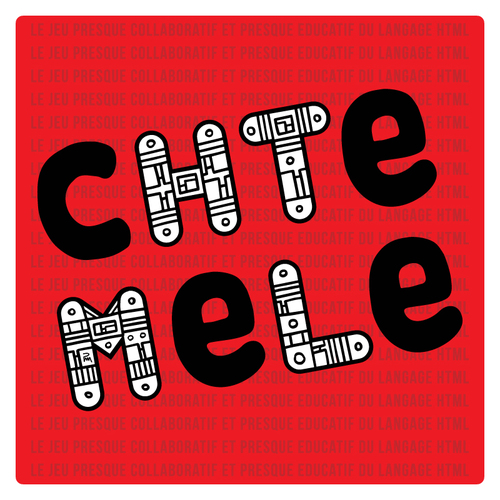
Mémoire – Projet tuteuré

cHTeMeLe Surface

# Projet :

|  |  |
| --- | --- |
| Nom du projet | cHTeMeLe Surface |
| Tuteur | Adélaïde Albouy-Kissy |
| Relation avec une entreprise | Non |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Participants |  |  |  |  |
| Barbara Schiavi | Valentin Mourot | Clément Janisset | Pierre-Rémi  Nouvet-Gire | Volodia Mandaud |



# Remerciement

Tout d’abord, nous tenons à remercier notre tutrice Adélaïde Albouy-Kissi, qui nous a donné l’opportunité de travailler sur la table interactive à travers le jeu de carte cHTeMeLe.

Un grand merci à Mr Pascal Mabille et Mr Olivier Chambon, les créateurs du cHTeMeLe.

Nous sommes très reconnaissants envers Delphine Huguel, pour sa gentillesse, sa disponibilité et son soutien. Merci de son aide précieuse et d’avoir pris le temps de répondre à nos questions.

Enfin, nous remercions l’IUT du Puy-en-Velay et de Clermont-Ferrand, pour l’enseignement fourni et la qualité de la formation qui nous est offerte.

# Résumé du projet

Le cHTeMeLe est un serious game permettant aux débutants d’apprendre le langage web HTML5 tout en s’amusant.

Afin d’adapter le jeu de cartes en jeu sur table interactive, il a tout d’abord fallu modifier les règles. Nous avons également réorganisé les cartes, en en ajoutant et supprimant certaines d’entre elles. Cela rend le jeu plus facile à prendre en main, plus vivant, et plus équilibré.

Nous avions des contraintes à respecter données par notre tutrice : la reconnaissance des cartes par les tags, 3 différents choix de style CSS, les règles syntaxiques du HTML5 et le respect de l’indentation.

Notre application permet de créer une partie de 2 à 4 joueurs pour un nombre de tours personnalisable où les joueurs devront poser successivement des cartes pour former une page HTML. Le code créé par les joueurs est directement affiché dans la zone de visualisation Web et dans la zone de code où celui est automatiquement coloré et indenté. Chaque joueur peut influer sur sa progression et sur celle de ses adversaires à l’aide de cartes spécifiques.

# Table des matières

[Introduction 5](#_Toc382929716)

[Le projet cHTeMeLe Surface 5](#_Toc382929717)

[Pourquoi utiliser la table interactive Surface SUR40 ? 6](#_Toc382929718)

[Présentation du sujet 6](#_Toc382929719)

[Reconnaissance des cartes 6](#_Toc382929720)

[Choix de feuilles de style 6](#_Toc382929721)

[Règles syntaxiques 7](#_Toc382929722)

[Indentation automatique 7](#_Toc382929723)

[Présentation du projet 7](#_Toc382929724)

[Réécriture des règles 7](#_Toc382929725)

[Utilisation de l’interface tactile 9](#_Toc382929726)

[Technologies et appareils utilisés 10](#_Toc382929727)

[La table tactile Samsung SUR40 10](#_Toc382929728)

[Microsoft Surface SDK 12](#_Toc382929729)

[Pixelsense Input Simulator 12](#_Toc382929730)

[Langages : C# - Xaml – HTML – CSS 12](#_Toc382929731)

[Mémoire du développement 13](#_Toc382929732)

[Chapitre 1 : Découverte du SDK 13](#_Toc382929733)

[Chapitre 2 : Création de l’interface 13](#_Toc382929734)

[Chapitre 3 : Moteur de balise 14](#_Toc382929735)

[Chapitre 4 : Reconnaissance de tags et association avec les cartes 14](#_Toc382929736)

[Chapitre 5 : Documentation HTML 15](#_Toc382929737)

[Chapitre 6 : Création zone joueur 15](#_Toc382929738)

[Chapitre 7 : Création carte joueur 15](#_Toc382929739)

[Chapitre 8 : Création zone centrale 16](#_Toc382929740)

[Chapitre 9 : Navigateur internet - AWESOMIUM 16](#_Toc382929741)

[Chapitre 10 : Carte action 17](#_Toc382929742)

[Chapitre 11: Génération du code automatique 18](#_Toc382929743)

[Chapitre 12 : Indentation automatique 20](#_Toc382929744)

[Chapitre 13 : Coloration syntaxique 22](#_Toc382929745)

[V&C- Chapitre 14 : Gestion de la pose de cartes 23](#_Toc382929746)

[V- Chapitre 15 : Charger des fichiers CSS 23](#_Toc382929747)

[V-Chapitre 16 : Implémentation des règles 24](#_Toc382929748)

[Chapitre 17 : Design 24](#_Toc382929749)

[Résultats 25](#_Toc382929750)

[Conclusion et améliorations 26](#_Toc382929751)

[Annexes 27](#_Toc382929752)

# Introduction

## Le projet cHTeMeLe Surface

Créé par Olivier Chambon (Créateur) et Pascal Mabille (Illustrateur) en 2012, cHTeMele est un jeu de société éducatif qui a pour ambition de faire apprendre les bases du langage HTML 5 aux néophytes. Nous rappellerons que le HTML5 (HyperText Markup Language) est un langage de présentation de données créé par le W3C (World Wide Web Consortium) en collaboration avec le WHATWG (Web Hypertext Application Technology Working Group), c’est également un standard utilisé dans la création de sites web.

Disponible depuis le 20 Septembre 2012 sous la licence "Creative Commons CC-BY-NC" (Téléchargement libre et gratuit pour une utilisation non commerciale), cHTeMeLe est un jeu de cartes dont le but est de créer une page HTML correcte. Les joueurs posent des cartes à tour de rôle afin de construire une page HTML et de remporter le plus de points possible.

cHTeMeLe est téléchargeable sous la forme d'un jeu de 112 cartes représentant des balises HTML ainsi que 5 cartes de joueurs à l'image de 5 navigateurs web connus (Chrome, Firefox, Opera, Safari et Internet Explorer). cHTeMeLe est également disponible à la vente en jeu de plateau.

