La 3D sur le web avec BabylonJS

Les versions actuelles des navigateurs web supportent désormais la spécification en cours d'HTML5. Finie la guerre des standards : quand une fonctionnalité est implémentée, elle fonctionne exactement de la même façon dans IE, Firefox, Chrome, Safari, Opéra que ce soit sur un PC ou sur un smartphone. Ceci est une bonne chose pour les développeurs qui souhaitent voir leur application toucher le maximum d'utilisateurs.
Cependant, jusqu'à l'avènement de HTML5, la représentation graphique était loin d'être la panacée dans un navigateur. HTML5 propose maintenant un élément du DOM nommé *canvas* et dédié à l'insertion d'images pouvant être dynamiquement élaborées par le code exécuté dans une page web Cet élément *canvas* fournit au développeur deux contextes de programmation Javascript différents: *2D* et *WebGL*. Il est à noter que les appellations *2D* ou *WebGL* ne caractérisent pas en elles-même le fait de faire ou non de la 3D. Tant que nous ne disposons pas d'écrans holographiques, nous affichons bien toujours une image en deux dimensions uniquement, même lorsqu'on parle de 3D. Le contexte *2D* signifie ici que l'environnement de programmation donne accès au développeur directement au système de coordonnées 2D (abscisse, ordonnée) à l'intérieur du *canvas*.

C'est donc au travers de cet élément *canvas* que nous allons pouvoir afficher des images dites « en trois dimensions » et ceci, nativement, sans l'installation d'un plug-in de tierce partie et en n'utilisant que Javascript.

Pourquoi faire de la 3D dans le navigateur ?

Pour mille raisons qu'il serait impossible de toutes détailler ici.
S'il fallait n'en donner que quelques unes, on pourrait commencer par citer les mêmes raisons qui font qu'on utilise la couleur plutôt que le monochrome, qu'on utilise des interfaces graphiques plutôt qu'une simple console en ligne de commande, qu'on utilise des contenus multi-médias plutôt que du simple texte, etc. En d'autres mots parce qu'il s'agît avant tout d'une amélioration de l'expérience utilisateur.
Bien entendu, cela ne se limite pas loin de là à de simples considérations de confort d'utilisation. En effet, disposer d'une dimension supplémentaire, la profondeur, fournit un apport essentiel dans les domaines de l'architecture, de l'imagerie médicale, de la conception mécanique ou même de la simple représentation de données numériques (charts, etc), sans parler évidemment du caractère immersif pour le monde des jeux vidéos.

Les techniques de la 3D

Les données
Pour représenter une vue 3D, on manipule un ensemble de concepts, toujours les mêmes :

- une scène qui est le monde conteneur de tout ce qui sera potentiellement visible dans la vue,
- une ou plusieurs sources de lumières (*lights*) qui vont éclairer les objets de la scène,
- un ou plusieurs objets 3D, appelé encore maillages (*mesh*), qui représentent des volumes dans la scène,
- un point de vue sur cette scène, nommé caméra.

Un mesh est constitué d'un ensemble de facettes (*faces*) contiguës. Chacune de ces facettes est simplement un triangle ayant pour sommets (*vertex, vertices au pluriel*) trois points dans l'espace (*vectors*), trois points suffisant à déterminer un plan. De cette manière on peut représenter n'importe quel volume, y compris une sphère : imaginez vous une boule disco couverte de centaines de minuscules triangles plans réfléchissants.
Récapitulatif : un *vertex* est un sommet, il nécessite donc trois nombre flottants (*floats)* pour ses coordonnées x,y, z dans l'espace. Il faut trois sommets pour constituer une facette. Il faut des

