programme OCaml



2.8.1 Onyx

La bibliothèque Onyx enrichit l'application en apportant un ensemble de style prédéfinis pour des objets de bases. L'objectif étant d'obtenir une interface graphique correcte sans avoir à passer du temps sur le style de l'application.

2.8.2 Canvas

Cette extension gère la partie canvas d'HTML5 d'Enyo. Elle définit des objets pour simplifier le code JavaScript et ainsi s'abstraire légèrement de la couche basse d'HTML5.

Cependant, cette bibliothèque a ses limites et l'intérêt décroit à mesure que l'application se complexifie. Il est parfois inévitable de recourir aux fonctions de bases de manipulation de canvas pour obtenir le résultat escompté.

2.8.3 Layout

Permet de disposer les composants de l'application avec flexibilité. L'usage de cette bibliothèque n'est malheureusement pas très intuitive, elle nécessite exemples et lecture de la documentation présente sur le site d'Enyo.

Chapitre 3

Interfaçage d'Enyo en OCaml : Nouveau plan

3.1 Interopérabilité entre JavaScript et OCaml

Pour une adaptation en OCaml d'Enyo, il est nécessaire d'établir une communication entre les deux langages. Cela présente une première difficulté étant donné le peu de points communs existants. Deux projets ont été développés sur le sujet : Le premier "OBrowser"[1], implémente une machine virtuelle interprétant le bytecode OCaml en JavaScript au runtime. Le second "Js_of_ocaml", compile le bytecode pour générer un fichier Javascript qui pourra être ajouté dans une page web.

Le problème principal des deux projets réside dans la représentation des valeurs d'un monde à l'autre. Pour chacun des cas, il a fallu trouver un moyen efficace de pouvoir transmettre et interpréter des valeurs tout en assurant le typage. Cela est réalisé par l'analyse du bytecode qui en redéfinissant les blocs OCaml en valeur JavaScript permettent leur utilisation dans ce monde-ci et inversement pour les valeurs JavaScript traduies en blocs. Fournissant une sur-couche d'API JavaScript, le programmeur OCaml est ainsi à même d'utiliser des fonctions JavaScript au sein de son programme en pouvant s'assurer du typage. L'utilisation de fonctions OCaml en JavaScript est également possible.

Le problème de l'interopérabilité avec Enyo suit les mêmes contraintes : Le dynamisme de JavaScript empêche une traduction directe. Il est donc intéressant d'utiliser l'un de ces outils afin de nous fournir une couche haute pour l'implémentation de ces valeurs.

3.2 Js_of_ocaml : Présentation des outils

Le choix s'est porté sur Js_of_ocaml. Celui-ci fournit une représentation assez complète des objets JavaScript et une manipulation aisée des champs.

Une telle liberté peut cependant être source d'erreurs, venant en particulier du fait que nous utilisons certaines fonctions non-typés. L'intérêt se limite ainsi à l'interopérabilité et nous nous assurerons du typage lors de la définition du modèle.

Nous utiliserons en particulier le module Js.Unsafe qui contient des fonctions assez bas niveau de l'API "Js_of_ocaml" pour nous permettre les libertés dont on a besoin pour créer nos objets. Nous nous intéresserons en particulier aux fonctions suivantes :

```
(*Type haut permettant de contenir l'ensemble des valeurs
    JavaScript possibles*)
type any

(*Fonction d'accession aux variables JavaScript
  => variable "monobj", renvoie l'objet "monobj" defini dans l'
      environnement JavaScript*)
val variable : string -> 'a

//Coercition d'une valeur vers le type haut
val inject : 'a -> any

(*Fonctions d'accession et de modification de champ des objets
    JavaScript*)
val get : 'a -> 'b -> 'c
val set : 'a -> 'b -> 'c -> unit

(*Appel de fonction JavaScript avec un tableau d'argument
  => fun_call (variable "alert") [| inject 42 |]
  = affiche 42*)
val fun_call : 'a -> any array -> 'b

(*Appel d'une methode d'un objet JavaScript
  => let unsafe_str = meth_call unObj "toString" [| |]*)
val meth_call : 'a -> string -> any array -> 'b

(*Creation d'un objet JavaScript*)
val new_obj : 'a -> any array -> 'b
```

L'usage de ces fonctions n'étant pas sûr, il va être indispensable de "recouvrir" ces appels à l'aide à notre système de type.