Le projet cHTeMeLe Surface a été créé dans le cadre des projets tuteurés de l’Université D’Auvergne au sein du département Imagerie Numérique du site du Puy-en-Velay. Le jeu cHTeMele étant sous licence Creative Commons, Mme Adélaïde Albouy-Kissi a donc lancé un projet d'adaptation du jeu sur une table interactive. Le but était de créer une application fonctionnant sur la table interactive Samsung SUR40 disponible à l'IUT, mais aussi, et de façon plus générale, sur toutes les tables interactives disposant du SDK Surface et de la capacité à reconnaitre des objets réels.

Le principal intérêt de porter le jeu cHTeMeLe sur une table interactive est de pouvoir avoir un aperçu en temps réel de la page HTML que les joueurs construisent. En effet, dans le jeu initial, les joueurs ne peuvent voir le résultat que si, à la fin de la partie, ils rédigent le code dans un fichier HTML et le lance sur leur navigateur (ce qui est fastidieux mais qui peut aussi paraître compliqué pour le public visé par le jeu : les néophytes). La table interactive apporte aussi un certain confort lors du jeu dans le sens où elle comptera automatiquement les points des joueurs elle permet aux joueurs d'accéder directement à une documentation personnelle et spécialisée pour cHTeMeLe afin de pouvoir renseigner les joueurs lorsqu'ils le désirent.

## Pourquoi utiliser la table interactive Surface SUR40 ?

L’UdA, ayant investi dans des tables interactives Samsung SUR40, souhaite faire développer des applications par les élèves pour d’autres élèves car ils ont pour projet d’en acheter plusieurs afin de les placer dans les salles de TP. Ce sujet nous a été proposé par Mme Adélaïde Albouy-Kissi, fruit de ses propres recherches. Ce sujet concernait le cHTeMeLe, un jeu libre de modification, dont elle nous a proposé l’adaptation sur une table interactive dans le cadre du projet tuteuré.

# Présentation du sujet

Pour développer cette édition tactile de cHTeMeLe, nous étions libres de remodeler le jeu selon nos envies. Cependant nous devions respecter certains critères dans l’application, nous allons vous les expliquer ci-dessous.

## Reconnaissance des cartes

Le jeu se doit de permettre aux joueurs de poser leurs cartes sur la table interactive, celle-ci se chargera de les reconnaître et d’interagir avec. *(Ex : Faire apparaitre des informations supplémentaires autour de la carte, pré-compléter le code HTML en surbrillance, etc)*

## Choix de feuilles de style

L’idée est de proposer aux joueurs plusieurs manières de visualiser ce que donne leur code HTML dans un navigateur. Les sites web utilisent des feuilles de style écrites en CSS (Cascading Style Sheet) associées à leur page HTML pour rendre la page visuellement plus attrayante. cHTeMeLe Surface devait proposer aux joueurs la possibilité de changer à tout moment la feuille de style CSS parmi plusieurs proposées, afin de changer le style visuel de leur page HTML.

## Règles syntaxiques

cHTeMeLe étant un jeu à but éducatif, il fallait qu’il détecte les erreurs dans le code créé par les joueurs ou qu’il les empêche de les faire.

## Indentation automatique

Pour garder un code propre, le jeu doit faire en sorte que l’indentation du code soit respectée.

<html>

<head>

<title>**Ceci est le titre de la page**</title>

</head>

<body>

<h1>**Ceci est une balise sur une ligne.**</h1>

<p>

**Ceci est une balise sur plusieurs lignes.**<br/>

**Il contient des balises qui n'engendrent pas de**

**retours à ligne** <em>**comme celle-ci**</em>**.**

</p>

<div>

<h2>**Exemple**</h2>

</div>

</body>

</html>

Exemple d’indentation de code HTML

Globalement une bonne indentation de code HTML distingue trois types de balises :

* Les éléments multi-lignes : les deux balises ouvrante et fermante occupent chacune une ligne entière et ce qui se trouve entre les deux est indenté au niveau inférieur.

*(Ex : <p>, <div>, <body>, etc.)*

* Les éléments mono-ligne : l’élément entier occupe une ligne entière.

*(Ex : <h1>, <h2>, <title>, etc.)*

* Les éléments inline : l’élément est situé dans un autre élément non-inline. Il n’engendre aucun retour à la ligne.  
  *(Ex : <em>, <br>, <strong>, etc.)*

*Un élément html est composé d’une ou deux balises (une ouvrante, et souvent une seconde fermante), ainsi que d’un contenu qui peut être du texte et/ou d’autres éléments html.*

# Présentation du projet

## Réécriture des règles

À partir de cet énoncé nous avons imaginé une nouvelle version du jeu cHTeMeLe. Nous avons réécrit les règles pour que la prise en main soir la plus intuitive possible.

De plus, dans le jeu original sont présentes de nombreuses cartes « Action » permettant d’entraver la progression de ses adversaires ou de faciliter la nôtre. Beaucoup d’entre elles étaient trop complexes ou trop contraignantes. *(Par exemple l’une d’entre elles demandait aux joueurs de recommencer la partie en conservant le score actuel des joueurs.)*  
Nous avons donc refait toutes ces cartes « action » en diminuant leur nombre et en rendant leur puissance plus homogène.

Selon le jeu original, le joueur doit débuter la partie en posant la carte **<html>** et n’importe quel joueur peut terminer la partie prématurément en posant, s’il l’a, la carte **</html>**, celle-ci lui permettant de remporter un grand nombre de points, ce qui, le plus souvent, permet au joueur de remporter la victoire.  
Nous avons décidé de retirer cette carte du jeu et de rendre le système de score moins chaotique. En effet quand un joueur posait une carte Balise fermante (</balise>), il était gratifié d’un nombre de points en fonction du contenu de la balise. Or l’élément **<html>** englobe toute la page web. Le joueur qui posait la carte **</html>** gagnait un score égal au total des points qu’octroyaient chaque carte posée durant la partie.  
Le nouveau système de score ne récompense plus la fermeture d’un élément html que par un nombre de points fixe, égal au nombre de points qu’octroie la pose de la balise ouvrante de l’élément html.  
De ce fait, il fallait réfléchir à de nouvelles conditions de fin de partie. Les règles originales stipulent que la 10e ligne de code soit également la dernière. Nous avons trouvé cela inadapté et avons décidé de fixer un nombre de tours limité au début de la partie (10 par défaut, mais modifiable). Cela vaut mieux car souvent les joueurs n’ont pas les cartes nécessaires pour bien clôturer le code qu’ils posent. Le jeu original sanctionnait ce fait par une perte de points. Nous avons donc décidé de ne pas limiter le nombre de lignes et de ne pas sanctionner le code incorrect en fin de partie.