centaines ou des milliers de facettes pour représenter un unique *mesh*. Une scène peut contenir des dizaines, des centaines ou des milliers de meshes.
Ceci commence à donner une idée du volume de données numériques à manipuler dans le code. Mais nous sommes pourtant loin du compte. En effet, la scène étant éclairée, il faut calculer et associer à chaque vertex de chaque mesh un vecteur, nommé normale (*normal*), soit trois *floats x, y, z* de plus, définissant dans quel sens la lumière va se réfléchir en ce point sur le mesh. Par ailleurs, des textures (*texture*), c'est à dire des images 2D pouvant provenir de fichiers jpg ou png, sont généralement appliquées sur certaines faces du mesh pour l'habiller et lui donner par exemple l'aspect d'un matériau particulier. Ceci nécessite donc d'associer à chaque vertex une paire de coordonnées supplémentaires *u, v* dans le plan de l'image utilisée pour la texture afin de définir en quels endroits cette dernière doit se « plier » ou « s'enrouler » sur la surface du mesh.
On comprend maintenant que le volume de données numériques à traiter risque d'être assez conséquent.

Les traitements

Nous savons maintenant de quelles données nous avons besoin pour décrire un ou plusieurs volumes dans une scène, à savoir essentiellement des coordonnées de sommets et des relations entre ces sommets pour définir des faces. Nous n'avons donc jusqu'à présent défini que des relations entre des constituants d'un même volume dans un référentiel local à ce volume nommé *Model*.

Cela ne suffit évidemment pas. Il nous faut en effet placer tous ces différents volumes les uns par rapport aux autres dans un référentiel commun, celui de la scène, appelé *World.* Il faut aussi donner à chacun une orientation dans cette scène, orientation définie par trois valeurs d'angle de rotation autour des axes *X, Y, Z* du *World*. Il faut enfin leur donner une échelle, c'est à dire un ratio d'étirement sur les axes *x, y ,z* du *Model* (ce qui permet d'étirer un cube vers la forme d'un pavé par exemple).

Ces trois opérations successives, mise à l'échelle, rotation et translation, font donc passer le mesh de son référentiel local à sa place finale dans la scène. Cette transformation est nommée *Model to World* et est définie mathématiquement par une matrice de transformation de dimensions 4 x 4.
On multiplie donc tous les triplets de coordonnées connus dans *Model* (ils sont très nombreux comme nous venons de le voir précédemment) par cette matrice de transformation *Model to World* pour obtenir les triplets de coordonnées transformés dans *World.*
Beaucoup de données et beaucoup de calculs jusqu'ici… et pourtant ce n'est pas fini.

Nous ne venons que de placer les meshs dans le *World.* Encore faut-il les visionner.
Selon les coordonnées de la caméra dans la scène et la direction de vue choisie, on définit une matrice 4 x 4 supplémentaire nommée *World to View.* Et comme auparavant, on obtient les coordonnées de tous les meshes dans le référentiel de vue en multipliant les résultats précédents par cette matrice *World to View.*

Nous y sommes presque. Il nous reste à passer de la vue 3D à l'écran, c'est à dire à opérer une projection des dernières coordonnées spatiales obtenues en coordonnées de l'écran. Ces dernières vont dépendre en autres de la forme de l'écran (4/3, 16/9, autre) et de l'angle de vision choisi, un peu comme lorsqu'on règle un zoom sur un appareil photo. Encore une dernière fois, cette projection se réalise par la multiplication du dernier résultat obtenu par une matrice de projection nommée *View to Projection*.

En résumé :

coord. à l'écran = (*View to Projection*) * (*World to View*) * (*Model to World*) * coord. Model

Par ailleurs, j'ai volontairement omis d'évoquer les mêmes transformations à opérer sur les vecteurs des normales, sur le calcul des couleurs de chaque pixel en fonction des lumières et des éventuels ombrages ou de l'application des textures ...

On se rend donc vite compte de deux choses :

Le nombre de traitement à implémenter en Javascript est lourd et fastidieux, d'autant que ses fonctions mathématiques natives sont plutôt limitées (pas de calcul matriciel!). On aura donc tout intérêt à utiliser une librairie ou un framework qui propose une fois pour toutes ces opérations.