Lorsque l'on passe des arguments aux appels de méthodes, il est nécessaire des les transcrire en valeurs JavaScript et inversements pour récupérer les valeurs retournées.

La traduction de ces valeurs s'opérent grâce aux fonctions de traductions étant elles typées.

```
val bool : bool -> bool t
val to_bool : bool t -> bool
val string : string -> string t
val to_string : string t -> string
val array : 'a array -> 'a js_array t
val to_array : 'a js_array t -> 'a array
```

```
(* ... *)
```

Les traductions de ces valeurs s'effectuent par copie. La modification d'une chaine de caractères dans le monde OCaml n'aura donc pas d'effet sur l'objet *String* instancié.

Il faut aussi s'assurer de la sécurité des appels des fonctions ou méthodes, par exemple, pour ne pas générer d'erreurs lors de l'ajout de propriétés dans un objet. Si l'on veut ajouter un champ de type string dans un objet JavaScript, il faudra faire appel à la fonction *string* qui se chargera de l'instanciation d'un objet JavaScript *String*.

```
let unObj = newobj (variable "Object") [||] in

(* Correct *)
set unObj "couleur" (string "vert");
(* Leve une erreur au runtime *)
set unObj "couleur" "rouge";

get unObj "length"
```

3.3 Typage OCaml d'Enyo

3.3.1 Typage direct

Bien que *Js_of_ocaml* nous permette d'approcher un typage correct de valeurs, nous ne souhaitons pas que l'utilisateur ait à manipuler sa bibliothèque. Il faut alors se munir d'un ensemble de valeurs dont on est susceptible de passer à JavaScript et/ou de les récupérer afin d'assurer la sécurité du typage.

```
type js_value = Int of int | Float of float
              | Bool of bool | String of string
              | Array of js_value list
              | Component of any_id obj | Dom_node of dom_node
```

Notes :

- J'ai préféré représenter le tableau JavaScript sous forme de liste. Sa manipulation étant plus similaire que celle d'un tableau OCaml
- Le *Dom_node* et le *Component* seront repris plus tard

Ce type nous permettra également de savoir quel type de conversion est nécessaire selon la valeur passée. Par exemple, si le champ contient un type *String v*, nous savons qu'il faudra appliquer la fonction *Js.string* de *js_of_ocaml* pour correctement la transcrire.

3.3.2 Types fantômes et sous-typage

Rappel : Un type fantôme est un type paramétrique dont au moins un paramètre n'est pas utilisé dans sa définition.

```
type +'a kind
type +'a obj
```

Notes :

- *kind* représente la description de l'objet et *obj* l'objet JavaScript correspondant à la génération puis à l'instanciation Enyo. Le sujet est plus amplement abordé dans les sections suivantes.
- La variance sera nécessaire pour l'usage combiné aux variants.

L'utilisation de types fantôme s'explique ici par la représentation équivalente des données décrivant les composants Enyo. Nous souhaitons cependant établir leur différenciation. Les composants ne partageant pas tous les même méthodes, leur abstraction permet d'assurer un typage correct.

Combinés aux variants polymorphes, on peut profiter de leur sous-typage. Pour la librairie, un ensemble d'identifiants a été défini [1].

```
type any_id = [ `CONTROL | `INPUT | `BUTTON ]
```

Avec un typage simple, si nous avions voulu définir une méthode s'appliquant à chaque objet, il aurait fallu la redéfinir pour chaque type et générer des noms de méthodes supplémentaires. Or, le corps de la méthode aurait été le même. Grâce au sous-typage, nous profitons d'un partage des noms de méthodes tout en s'assurant de la validité du typage en plus de synthétiser la définition des composants.

Ainsi, des méthodes *getContent* s'appliquant à un ensemble spécifique de composants Enyo peuvent s'écrire :

```
val meth1: [< `CONTROL | `INPUT | `BUTTON ] obj -> unit
val meth2: [< `INPUT | `BUTTON ] obj -> int
```

3.4 Description en OCaml des déclarations d'Enyo

3.4.1 Modèlisation

Nous avons décrit dans la partie consacrée à la présentation d'Enyo, les trois grandes phases, à savoir : La description, la génération et l'instanciation.

Le résultat de la description de l'objet permettra à Enyo d'établir la génération de cet objet en lui fournissant méthodes et initialisation d'attributs. L'instanciation de cet objet permettra l'appel des méthodes qu'Enyo lui a fourni. Ainsi, il est possible de décrire un objet mais il faudra le générer puis l'instancier pour être à même de faire des appels de méthodes sur celui-ci.