Nous avons également changé la liste des balises HTML que le jeu proposait. Cela dit certains éléments HTML ne peuvent être posés qu’à l’intérieur de certains autres, ce qui les rend très difficiles à jouer. C’est pour cette raison que le jeu ne propose pas de créer des listes et des tableaux. Cela dit, quelques cartes restent plus difficiles à jouer, comme certaines cartes attributs *(src, href, title)* car ces attributs ne peuvent pas aller dans tous les éléments html. Ces cartes auront plus de valeur que les autres en termes de score.

Enfin, nous avons préféré augmenter la quantité de code que poseront les joueurs au fil du jeu. Nous leur donnons non plus 6 mais 10 cartes au début de chaque tour et la possibilité de poser autant de cartes html qu’ils le souhaitent. Le code créé par les joueurs sera plus vaste et le jeu deviendra plus dynamique.

## Utilisation de l’interface tactile

La table tactile Samsung SUR40 ne peut reconnaître simultanément qu’un nombre limité de d'objets physiques, et donc de cartes. Pour des raisons d’espace et techniques, nous avons donc décidé de ne pas laisser les cartes posées sur la table tout le long de la partie. Un joueur posera sa carte puis la défaussera une fois que sa pose sera validée (appui sur un bouton sur l’écran de la table). Cela permet de ne pas limiter la taille du code, au vu des dimensions des cartes et des limites techniques de la table.

Pour des raisons de confort, nous devons limiter le jeu à 4 joueurs (un de chaque côté de la table) et non 5 comme dans le jeu original, et encore moins 6 comme il était proposé dans le sujet.

Nous proposons aux joueurs de rentrer eux-mêmes le contenu textuel des éléments html qu’ils posent sur la table, et donc nous retirons du jeu les cartes **Lorem** et **Ipsum** qui servaient de cartes texte. Écrire du texte ne rapporte donc plus aucuns points.

Certains contenus ne sont pas textuels : l’attribut **src** d’un élément **<img>** doit contenir un chemin vers un fichier image. Le jeu proposera une petite banque d’image permettant au joueur de choisir visuellement l’image qu’il veut insérer.

Certains contenus textuels ne devraient pas être totalement libres, sinon ils seraient inutiles. C’est le cas de l’attribut **class** : les classes sont définies dans le code HTML et sont utilisées dans le code CSS. Or le fichier CSS n’est pas accessible aux joueurs. Le jeu proposera donc pour la carte **class** une liste de valeurs possibles qui seront toutes implémentées dans les feuilles de style proposées par cHTeMeLe.

Certaines cartes « Action » sont appelées cartes « Attaque » car elles ciblent un adversaire afin de le désavantager. Le jeu proposera au joueur de choisir sa cible via des boutons sur la table tactile, et l’application gèrera les conséquences de l’utilisation des cartes « Action » automatiquement si possible. Sinon, elle affichera une bulle d’aide aux joueurs pour leur indiquer la marche à suivre.

Au fil des tours, l’interface du jeu tournera pour se mettre dans la direction du joueur actuel. La zone où se trouve le code sera donc toujours de son côté et la visualisation du navigateur en face. Le joueur, son tour venu, aura à sa disposition un bouton pour passer la main au joueur suivant.

Chaque joueur aura à sa disposition une petite interface servant de documentation sur les cartes du jeu et un lien vers les règles. Se situera à côté sa « carte joueur », constituée d’une interface affichant son nom, son score total, les effets des cartes « Action » qui sont actives sur lui et des informations sur son tour précédent (nommé « dernière combinaison »).

# Technologies et appareils utilisés

## La table tactile Samsung SUR40

### Caractéristiques techniques

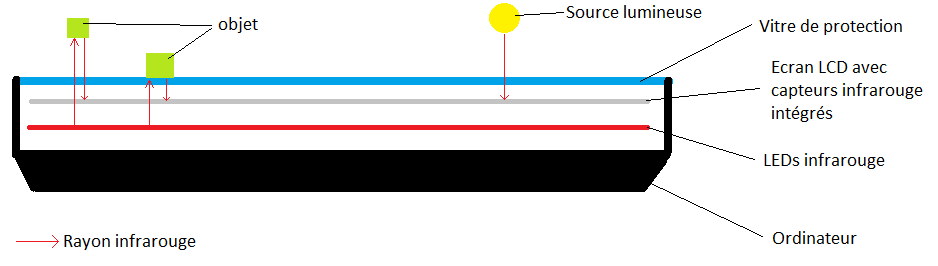
|  |  |
| --- | --- |
| Taille de l’écran | 40 pouces |
| Processeur | Athlon II X2 Dual-Core |
| Mémoire interne | RAM 4Go  HDD 320Go |
| Processeur graphique | AMD Radeon HD 6750 |
| Interfaces USB | 4 |
| Système d’exploitation | Windows 7 professionnel |

La table SUR40 est recouverte d’un verre Gorilla Glass, qui est chimiquement renforcé. La technologie utilisée pour capter les entrées tactiles est basée sur un système infra-rouge. Elle peut détecter principalement les touchers, les survols et les objets réels.

### Reconnaissance des entrées

La table utilise la technologie PixelSense pour récupérer et reconnaître les points de contact. Cette technologie consiste à envoyer des rayons infrarouges grâce à des émetteurs situées en dessous de l'écran, et récupère les rayons infrarouges renvoyés par des objets proches de la surface. Plus un objet est proche, plus il renvoie de rayons infrarouges. Les rayons infrarouges sont ensuite redirigés vers des capteurs qui transforment les informations lumineuses en signaux électriques, puis en image dans des niveaux de gris. Cette image est ensuite traitée afin d’éliminer le bruit parasite et de reconnaître les éléments de contact. Ce système dispose de plusieurs avantages : il permet d'une part de reconnaître jusqu'à 52 points de contact simultanément (qu'ils soient en contact avec la surface ou qu'ils survolent simplement l'écran), mais aussi de reconnaître leur forme, leur taille et leur orientation.

Mais en contrepartie, la technologie PixelSense dispose d'un inconvénient majeur : la reconnaissance se faisant uniquement sur l'acquisition de rayons infrarouges, la table devient donc extrêmement sensible à certaines sources lumineuses émettant des rayons infrarouges. Or la table ne possède aucun moyen de vérifier si la source de rayon infrarouge captée est interne ou parasite. Par conséquent la table prendra également en compte les sources lumineuses externes (comme une ampoule ou la lumière du soleil) et les interprétera comme des points de contact.