Le nombre de calculs flottants à réaliser dans chaque traitement est particulièrement conséquent d'autant qu'on peut avoir des dizaines de milliers de coordonnées à traiter. Le temps de traitement est en effet critique : le navigateur peut rafraîchir l'affichage, c'est à dire redessiner toute la scène, à la fréquence optimale de 60 trames par seconde (*frames per second ou fps*), soit un délai de 16 ms entre deux trames. Il faut donc parvenir à effectuer tous ces traitements dans ce laps de temps avec Javascript, un langage monothreadé à typage dynamique … et encore, je n'ai pas parlé des autres traitements qui consisteraient à donner un peu de logique applicative à la scène par des mouvements des solides, des déformations ou la mise en œuvre d'un moteur physique pour émuler la gravité ou calculer les collisions entre les meshes.

La solution à ce deuxième point va être l'utilisation de WebGL.
WebGL est une partie de l'API HTML5 accessible depuis l'environnement JS du browser. Elle permet d'envoyer des commandes OpenGL ES (*OpenGL for Embedded Systems*) directement à la carte graphique. La bonne nouvelle c'est que la quasi-totalité des appareils actuels, du PC, au smartphone en passant par la tablette intègrent maintenant une carte graphique dédiée, le GPU, compatible WebGL.
On peut donc déléguer une énorme quantité de traitements et bien entendu tout le rendu au GPU, qui justement est capable de paralléliser intensément tous les calculs matriciels de flottants. C'est ce que l'on appelle l'*accélération matérielle* puisqu'un autre processeur que le CPU se charge des calculs dédiés à l'affichage. Ce déport de traitements sur le processeur graphique décharge donc d'autant le CPU dont les ressources pourront alors être utilisée pour coder principalement la logique applicative en Javascript.
L'accès à WebGL demeure très verbeux et de bas niveau : il faut déclarer des buffers de transfert de données du CPU vers le GPU, récupérer des pointeurs de variables utilisées sur le GPU, réaliser des liaisons (*binds*) entre les deux contextes, etc. Il est donc très complexe d'implémenter la logique applicative côté GPU, contrairement aux calculs matriciels déjà cités qui sont toujours les mêmes quelle que soit l'application.

Ce qui nous amène à la solution du premier point : le choix d'un moteur 3D HTML5.
Tous les moteurs 3D web existants dignes de ce nom utilisent exactement de la même façon les points énoncés précédemment, à savoir des transformations matricielles entre des systèmes de coordonnées successifs calculés par le GPU.
Les choix se feront donc sur d'autres critères comme les fonctionnalités proposées, les objectifs visés par le moteur ou les paradigmes utilisés.
Actuellement on dénombre trois grandes catégories de moteurs : les plate-formes commerciales non-web de développement de jeux vidéo, les plate-formes commerciales et/ou libres de développements de jeux HTML5 et les moteurs libres 3D HTML5.

Les plate-formes non-web de développement de jeux telles que les grands standards *Unity3D* ou *Unreal Engine* fournissent un ensemble d'outils qui permettent d'élaborer des jeux video quasiment sans programmation de la part de l'utilisateur. Les exécutables de jeux sont générés ensuite pour les

OS cibles. Il est possible d'exporter le produit fini au format HTML5. Malheureusement, le code produit dans le cas de la 3D est beaucoup trop lourd pour être raisonnablement déployé sur le web, si bien qu'il est peut s'avérer plus judicieux d'utiliser ces plate-formes pour concevoir les objets 3D et les exporter ensuite dans un outil web plus performant.

Les plate-formes de développement de jeux HTML5, comme *Playcanvas*, *Turbulenz* ou *Goo Engine* fournissent en général à l'utilisateur une API de programmation libre et un accès payant à des outils en ligne de conception (éditeur 3D, réalisation de scènes), de génération du code (build) ou de distribution des jeux conçus. Ils se focalisent sur la réalisation de jeux uniquement et sur une plate-forme cible : le navigateur.