Nous représentons la description avec le type *kind* et l'objet instancié par le type *obj*.

Nous nous intéresserons ici à la description de l'objet et à son implémentation en OCaml.

1. Nous n'en prendrons que trois pour l'exemple

Un composant Enyo étant défini par des propriétés, méthodes et sous-composants, il a fallu se munir d'une structure les regroupant.

```
type +'a kind = {id:string;
                 components: any_id kind list;
                 prop_list : (string * js_value) list;
                 handler_list:handler list
                }
```

Note : le champ ID sera utilisé pour l'instrospection (voir 3.5.2).

Ainsi, nous avons 3 différentes listes :

- Une liste de description de sous-composants *components* similaire à une implémentation en Enyo.
- Une liste de propriété prenant un couple composé du nom du champ ainsi que de la valeur associée.
- La liste des fonctions de traitement d'événements que nous étudierons plus tard.

Contrairement à Enyo, la définition de nouvelles méthodes n'est pas intéressante pour le programmeur qui va plutôt souhaiter définir ses propres fonctions et son modèle de calcul en OCaml. Cependant, il est nécessaire de définir les fonctions de traitements d'événements qui permettront d'établir la logique de l'interface graphique et de l'appel du modèle de calcul de l'application.

Ainsi défini, l'utilisateur doit choisir les attributs de l'objet (texte du bouton, valeur initiale d'un champ de saisie, etc.). Le nombre d'arguments peut croitre de manière assez importante et devenir vite génant pour l'utilisateur. C'est pourquoi, en s'inspirant des modules d'interfaces graphique d'OCaml (Tk, lablGtk), il a été intéressant de procéder par arguments optionnels. L'utilisateur choisi à partir d'un ensemble d'arguments, ceux qu'il lui conviennent et génère ainsi la description.

Nous obtenons un type comme celui-ci pour un constructeur d'objet :

```
val input:
          ?components:any_id kind list
          -> ?value:string
          -> ?content:string
          -> ?name:string
          (* ... *)
          -> ?ontap:([`INPUT] obj -> any_id obj -> [`GESTURE]
              obj -> bool)
          -> unit -> [>`INPUT] kind
```

A l'usage, le programmeur pourra obtenir un code clair et concis :

```
let mon_champ = input ~value:"Bonjour" ()
```

Le corps de la création de description est relativement simple. On spécifie le type du composant en ajoutant la propriété *kind* et la valeur correspondante au composant en cours de création. Il suffit ensuite de tester pour chaque paramètre optionnel son existance et de l'ajouter si il est définit. On retourne ensuite la structure munie des listes.

```
let input
        ?(components=[])
        ?value
        ?content
        ?name
        (*...*)
        ?ontap
        () =
  let prop_list= ref [("kind", String "Input")]
  and handler_list= ref [] in
  (match value with
      Some v ->
        prop_list := ("value",String v)::!prop_list
    | None -> ());
  (match content with
      Some v ->
        prop_list := ("content",String v)::!prop_list
    | None -> ());
 (match name with
      Some v ->
        prop_list := ("name",String v)::!prop_list
    | None -> ());
 (*...*)
 (match ontap with
      Some v ->
        handler_list := Handler ("ontap",v)::!handler_list
    | None -> ());
 {id="INPUT"; components=components;
 prop_list=(!prop_list); handler_list=(!handler_list)}
```

3.4.2 Propriétés

Les propriétés déclarées à la construction de la description sont toutes de type *Published* comme le veut la spécification Enyo (voir 2). Il faut donc leur fournir des méthodes d'accessions et de modifications. Je n'ai pas jugé utile de permettre la définition des méthodes de traitement lors de la modification de valeur (*<valueName>Changed*) : il n'est pas utile pour le programmeur d'avoir ce genre de fonctions, si un traitement est à faire lors du changement d'un champ, le programmeur peut le réaliser au même moment qu'à l'appel du modificateur.