******

Schema de fonctionnement de la technologie PixelSense

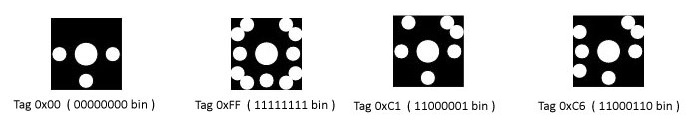
La technologie PixelSense classe les points de contact en 3 types distincts:

### Les fingers

Les fingers représentent une interaction sur la table avec un doigt (ou une petite surface tel qu'un stylo). Ils remplacent en général les interactions avec la souris et permettent donc d'interagir et de naviguer sous Windows tout au long de l’utilisation.

### Les tags

Les tags sont des petites images carrées en noir et blanc stockant de l'information (afin de pouvoir être interprétés sur l'image en niveau de gris. En effet : noir = infrarouges absorbés / blanc = infrarouges réfléchis). Ces petites vignettes peuvent être interprétées, une fois posées, par la table tactile qui est alors en mesure de restituer l'information qu'elles contiennent. Il existe toute sorte de tags permettant de stocker plus ou moins de données. Les tags qui sont reconnus nativement par la technologie PixelSense sont les Bytes Tags pouvant stocker une seule valeur (allant de 0 à 255).



Exemple de tags (source: msdn.microsoft.com)

### Les blobs

Les blobs représentent toutes les autres interactions que PixelSense ne peut pas classer dans les 2 types précédents, c'est à dire les gros points de contact (paume de la main, coude, feuille, etc...).

## Microsoft Surface SDK

Il s'agit d'un ensemble d'outils et de librairies permettant de développer des applications Surface. Elle contient toutes les fonctions spécifiques à l'utilisation de la table interactive.

## Pixelsense Input Simulator

Le logiciel Input Simulator, permet à n’importe quel PC fonctionnant sous Windows Vista et 7 de simuler tout les types d’entrées, les fingers, les tags, et les blobs. Ce simulateur permet grâce à un clavier et une souris de positionner un ou plusieurs contacts sur votre écran comme s’il s’agissait d’un écran tactile.

## Langages : C# - Xaml – HTML – CSS

Une application à destination d'une table interactive doit être codée en C# (langage de programmation) accompagné de XAML (langage de présentation), mais cette application utilise également deux autres langages de présentation (HTML et CSS) utilisé pour rédiger la documentation ainsi que le rendu de la page de code.

# Mémoire du développement

## Chapitre 1 : Découverte du SDK

La découverte du SDK s'est faite par application de tutoriels délivrés par Delphine Huguel, d’installation et de mise en route de projet avec le SDK Microsoft Surface.

Des tests ont étés réalisés avec des projets déjà compilés afin de comprendre son fonctionnement et découvrir la librairie Surface 2.0.

 En effet des exemples d’application au format WPF sont fournis même si la plupart d’entre eux ne concernait pas notre projet.

Le SDK Surface 2.0 comprend les logiciels :  
- Input simulator, qui permet donc de simuler tous les types d’entrée dit précédemment que la table reconnait.

- Input Visualizer, qui lui permet de visualiser le suivit d’une interaction avec la table. Cela permet avant tout d’afficher une trace suivant une entrée de type finger par exemple et de voir son point de départ et d’arrivé.

Une fois le SDK installé, Microsoft Visual permet de créer des projets Surface de type WPF en utilisant au minimum une version du Framework en 4.0, ce type de projet utilise donc le langage C# et XAML.  
  
La plupart de nos informations ont été glanés sur des forums de C# et d’autres forums plus restreints sur le développement Surface. Comme cela a été le cas pour l’intégration du navigateur Chromium.

## Chapitre 2 : Création de l’interface

Pour l’interface, nous avons choisi d’attribuer une zone de joueur pour chaque participant. Cette zone se trouvera en face de chacun des joueurs et leur permettra de s’informer sur les règles du jeu, le rôle de chaque carte ainsi que leur score actuel et s’ils sont touchés par des cartes Effets. La carte du joueur possédera aussi un bouton « Au suivant » afin de finir son tour et passer au joueur suivant.

Nous avons donc créé un ScatterView par joueur que nous avons divisé en trois parties à l’aide de l’attribut Grid (permettant de créer un tableau découpant l’objet). Une première case occupée par le bouton « Au suivant », une deuxième case qui contient un navigateur web permettant d’afficher la documentation et les règles du jeu, et enfin une dernière case où se trouve la carte du joueur. Afin que chaque joueur possède sa propre zone et qu’il n’y ait aucun anti-jeu (par exemple cacher le plateau de jeu avec les emplacements), nous avons décidé de fixer ces zones joueur.

L’espace au milieu de la table tactile est réservé à l’affichage de la page de code et de son rendu sur navigateur. Ces parties sont rafraîchies à chaque événement afin de garder le code et le rendu de la page web constamment à jour.

Afin de contenir ces deux parties, nous avons créé un autre ScatterView que nous avons divisé en 3 parties : une zone d’affichage du code, un navigateur web, et une petite bande centrale pour poser les tas de cartes. Celui-ci est également fixé pour des raisons pratiques et une nouvelle fois d’anti-jeu (par exemple un joueur aurait pu garder la zone de jeu vers lui et ainsi empêcher les autres joueurs de jouer correctement). Cet espace central subit également un traitement tout particulier : il s’oriente automatiquement vers le joueur dont c’est le tour pour qu’il lui soit plus facilement lisible. De plus nous assurons que la zone où s’affiche le code soit toujours au plus proche du joueur, étant donné que le code y est écrit petit.

Dans le souci de créer une application générique (compatible avec plusieurs tables n’ayant pas les même dimensions), toutes les zones évoquées précédemment ont une taille relative.

Nous avions également pensé à intégrer un clavier pour chaque joueur leur permettant de chercher directement dans la documentation et de pouvoir remplir les balises acceptant du texte. Après mûre réflexion, laisser chercher les joueurs dans la documentation grâce à un clavier demande une connaissance minimale des balises (au moins les noms), ce qui peut être dur pour le public que vise le jeu. Ainsi, nous avons utilisé le clavier de base de Windows pour entrer du texte dans les balises ; nous laissons également les joueurs naviguer dans la documentation à l’aide d’un menu.

## Chapitre 3 : Moteur de balise

* Structure de données

## Chapitre 4 : Reconnaissance de tags et association avec les cartes

Pour la version tactile de cHTeMeLe, le sujet nous imposait que les cartes du jeu soient reconnues par la table. Pour cela, une seule solution était possible: utiliser les tags fournis par Microsoft. Comme dit précédemment, il existe plusieurs types de tags : les Identity tags (pouvant contenir une valeur de 0 à 65535) et les Bytes tags (de 0 à 255). cHTeMeLe étant composé de 43 cartes différentes, nous avons donc opté pour des Bytes tags.