En ce qui concerne les méthodes *get* et *set*, leur implémentation reste très proche de celle de *js_ of_ ocaml* à ceci près qu'il faut veiller à traduire les valeurs selon le sens de communication des langages. Ainsi, pour les propriétés, leurs barrière d'abstraction différeront uniquement selon le type de la propriété. Un exemple de ces méthodes avec celles de la propriété *content* de l'objet *Control* :

```
 let getContent this () =
       let value = meth_call this "getContent" [||] in
              to_string value
```

```
let setContent this chaine1 =
      let _ = meth_call this "setContent" [|inject (string
          chaine1)|] in
              ()
```

3.4.3 Événements

Les événements ont la particularité d'être définis au niveau de la description mais agiront sur les objets instanciés au runtime. Dans Enyo, la spécification des événements et de leur fonctions de traitement gardent toujours la même forme.

```
1  fonctionTraitement: function(emetteur, evenement){
2                          //traitement
3                          return bool; }
```

La fonction attachée prend en paramètres l'émetteur de cet événement, c'est-à-dire, le composant qui a genéré ou propagé celui-ci, et l'objet événement lui-même. Celui-ci retourne un booléen spécifiant si la propagation doit être effectué. (voir : 2.4 pour plus de détails)

Pour adapter l'événement en OCaml, j'ai choisi ce type :

```
type any_event = [`GESTURE (* |  Ensemble des evenements *) ]
type handler = Handler of string * (any_id obj -> any_id obj
   -> any_event obj -> bool)
```

Note : nous utilisons également un type fantome pour les événements.

La gestion de l'événement est donc représentée par un couple contenant le nom de l'événement et la fonction de traitement associée avec en premier argument le *this* de la méthode.

Nous utilisons pour réaliser cela, la fonction *wrap_meth_callback* de *Js_of_ocaml* :

```
type (-'a, +'b) meth_callback
(* Fonction prenant en premier argument l'objet representant le
    "this"
   et en arguments suivants les differents parametres que la
   fonction doit prendre *)
val wrap_meth_callback : ('a -> 'b -> 'c) -> ('a, 'b -> 'c)
    meth_callback
```

Cette fonction "attache" à l'objet une méthode qui va se charger d'appeller la fonction OCaml en transmettant d'un monde à l'autre les arguments nécessaires.

En informant la signature dans chaque constructeur, il devient aisé de constituer un typage correct. Nous pouvons ainsi résoudre le *this* selon l'objet en cours de construction et l'événement attendu. L'émetteur ne peut cependant pas être résolu, nous y reviendrons plus loin.

Pour un champ de saisie, nous attendrons un argument de ce type :

```
val input:
  (* ... *)
  -> ?ontap:([`BUTTON] obj -> any_id obj -> [`GESTURE] obj ->
      bool)
  -> unit -> [>`BUTTON] kind
```

A l'utilisation, nous pourrons alors utiliser les méthodes de "button" sans causer de conflit de type puisqu'employé dans un contexte correct :

```
let traitement_bouton this emetteur evenement =
   setContent this "Nouveau_contenu";
   setDisabled this true;
   true
(*
val traitement_bouton :
  [< `BUTTON | `CHECKBOX | `INPUT ] obj -> 'a -> 'b -> bool
*)

let mon_bouton = button ~content:"contenu" ~ontap:
    traitement_bouton ()
```

Ici, l'utilisation du "this" dans la fonction de traitement sous-type la fonction qui est acceptable pour le type attendu par "ontap".

J'ai défini les événements de la même manière que les objets. Il n'est, en revanche, pas nécessaire de partager les méthodes appliquables sur les événements puisqu'ils ne sont pas liés entre eux. Il est cependant nécessaire d'affecter à chaque type d'événements des fonctions d'accesseurs pour récupérer les informations contenus.

Par exemple, pour l'événement souris, rebaptisé "gesture" pour se conformer à Enyo, les accesseurs, stipulés dans la définition IDL W3C, sont définis comme suit :

```
val gesture_screenX : [`GESTURE] obj-> int
val gesture_screenY : [`GESTURE] obj-> int
val gesture_identifier : [`GESTURE] obj-> int
val gesture_detail : [`GESTURE] obj-> int
(* ... *)
```

Á l'instanciation d'un objet pourvu d'un événement, il est nécessaire de passer l'événement traité dans un objet contenu dans le champ *handlers* de celui-ci. Ainsi, le *mon_ bouton* définit plus haut, aura cette forme en Javascript :

```
1  {kind:"Button", handlers:{ontap:"tap"},
2    tap:function(){ return callback_caml(this) /* gere par
         js_of_ocaml */ }}
```