Afin de reconnaître les tags, nous avons dû mettre un "TagVisualizer" englobant toute l'interface du jeu. Le TagVisualizer est chargé de détecter les tags posés sur la table interactive et d'interagir en fonction du tag. Chaque tag que nous voulons reconnaître doit être indiqué en paramètre dans le TagVisualizer, ce qui est un peu dommage car cela rend le code illisible (un gros pavé de code présent au début de la page de l'interface).

Une autre grosse partie du travail réalisé sur les tags fut de les associer à la création de la carte. Nous avons dû créer une classe Fabrique contenant pour chaque carte une méthode qui les instancie. Nous avons ensuite associé chacune de ces fonctions aux numéros de tags.

Toutes les fonctions de la Fabrique ont la même signature déclarée à l’aide d’un delegate Carte generateCarte().Le mot clef delegate étant spécifique au C#,(langage non présent dans notre formation), son utilisation a pris plus de temps que prévu. Les delegate remplacent la notion de pointeur de fonction présente en C++. Pour relier les numéros de tags avec les fonctions de la Fabrique, nous avons créé une classe d’association.

Les numéros des tags n’ont pas été pris au hasard. Nous avons choisi les tags les plus ressemblants car étant positionnés derrière les cartes, il aurait été facile aux joueurs adverses de les reconnaître.

## Chapitre 5 : Documentation HTML

## Chapitre 6 : Création zone joueur

Après avoir choisi le nombre de joueurs, le jeu démarre avec l’interface en cours de partie. Chaque joueur possède une zone de jeu personnelle en face de lui contenant :

* Une carte joueur
* Une documentation disposant des règles et d’une page d’indication sur chaque carte du jeu.
* Un indicateur affichant les messages durant la partie.
* Un bouton placé sur le côté gauche afin de passer au joueur suivant.

Elle comportait également le clavier auparavant.

## Chapitre 7 : Création carte joueur

Avant à jouer, chaque joueur doit choisir son « personnage » représenté par un navigateur web. (Un navigateur ne peut être choisi qu’une seule fois.)

Nous avons pour cela, dans un seul UserControl, disposé deux grids superposés : Le « sélecteur de navigateur » et la « carte joueur ».  
Le sélecteur propose cinq boutons, un pour chaque navigateur.

Une fois le personnage sélectionné, la carte change d’interface afin de présenter les différentes informations qui seront utiles en jeu (la « carte joueur ») :

* Le nom du navigateur choisi
* Le nombre de points
* Le nombre de points gagnés lors du dernier tour
* Les cartes que le joueur a posées lors du dernier tour
* Des icônes s’affichant lorsque le joueur reçoit un effet à retardement, avec un indicateur expliquant de quoi il s’agit

Lorsqu’on appuie sur une icône, un indicateur apparaît. Nous avons dû les ajouter manuellement avec un timer car nous n’arrivions pas à intégrer un pop-up prédéfinie. Au départ, nous avons eu des difficultés à faire fonctionner le timer, en effet il existe 3 namespaces proposant cette fonctionnalité. Mais pour notre utilisation nous n’avions pas choisi le bon et ne savions pas à ce moment qu’il existait d’autres namespaces…  
Finalement, grâce à Mme Huguel nous avons pu trouver chaussure à notre pied.

Il fut ardu d’intégrer le UserControl à l’interface car la solution n’est expliquée ni sur la documentation Microsoft ni sur les forums. Il a donc fallu se renseigner auprès de  
Mme Huguel. Il s’avérait qu’il suffisait de transformer le UserControl en Scatterview.

## Chapitre 8 : Création zone centrale

* Zone code HTML
* Zone pioche/défausse
* Zone page web
* Rotation auto

## Chapitre 9 : Navigateur internet - AWESOMIUM

La version surface de cHTeMeLe repose principalement sur l'affichage d'une page internet. Il est alors essentiel que l'application dispose d'un navigateur internet. Le programme utilise au total jusqu'à 5 navigateurs Internet simultanément, jusqu'à 4 navigateurs pour les documentations personnelles des joueurs et un navigateur pour le rendu HTML. Nous avons tout d'abord utilisé le navigateur web par défaut du SDK de Microsoft Surface, et c'est à ce moment précis que les problèmes commencent. En effet, il s'avère que le navigateur par défaut est impossible à tourner pour l'orienter vers les joueurs. Nous avons effectué de multiples tentatives infructueuses visant à effectuer une rotation du ScatterView. Mais nous avons tout de même pu émettre plusieurs solutions alternatives.  
Une première consistait à réaliser une documentation par joueur en intégrant une propriété CSS permettant de faire tourner un ou plusieurs éléments dans une page HTML. Le problème de cette solution est d'une part sa demande en mémoire, en effet, elle demande 4 documentations spécifiques pour chaque joueur. La principale raison de sa non-utilisation est également due au fait que cette propriété est spécifique à chaque navigateur Internet ; et après avoir essayé toutes les propriétés pour tous les navigateurs sans succès, cette solution fut écartée.

Une seconde solution avait été travaillée, l'idée était de se renseigner sur internet afin de voir s'il était possible d'intégrer un navigateur Internet autre que celui de base, là s’offraient à nous plusieurs navigateurs (WebOS, GeckoOS, Chromium, etc.). Mais bien que la réponse soit positive, parmi tous les navigateurs essayés, seulement un avait réussi à être intégré à l'application. Cependant celui-ci possédait malheureusement le même défaut que le navigateur de base, il ne tournait toujours pas.

Une dernière solution, qui fut d’ailleurs très vite abandonnée à cause de sa complexité, consistait à repartir d'un moteur de navigateur internet et de réaliser l'intégration du navigateur manuellement (ce qui est un travail titanesque).

A la suite de ce constat pour le moins alarmant, un retour sur les solutions s'imposait. Une recherche intensive fut réalisée afin d'approfondir toutes les solutions proposées précédemment, et le résultat fut concluant puisqu'au final, la deuxième solution fut employée.  
Nous avons donc trouvé et utilisé le navigateur Awesomium qui permet d'afficher une page internet tout en pouvant faire tourner le navigateur à notre guise. Awesomium est un navigateur internet développé par une petite équipe de développeurs passionnés. Il est gratuit pour toutes utilisations éducatives, non commerciales ou rapportant moins de 100 000 dollars de revenus.

## Chapitre 10 : Carte action

Les cartes Action sont des cartes qui permettent d’interagir avec les adversaires ou avec soi-même afin de s’accorder des avantages ou distribuer des désavantages.

Ces cartes Action se divisent en deux catégories : les Attaques et les Add-ons qui servent respectivement à désavantager ses adversaires et à s’accorder des bonus.