3.5 Génération et Instanciation

3.5.1 Procédure

La section précédente nous a permis d'obtenir une description des composants de l'application grâce à leurs constructeurs retournant les modélisation OCaml de ces objets JavaScript.

```
let app = let mon_bouton = bouton ~content:"Valider" in
    control ~components:[mon_bouton] ()
```

Il faut désormais pouvoir générer puis instancier l'application correspondante. Pour rappel, la génération correspond à l'appel de la fonction *enyo.kind* sur un objet pour l'étendre à l'environnement, tandis que la dernière phase instancie cet objet.

```
1  //description
2  var description = {kind:"Control", name:"App", content:"
       Bonjour"};
3  //generation
4  enyo.kind(description);
5  //instanciation
6  var app = new App()
```

La première étape sera de transformer la description représenté en OCaml en objet JavaScript.

Pour procéder, il va falloir créer un objet JavaScript vide et y ajouter propriétés et fonctions de traitement définies. Voici une version simplifié de la fonction effectuant ce traitement.

```
let rec obj_of_kind kind =
    let js_obj = new_obj (variable "Object") [|||] in
        ajouteProprietes js_obj kind;
        ajouteEvenements js_obj kind;
        set "components" array (Array.of_list (List.map
            obj_of_kind js_obj kind));
        js_obj
```

Les fonctions d'ajout se contentent de faire des appels à *Js.Unsafe.set* sur l'objet JavaScript *js_obj* en s'assurant de convertir correctement les valeurs selon les types. Pour les événements, il est nécessaire d'utiliser *wrap_meth_callback* pour lier la fonction OCaml à l'objet JavaScript. Enfin, on effectue une récursion sur tous les composants fils.

On obtient ainsi, une description d'objet Enyo conforme et prête à être générer. On procède avec un appel à *enyo.kind* :

```
let jsobj = obj_of_kind monkind

let _ = fun_call (variable "enyo.kind") [| inject js_obj |]
```

Désormais, l'environnement possède un objet de nom "App" instanciable.

Note : Le nom de l'application est généré si il n'est pas fourni. On supposera qu'"App" est le nom donné lors de la description.

Enfin, on l'instancie :

```
let application = new_obj (variable "App") [||]
```

L'appel à *enyo.kind* et l'instanciation du nouvel objet étant assez concis, les trois étapes sont effectuées dans la même fonction de type :

```
val instanciate : ([< any_id] as 'a) kind -> 'a obj
```

Cette fonction prend donc une description d'un sous-type d'"any_id" et retourne l'instanciation d'un objet de ce même sous-type. Etant instancié, on peut appeler les méthodes définies par Enyo pour ce type d'objet.

A partir d'un objet instancié, les méthodes dont il hérite ou qu'il implémente, définies dans l'API d'Enyo, sont désormais appelables. La fonction d'affichage de l'application peut désormais être appelée si l'objet racine l'implémente.

```
let _ = renderIntoBody app
```

3.5.2 Introspection des objets

La stratégie actuelle de traitement des événements pose un problème pour typer l'émetteur. Pour pallier à celà, on a fourni lors de la construction des objets un champ supplémentaire [2] *_onyo_id* qui va permettre de distinguer le type de l'objet.

Munis de cette propriété, nous sommes capable de typer le composant. C'est le rôle de la fonction *as_a* :

```
val as_a : ([< any_id] as 'a) -> [< any_id] obj -> 'a obj
```

Cette fonction prend un sous-type d'"any_id" ainsi qu'un objet instancié non-typé et retourne celui-ci typé. Si l'objet n'est pas de l'"id" spécifié, une exception est levée.

Avec ceci, nous pouvons typer l'emetteur :

```
let mon_bouton = ~content:"Bouton" ~tap:(fun _ _ _ -> false)
    ()

let traitement this emetteur evenement =
   try
       let bouton_emetteur = as_a `BUTTON emetteur in
       (* bouton_emetteur : [`BUTTON] obj *)
       setDisabled bouton_emetteur true;
       true
   with Bad_kind -> true

let mon_control = ~components:[mon_bouton] ~ontap:traitement
    ()
```

2. Le champ kind étant effacé à l'instanciation Enyo, il n'est pas possible d'en faire usage

Chapitre 4

Langage de Description d'Interface (IDL)

Enyo fournit une API suffisamment grande (une centaine d'objet environ) pour qu'il devienne rébarbatif d'implémenter ceci à la main. L'implémentation d'un IDL s'est donc révélé être une bonne option pour pouvoir représenter facilement la hiéarchie de composants.

OCaml étant un outil tout indiqué pour ce type de travail, je n'ai pas eu à faire de parallèle avec d'autres langages (mis à part quelques scripts Perl pour parser la hiérarchie).

En premier lieu, il convient de présenter les structures choisies pour abriter les objets. Puis, j'expliquerai brièvement la génération du code et les problèmes rencontrés.

4.1 Interface des objets

Un objet Enyo présente 3 grandes parties distinctes :
- Les méthodes,
- les propriétés,
- les événements qu'il est capable de "capturer".

Pour les méthodes et les propriétés, il est nécessaire de fournir le type à l'IDL pour assurer à OCaml la sûreté du typage.

Les événements, quant à eux, possèdent un lien vers le type d'événement et les champs qu'il faut préciser.

```
type values_rep =  String | Int | Float | Bool | Unit
                 | Component | Array of values_rep | Dom_node

type method_rep = Method of string * values_rep list

type attributes_rep = Attribute of string * values_rep * bool (*
    bool => Generate propChanged function? *)
```

```
type event_rep = Event of string * attributes_rep list

type handler_rep = Handler of string * event_rep

type object_rep = Type of string * method_rep list *
    attributes_rep list * handler_rep list
                | Type_gen of object_rep * string * method_rep
                    list * attributes_rep list * handler_rep
                   list
                (* Le deuxieme element de la somme n'a plus de
                    reel utilite, mais n'ayant pas eu le temps
                de retravailler le code, il est a ignorer *)
```

J'ai donc choisi de représenter l'objet Enyo par :

- Son nom Enyo,
- une liste de représentation de méthodes,
- une liste de représentation de propriétés [1],
- une liste des noms d'événements que ce type d'objet est capable de traiter.

On obtient ainsi pour l'objet "Control" d'Enyo, la représentation suivante :

```
Type ("Control",
      [
        Method ("render", [Unit; Unit]);
        Method ("rendered", [Unit; Unit]);
        Method ("renderInto", [Dom_node; Unit]);
        Method ("setBounds", [Int; Int; Int; Int; Unit]);
        Method ("show", [Bool; Unit]);
        Method ("write", [Unit; Unit]);
        (*...*)
      ],
      [
        Attribute ("style", String, true);
        Attribute ("content", String, true);
        Attribute ("allowHtml", Bool, false);
        Attribute ("src", String, true);
        (*...*)
      ],
      [Handler ("ontap", Gesture_event._gesture)]
)
```

En étudiant l'API, j'ai pu transcrire les méthodes et propriétés en m'efforçant de typer au plus juste. En revanche, quelques libertés ont été prises :

- Je n'ai pas jugé utile l'ajout de méthodes trop polymorphes telle que : *setProperty(name, value)*.
- Par manque de documentation de l'API sur certaines, il a été quelques fois nécessaire de compléter moi-même les signatures de méthodes en

1. L'utilisation du booléen pour savoir si une fonction de type <value>Changed doit être implémenté est temporairement abandonné, pour les raisons spécifiées dans la partie de gestion des propriétés **??**

effectuant des tests en restant toutefois rigoureux.
- J'ai omis d'inclure les méthodes protégées, pour la simple raison que nous ne traitons pas l'extension objet, ici.

Je complète ainsi chaque objet et l'inclus dans un arbre d'"object_rep" recréant ainsi la hiérarchie Enyo nécessaire permettant à chaque composant d'hériter des méthodes et attributs. La hiérarchie désormais complète, la génération de code peut débuter.

4.2 Générateur de code

L'objectif de cette génération de code est d'obtenir en sortie un module complet basé sur la représentation IDL avec le modèle choisi 3.

La difficulté de cette étape a été le parcours de l'arbre pour résoudre les dépendances :

- Un objet doit transmettre à ses fils toutes les propriétés et "handlers" d'événements qu'il possède afin que leurs constructeurs soient en mesure de les proposer à l'utilisateur
- En revanche, les méthodes auront besoin de la liste de tous les identifiants de sa descendance pour pouvoir spécifier les méthodes et permettre aux enfants de les appeller

Une simple récursion sur l'arbre permet l'obtention d'un résultat satisfaisant.