Certaines cartes Attaque donnent lieu à des effets différés sur leur cible :

* Le joueur à qui on inflige un Freeze doit passer son prochain tour.
* Le joueur à qui on inflige un CrashBrowser piochera quatre cartes de moins au début de son prochain tour.

Une seule carte Add-on octroie un bonus différé à celui qui la pose :

* Le joueur utilisant un BrowserUpdate piochera 2 cartes de plus au début de son prochain tour.

Pour afficher les effets différés sur la carte joueur, il a fallu créer une énumération des trois effets existants et une classe pour chacun d’eux. Ces classes héritant toutes les trois d’une quatrième classe Effet.

Voici un exemple de code avec l’effet BrowserUpdate :

* Dans le MdlCarteJoueur :

public bool hasBrowserUpdate()

{

foreach (Effect effect in \_player.effects())

{

if (effect.getTypeEffect() == Effect.EffectType.BROWSERUPDATE)

return true;

}

return false;

}

* Dans CartesJoueurs :

//affichage des effects

if (\_mdl.hasBrowserUpdate())

{

EffectBrowserUpdate.Visibility = System.Windows.Visibility.Visible;

EffectBrowserUpdate.IsEnabled = true;

}

Une carte spéciale a été créée afin de contrecarrer une carte Attaque : l’Antivirus. Elle est classée dans les Add-ons mais se comporte et se joue différemment : le joueur qui souhaite se protéger d’une attaque qui le cible doit poser sa carte Antivirus sur la table avant que son adversaire n’ait validé sa carte Attaque. Dès l’Antivirus posé sur la table, si une carte Attaque est présente, la carte Attaque sera annulée et les deux cartes seront alors défaussées. L’Antivirus est donc la seule carte du jeu pouvant être posée sur la table alors qu’une autre carte y est déjà, et est aussi la seule carte que les joueurs peuvent poser hors de leur tour.

Voici comment nous l’avons implémentée :

// TOUT DOUX : paragraphe sur l’implémentation des cartes fantômes et de l’antivirus

## Chapitre 11: Génération du code automatique

La page HTML étant stockée sous forme d’arbre contenant des objets personnels, il fallait créer l’algorithme permettant de rendre une version textuelle (fichier \*.html) de la page, afin qu’elle puisse être affichée dans le visualiseur de code et dans le visualiseur HTML.

Pour rappel (voir chapitre 3), il existe différentes classes :

* **HtmlElement** représente un élément HTML, contant deux balises (une ouvrante et éventuellement une fermante), une liste de fils étant des HtmlTagContent et une liste d’attributs.
* **HtmlTagContent** représente un contenu d’élément HTML qui peut être :
  + **HtmlText** : représente une chaîne de texte contenue dans un HtmlElement.
  + **HtmlElement**: en effet un HtmlElement peut en contenir d’autres, ce qui fait de cette structure de données un arbre N-aire.
* **HtmlTagAttribute** : représente un attribut qui peut être ajouté à la balise ouvrante des HtmlElement.
* **HtmlPage** contient les éléments caractéristiques d’une page Html, notamment plusieurs HtmlElement nécessaires à toute page HTML (<html>, <head>, <meta>), un autre pour le style CSS (<link>), et l’élément <body>, l’élément racine, contenant le reste de la page.

L’idée qui est venue tout de suite est de faire une fonction récursive qui génère le rendu textuel d’un HtmlElement. Il suffisait alors de déclencher le rendu de l’élément <html> pour faire un rendu de toute la page (sans oublier d’ajouter de <!DOCTYPE html> au début du fichier.)

Après plusieurs réflexions et schémas, il fut décidé que chaque classe ci-dessus aurait une fonction toString() qui renverrait le rendu html de l’objet.

Par la suite ces fonctions ont été renommées et déplacées pour les mettre dans une classe Modèle. Deux classes Vue utilisent les données de la page HTML : PageCode et PageRendu.  
Chacune d’elle possède une classe Modèle, respectivement MdlPageCode et MdlPageRendu.

Dupliquer ces fonctions de rendu permettra par la suite de changer le comportement du rendu selon que l’on veuille un rendu coloré pour la PageCode ou un rendu texte brut pour la PageRendu ; les deux étant correctement indentés dans tous les cas.

Etudions en détails le fonctionnement des fonctions de rendu des éléments pour ce qui est du **rendu textuel** (non coloré).

**RenderPage()** : Son fonctionnement est on-ne-peut plus basique : elle fait un rendu du doctype, puis appelle le rendu du HtmlElement <html>, contenant tout le reste de la page.

**RenderHtmlElement()** : En premier lieu, il faut distinguer trois type d’éléments :

* Les éléments multilignes : Leurs balises ouvrante et fermante occupent chacune une ligne entière, le contenu de l’élément étant indenté d’un niveau supplémentaire.
* Les éléments monoligne : L’élément entier occupe sa propre ligne.
* Les éléments inline : Plusieurs éléments inline peuvent se trouver sur la même ligne, mais pour des raisons esthétiques, on ajoute un retour à la ligne après certains, comme la balise <br>.

Une fois la nature du tag déterminée, on peut commencer à textualiser l’élément html.  
On commence par générer la liste des attributs (qui sera ajouté au texte de rendu plus tard) à l’aide de la fonction renderHtmlAttribute().

On rend ensuite la balise ouvrante de l’élément grâce à renderHtmlTag() ainsi que la liste des attributs générée plus tôt.

Puis vient une boucle qui va parcourir tous les HtmlTagContent de l’élément pour leur faire un rendu avec renderHtmlElement() (cette fonction est donc récursive).

Enfin, on rend la balise fermante si elle existe à l’aide de renderHtmlTag().

**RenderHtmlTag()** : Il existe deux types de tags : ouvrant ou fermant.  
Un tag ouvrant est de la forme <nom\_du\_tag [attributs]> (les attributs sont facultatifs)

Un tag fermant est de la forme </nom\_du\_tag>.

**RenderHtmlTagAttribute()** : Un attribut est composé d’un nom et d’une valeur. Il est de la forme nom="valeur".

**RenderHtmlText()** : Juste du texte.

Ainsi, en parcourant l’ensemble de l’arbre composant la page de façon récursive, nous sommes à même de recomposer un code Html.

Pour afficher dans le navigateur Web la page correspondant au code généré, il a fallu l’enregistrer sur le disque dur dans un fichier temporaire afin de faire charger celui-ci par le navigateur. Bien qu’Awesomium possède une fonction loadHTML() permettant d’afficher directement depuis une variable un rendu de page Web, il nous était obligatoire de passer par le disque dur sans quoi nous n’aurions pu intégrer des ressources (images, feuilles de styles, etc.) à la page.

## Chapitre 12 : Indentation automatique

Pour gérer l’indentation, nous avons décidé de parcourir ligne par ligne le code généré par les fonctions de rendu afin d’ajouter les espaces blancs au début de chaque ligne.

Nous avons utilisé les expressions régulières gérées nativement en C# grâce à la classe Regex (Regular Expression), à travers la fonction Replace, permettant de substituer la chaîne de caractères par une autre. Les deux chaines seront ici identiques à ceci près que la seconde sera correctement indentée.

Nous avons trois fonctions : une première permettant de parcourir le fichier html, une seconde permettant de supprimer les sauts de ligne multiples et d’indenter une ligne en fonction du résultat de la troisième fonction qui permet d’analyser les balises présentes dans la ligne.

Ces trois fonctions se nomment respectivement **autoIndent()**, **fetchLine()**, et **fetchIndentItem()**.

Ces fonctions utilisent également des variables communes contenant le niveau d’indentation au fil du parcours du fichier (indentLevel), sa variation au fil des lignes (computedIndentChanges) et une constante qui définit la longueur des niveaux d’indentation (indentSize).

Analysons leur fonctionnement :

**autoIndent()** : Cette fonction découpe le code ligne par ligne. Pour chacune d’elles, elle appelle la fonction de callback fetchLine() à l’aide de la Regex \n+(?<line>.\*) . Celle-ci commence par le caractère ‘\n’, cela signifie que la première ligne du fichier ne sera pas prise en compte, cela dit ce n’est pas gênant car la première ligne est le doctype et ne contient pas de balise modifiant l’indentation.

**fetchLine()** : Cette fonction va chercher à l’aide d’une autre Regex tous les éléments influant sur l’indentation, c’est-à-dire les balises ouvrantes et fermantes. La Regex en question est </?(?<tagname>\w+)(\s[^>]\*)?>. Elle permet de détecter un tag ouvrant ou fermant et de stocker son nom dans le tableau des éléments repérés par la Regex sous l’index « tagname ».

Le traitement de la chaine récupérée par cette Regex (donc chaque balise) s’effectue dans la fonction de callback fetchIndentItem(). Après le traitement de cette fonction, il est question d’indenter correctement la ligne en fonction des trois variables citées précédemment.

L’utilisation d’une variable intermédiaire computedIndentChanges est nécessaire car il faut distinguer le cas où l’on augmente le niveau d’indentation du cas où on le diminue. En effet l’augmentation du niveau d’indentation doit s’effectuer sur la ligne suivante et non sur la ligne actuelle, contrairement au cas de la diminution du niveau d’indentation.

**fetchIndentItem()** : Cette fonction utilise deux Regex pour distinguer trois types de balises : le doctype, les balises ouvrantes et les balises fermantes. Le premier est sans effet sur l’indentation, le second l’augmente d’un niveau s’il ne s’agit pas d’une balise singletag (sans balise fermante) et le dernier la diminue d’autant.

Ces changements s’effectuent sur la variable computedIndentChanges et non directement sur indentLevel pour la raison évoquée précédemment.

Les deux Regex utilisées sont les suivantes : ^<!.+>$ correspond au doctype, ^<(\w+)(\s[^>]\*)?>$ correspond aux balises ouvrantes. Vu le contexte dans laquelle cette fonction est appelée, le troisième cas correspond forcément à une balise fermante, ce qui fait que l’on peut se permettre d’économiser la troisième Regex.

Une fois que les calculs sur computedIndentChanges sont terminés, la fonction fetchLine() pourra indenter la ligne correctement.

## Chapitre 13 : Coloration syntaxique

Pour implémenter la coloration syntaxique nous avons décidé d’utiliser la classe Run de Surface 2.0 permettant de formater une chaîne de caractères. Un Run est composé d’un bout de texte et d’informations concernant le style de texte. Il faut donc créer un Run par élément stylisé.

Il a donc été pratique de distinguer le rendu du MdlPageCode de celui de MdlPageRendu, car finalement celui de MdlPageCode fonctionnera différemment et ne retournera plus le code HTML sous la forme d’une chaîne de texte, mais une liste d’associations Texte-Couleur.

Pour respecter l’architecture MVVM, on a découpé cette association en deux étapes. La première s’effectue dans MdlPageCode et associe une chaîne de caractères à une valeur d’une énumération StrType. La seconde s’effectue dans PageCode et associe les valeurs de StrType à une couleur.

Dans le Modèle (MdlPageCode), nous retrouvons les fonctions habituelles de rendu (**RenderPage()**, **RenderHtmlElement()**, **RenderHtmlTag()**, **RenderHtmlTagAttribute()** et **RenderHtmlText()**), mais leur fonctionnement a changé.

Le modèle se charge de faire un rendu sous forme d’une liste de StrTypePair, une classe désignant l’association Texte-StrType. Voici les différentes valeurs que peut prendre un StrType :

public enum StrType

{

DOCTYPE,

USR\_TAG,

USR\_OPEN\_TAG,

USR\_END\_TAG,

USR\_TAG\_NAME,

USR\_TAG\_ATTR\_NAME,

USR\_TAG\_ATTR\_AFFECT,

USR\_TAG\_ATTR\_VALUE,

USR\_TEXT,

AUTO\_TAG,

AUTO\_OPEN\_TAG,

AUTO\_END\_TAG,

AUTO\_TAG\_NAME,

AUTO\_TAG\_ATTR\_NAME,

AUTO\_TAG\_ATTR\_AFFECT,

AUTO\_TAG\_ATTR\_VALUE,

AUTO\_TEXT,

COMMENT,

LINE\_BREAK,

HOLOGRAM

}

On distingue les éléments USR, crées par les joueurs, des éléments AUTO, déjà présents en début de partie, afin d’afficher les éléments AUTO moins tape-à-l’œil tout en conservant leur coloration syntaxique.

Il y a également quatre valeurs plus particulières : une destinée au doctype, une autre aux commentaires, une aux retours à la ligne et une réservée aux éléments en surbrillance.

Etudions le fonctionnement du rendu de MdlPageCode :

Contrairement à l’algorithme de MdlPageRendu, celui-ci n’a pas besoin d’utiliser de RegEx étant donné que tous les sauts de ligne sont chacun de type StrType.LINE\_BREAK.  
En parcourant la collection de StrTypePair, il suffit de vérifier de quelle nature est le morceau de texte pour voir s’il s’agit d’une balise ouvrante, fermante, ou d’un saut de ligne, ce qui est bien plus pratique.

Les fonctions de rendu fonctionnent de la même façon que dans MdlPageRendu, au lieu d’ajouter du texte, elles ajoutent des StrTypePair à une liste.