Plus tard, j'ai constaté qu'un cas spécial ne permettait pas un parcours aussi banal : lorsque deux objets définissent des méthodes ou proprietés de même noms sur des branches différentes de l'arbre, il y a collision de noms et il est impossible de factoriser simplement les dépendances [2].

Afin de résoudre ceci, je réalise des parcours préalables en remplissant des tables de hachage contenant en clé le nom de méthodes et de proprietés avec en valeur leurs listes des dépendances. J'applique ensuite la récursion, en consultant la table de hachage pour résoudre les incohérences.

Note : On aurait également pu "tagger" les différentes incohérences au moment de la représentation de l'objet en spécifiant que cette méthode ou propriété est partagée avec un autre objet. Plusieurs solutions sont possibles...

L'arbre de résolutions de dépendances construit, il ne reste plus qu'à réaliser les impressions pour pouvoir générer le module OCaml prêt à l'emploi.

2. J'omets le fait que deux méthodes de même nom peuvent être de types différents, ce qu'il faudrait traiter si le cas était présent

Chapitre 5

Conclusion

A travers ce projet, nous avons pu établir un certain nombre de choses. Tout d'abord, nous avons pu établir que le style de programmation d'Enyo nous offrait un nombre important de possibilités. Il a fallu étudier ce style, son fonctionnement et en abstraire les concepts afin de définir un modèle intéressant et cohérent.

La scission de la représentation et de l'instanciation de l'objet Enyo sur lequel j'ai basé mon modèle répondent à l'objectif de sûreté que nous nous étions fixé tout en assurant un degré important de liberté vis-à-vis de l'API d'Enyo.

Enfin, la définition d'un IDL a permis une génération fiable de l'implémentation finale mais aussi de permettre de tenir à jour l'API Enyo de manière assez aisée en cas de besoin.

Cependant, le rendu final subit quelques défauts :

- L'utilisation de variants, bien que permettant une bonne modélisation, peut être perçue comme trop complexe pour une prise en main rapide. Cela fournit également des messages d'erreurs trop aggressifs. Cette dernière remarque s'applique aussi pour l'utilisation des paramètres optionnels lors de la construction des objets.
- L'accession aux différents composants instanciés lors de la définition du traitement des événements n'est pas satisfaisante et gagnerait à être améliorée.
- L'IDL ne traite pas certains cas particuliers et reste relativement primaire.

Pour permettre une continuation du projet, il peut-être intéressant de reprendre les problèmes énoncés avec, par exemple, les différentes approches énoncées dans les différentes sections. Pour résumer :

- Réfléchir à une représentation différentes que les variants, quitte à rendre le module plus lourd, par une approche objet ou type simple.
- Pouvoir accéder aux instanciations Enyo par le "kind" de l'objet, en s'assurant de son existence. Par exemple, ainsi :

```
let bouton_obj = getObj button_kind (* 'a kind -> 'a
    obj *)
setContent bouton_obj "Nouvelle valeur"
```

- Il serait bien de compléter la hiérarchie en fournissant des solutions adéquates aux cas particuliers. Ensuite, pouvoir fournir des possibilités d'extension à l'IDL en permettant des définitions internes de composants. Par exemple, être capable de créer un nouvel objet "ChampValidation" possédant deux composants "Input" et "Button" ainsi qu'une fonction de validation.
- Il faudrait réfléchir à une représentation des noeuds du DOM. Est-ce que l'on souhaite passer par "js_of_ocaml", fournir une chaine le représentant, ou encore définir quelques noeuds principaux et brider leur utilisation ?

Sur le plan personnel, ce stage m'a été, je pense, très profitable ; j'ai pu consolider mes bases en OCaml ainsi que mon engouement pour ce langage. Je le dois en bonne partie à l'équipe capable et dynamique dont j'ai eu la chance d'être entouré pendant ces deux mois. De plus, cette expérience au sein d'un laboratoire m'a sérieusement fait réfléchir à une éventuelle formation vers la recherche. Je me satisfait donc d'avoir pu effectuer ce stage dont j'en sors enrichi.