Il faut savoir qu’un tag HTML, bien qu’il doive apparaitre uniformément stylisé dans la fenêtre de code, n’est pas constitué d’un seul StrType. Pour des raisons pratiques, nous avons scindé le code d’un tag avec trois StrType : \*\_OPEN\_TAG contient le marqueur d’ouverture de balise, soit un chevron ouvrant « < » ; \*\_END\_TAG contient le marqueur de fermeture de balise, soit traditionnellement le symbole « /> » ; et \*\_TAG\_NAME contenant uniquement le nom du tag (ce qui facilite la reconnaissance de la nature du HtmlElement).

Cependant c’est l’indentation automatique qui posa problème : on ajoutait les tabulations aux StrTypePair de type LINE\_BREAK, mais pour cela il fallait stocker une référence vers le dernier LINE\_BREAK rencontré (celui qui début la ligne en cours d’indentation), sans quoi il était impossible d’y ajouter ou retirer les tabulations lorsqu’on rencontrait un \*\_OPEN\_TAG ou \*\_END\_TAG sur la même ligne. C’est le rôle de la variable\_currentLineBreak .

Il a aussi fallu ajouter une fonction removeDoubleLineBreaks() qui sert comme son nom l’indique à fusionner les LINE\_BREAK juxtaposés en un seul pour la beauté du code.

Il restait encore une chose à implémenter avant d’avoir un beau code coloré et indenté : il fallait associer des couleurs à chaque StrType. Certains ont la même couleur que d’autres pour qu’ils apparaissent comme une seule entité aux yeux des joueurs, comme c’est le cas pour les tags HTML qui sont comme dit plus haut scindés en trois StrType.

Nous avons repris les couleurs utilisées dans le logiciel Notepad++, c’est-à-dire des balises bleues, du texte noir, et des attributs rouges et violets. Les hologrames sont censés être peu visibles, ce qui fait que nous les affichons en gris, le fond de la PageCode étant gris clair.

Ces associations sont stockées dans le dictionnaire colorTable . Une fois la page générée, la fonction ShowCode() de la PageCode parcours la liste des StrTypePair rendus par RenderPage() et créée un Run à partir du texte et de la couleur associée au StrType, puis l’ajoute à PageCode. Une fois le tableau parcouru, nous avons enfin notre beau code indenté et coloré.

## V&C- Chapitre 14 : Gestion de la pose de cartes

## V- Chapitre 15 : Charger des fichiers CSS

* Switcher entre 3 styles différents

## V-Chapitre 16 : Implémentation des règles

* Cartes action effet
* Carte HTML
* Antivirus (cartes fantômes)

## Chapitre 17 : Design

Afin de rendre notre application visuellement plus agréable et moins monotone, nous lui avons ajouté un background, une icône ainsi que des sons lors du choix des personnages et de la validation de cartes.  
 De plus, après modification des règles et des cartes, il a fallu créer les nouvelles cartes que nous avons intégrées au jeu. Après quelques recherches sur internet afin de connaître la police des cartes de base, il s'est avéré qu'il s'agissait de "Source Code Pro". Les nouvelles cartes ont été ensuite créées à partir des cartes de base en respectant le code couleur du jeu initial. Ce nouveau design des cartes a servi à intégrer les tags derrière chaque carte, afin de simplifier leur impression (cela évite de coller les tags à la main).

Un test d'impression des cartes sur papier cartonné fut réalisé avec succès avant de lancer l'impression de toutes les cartes du jeu.

# Résultats

### Différents essais lors du développement

Tout le long du développement de l’application nous avons utilisé l’Input Simulator sur nos ordinateurs. Mais lorsque l’application fut fonctionnelle début février, nous avons commencé à effectuer quelques tests directement sur la table interactive.

Les tout premiers essais furent de tester la reconnaissance des tags. Après ces tests concluants, nous avons lancé une première impression sur papier d'une page de carte. Ces cartes se sont révélées trop transparentes, perturbant ainsi la reconnaissance des tags. Nous avons donc acheté du papier cartonné pour de palier au problème de la transparence, mais aussi afin de créer des cartes plus solides. Après une nouvelle impression d'une page de carte, les tags étaient reconnus parfaitement. Nous avons donc lancé une impression de toute les cartes du jeu en couleur sur papier carton, puis nous les avons testé une par une afin de vérifié si elles étaient toutes reconnues. Toute les cartes ont été reconnu du premier coup. Les tests suivants ont permis de repérer les éventuels bugs, mais aussi de vérifier directement sur la SUR 40 en condition réel (perturbation, entrée non parfaite).

1. **L’essai final avec les cartes**

Nous avons réalisé un essai final afin de s'assurer du bon déroulement de l'application. Ce test fut concluant, mais il nous a permit de prendre conscience de plusieurs éléments manquant, et de certain défaut de l'application que nous allons vous détailler plus bas.

# Conclusion et améliorations

Conclusion

Ce qui nous restera particulièrement en mémoire à la fin de cette période de développement, les événements les plus marquants, sont :

* Awesomium, l’intégration du navigateur web
* L’implémentation aventureuse du moteur HTML, ainsi que sa génération de code
* Les efforts que nous avons faits pour maintenir la motivation de chaque membre du groupe pour le bien du développement du cHTeMeLe

Ce fut un projet très agréable à réaliser car le cHTeMeLe est non seulement un bon jeu de cartes, mais aussi un concept innovant. L’adapter sur table interactive est ce qui lui manquait pour être de nouveau mis au goût du jour.

Le sujet était très intéressant, car travailler sur la table interactive fut nouveau et formateur pour chacun d’entre nous. De plus, c’est la première fois que nous travaillons ensemble sur un périphérique tactile.

Développer le cHTeMeLe nous a permis d’acquérir de nombreuses connaissances sur un langage de programmation que nous n’avions alors pas étudié à l’IUT : le C#. Le SDK utilisé pour développer cHTeMeLe a également été une découverte intéressante, bien que son développement ait été abandonné. Nous avons eu l’occasion d’apprendre le pattern de conception MVVM, qui nous était inconnu jusqu’alors.

Un si long projet nous a permis d’apprendre à gérer au mieux notre temps, cependant il y a avait beaucoup à faire, nous n’avons malheureusement pas réussi à atteindre l’application que nous espérions. Nous avons également essayé de se motiver les uns et les autres afin d’avancer ensemble.

Améliorations envisageables

Notre version du cHTeMeLe n’est qu’une version alpha. Pour que l’application soit plus complète on pourrait lui ajouter les fonctionnalités suivantes :

* La détection des erreurs si on pose une mauvaise carte
* Expliquer pourquoi on n’a pas le droit de poser telle ou telle carte
* Des animations
* Une application plus jolie
* Charger sa propre feuille de style CSS
* Sauvegarde/chargement des parties
* Afficher une description de la carte dans une pop-up lorsqu’elle est posée sur la table
* Une documentation plus jolie et complète

# Annexes