Chapitre 6

Annexes

6.1 Hiérarchie complète Enyo

```
|-- Layout
|   |-- Arranger
|   |   |-- CardArranger
|   |   |   |-- CardSlideInArranger
|   |   |-- CarouselArranger
|   |   |   |-- CollapsingArranger
|   |   |-- LeftRightArranger
|   |       |-- TopBottomArranger
|   |-- BaseLayout
|   |-- FittableLayout
|-- Object
    |-- Async
    |   |-- Ajax
    |   |-- JsonpRequest
    |-- Component
        |-- _AjaxComponent
        |   |-- WebService
        |-- Animator
        |-- DragAvatar
        |-- ScrollMath
        |-- Selection
        |-- Signals
        |-- UiComponent
            |-- canvas.Control
            |   |-- canvas.Image
            |   |-- canvas.Shape
            |       |-- canvas.Circle
            |       |-- canvas.Rectangle
```

```
|       |-- canvas.Text
|-- Control
    |-- Canvas
    |-- FittableColumns
    |-- FittableRows
    |-- FlyweightRepeater
    |-- Group
    |   |-- onyx.RadioGroup
    |-- GroupItem
    |   |-- ToolDecorator
    |       |-- Button
    |       |   |-- onyx.Button
    |       |   |-- onyx.MenuItem
    |       |-- onyx.InputDecorator
    |-- Image
    |-- Input
    |   |-- Checkbox
    |   |   |-- onyx.Checkbox
    |   |-- RichText
    |   |   |-- onyx.RichText
    |   |-- TextArea
    |       |-- onyx.TextArea
    |-- Node
    |-- onyx.Drawer
    |-- onyx.Grabber
    |-- onyx.Groupbox
    |-- onyx.GroupboxHeader
    |-- onyx.Icon
    |   |-- onyx.IconButton
    |-- onyx.Item
    |-- onyx.MoreToolbar
    |-- onyx.ProgressBar
    |   |-- onyx.ProgressButton
    |   |-- onyx.Slider
    |-- onyx.Scrim
    |-- onyx.Spinner
    |-- onyx.ToggleButton
    |-- onyx.Toolbar
    |-- onyx.TooltipDecorator
    |   |-- onyx.MenuDecorator
    |       |-- onyx.PickerDecorator
    |-- Option
    |-- OwnerProxy
    |-- Panels
```

```
|-- Popup
|   |-- onyx.Popup
|       |-- onyx.Menu
|       |   |-- onyx.Picker
|       |       |-- onyx.FlyweightPicker
|       |-- onyx.Tooltip
|-- Repeater
|-- Scroller
|   |-- List
|       |-- PulldownList
|-- ScrollStrategy
|   |-- TouchScrollStrategy
|       |-- TranslateScrollStrategy
|-- ScrollThumb
|-- Select
|-- Slideable
```

6.2 Déploiement d'applications

Le déploiement d'applications sur les différentes plateformes nécessite quelques changements mineurs de l'application en fonction de la cible visée.

Tout d'abord, Enyo fournit un "bootplate" décrivant l'organisation standard d'une application. Ce dossier contient un script "deploy" permettant de préparer l'application au déploiement ainsi que l'analyse du programme et la suppression des fichiers inutilisés.

A partir de là, pour un déploiement sur tablette HP, par exemple, il est nécessaire d'obtenir les outils développeur de la tablette permettant de "packagé" l'application. Pour un déploiement sur téléphone ("android", "iOs", etc.), il est nécessaire de télécharger "PhoneGap" [lien] qui propose ce genre de "packaging". Il est toutefois nécessaire de fournir un noeud *meta* dans le document Html principal de l'application selon la plateforme. Les détails sont accessibles à cette adresse :
`https://github.com/enyojs/enyo/wiki/Platform-Specific-Deployment`

6.3 Exemples de programme

Les exemples d'applications se situent dans le dossier "exemples" à la racine du projet.

On pourra y trouver quelques applications JavaScript ainsi que des jeux utilisant la bibliothèque inspiré des exemples contenu sur le cédérom du manuel "DAOC"[2].

Table des matières

Bibliographie

[1] Benjamin Canou. *Programmation Web Typée*. PhD thesis, Université Pierre et Marie Curie, 2011. `www.pps.univ-paris-diderot.fr/~canou/these.pdf`.

[2] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'Applications avec Objective Caml*. O'Reilly France, 2000. `http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/`.

[3] Développeurs Enyo. *Documentation Enyo 1.0*. HP, 2011. `https://developer.palm.com/content/api/index.html`.

[4] Développeurs Enyo. *Documentation Enyo 2.0*. HP, 2012. `http://enyojs.com/docs`.

[5] Ocsigen Team. *Js_of_ocaml*. Ocsigen, 2010-2012. `http://ocsigen.org/js_of_ocaml/manual/`.