Enfin les moteurs libres 3D web ne fournissent en général qu'une API de programmation et n'imposent aucune contrainte au développeur. Pour trier parmi la masse de projets existants, on ne retiendra que ceux qui ont une activité depuis au moins un an et qui ont déjà sorti au moins une version stable, à savoir les deux projets open-source *Three.js* et *Babylon.js*.
Three.js, le plus ancien des deux, n'est pas dédié a priori uniquement à la conception de jeux et permet d'effectuer des rendus visuels aussi bien en WebGL, qu'en Canvas 2D ou qu'en CSS3. Aussi fournit-il une quantité impressionnante de fonctionnalités afin d'essayer de couvrir tous les usages. Babylon.js se concentre au contraire uniquement sur le rendu WebGL et la performance dans les animations. Bien qu'il ne soit pas uniquement dédié à cet usage, il se veut capable de produire de vrais jeux 3D pour le navigateur, objectif pour lequel il a déjà été éprouvé avec succès comme le montrent par exemple les réalisations de Microsoft Edge (Flight Arcade) ou d'Ubisoft (Assassin's Creed Pirates). C'est à Babylon.js que nous allons nous intéresser dans la suite de cet article.

Babylon.js

Babylon.js a été créé par deux Français travaillant chez Microsoft, David Catuhe et David Rousset, en 2013. Il s'agit d'un port vers Javascript et WebGL d'un précédent moteur 3D éponyme conçu par David Catuhe pour la plate-forme XNA. BabylonJS est un projet open-source sous la licence Apache 2.0 accessible à tous sur Github. Il propose de plus un site web portail http://babylonjs.com donnant accès à tout un écosystème d'outils : le dépôt Github, un forum animé par une communauté particulièrement dynamique et réactive où de très nombreux français sont présent même si la langue d'échange y est l'anglais, un site de documentation dédié aussi bien pour l'API que pour les tutoriels, des exemples de code par type de fonctionnalités, un éditeur en ligne associé à un rendu en direct, un éditeur de matériels, un éditeur de shaders (des programmes exécutés directement côté GPU), un importateur de données à partir de formats externes, etc.
Les fonctionnalités proposées par BabylonJS sont trop nombreuses pour être détaillées ici. Je vous invite à les découvrir dans la documentation, les nombreux tutoriels, cours vidéo ou les exemples de démonstrations en ligne. Le credo de BabylonJS est « simple, powerful WebGL », c'est à dire qu'il suit principalement deux buts dans sa conception : fournir à l'utilisateur l'API la plus simple possible en terme d'abstraction de la couche WebGL et toujours optimiser la performance, à tel point qu'il est actuellement le seul moteur à implémenter des *webworkers* afin de threader le calcul des collisions entre la caméra et les meshes de la scène ou encore de prendre déjà en compte dans les navigateurs qui le supportent la technologie *SIMD (single instruction on multiple data)* pour paralléliser sous Javascript les calculs matriciels de flottants demeurant côté CPU.

Une application Babylon.js

Concrètement, une application BabylonJS n'est rien d'autre qu'une page HTML5 embarquant du code Javascript, la librairie babylon.js et contenant un élément *canvas*. Tout est exécuté dans le navigateur. Il sera nécessaire cependant de charger la page HTML depuis un serveur web, qui peut être local, plutôt que directement depuis le système de fichiers de la machine, du fait des restrictions

de sécurité dans les navigateurs.

Commençons donc par une page HTML minimale : *index.html*.
Dans cette page nous allons charger *babylon.js*, la librairie BabylonJS elle même, *hand.js* un petit programme permettant de prendre en charge et d'abstraire tous les types de pointeurs (souris, doigts, etc), ce qui nous permettra de déplacer la caméra de la même façon sur un PC ou une tablette et le fichier dans lequel nous coderons notre application : *mon_appli.js*.
Comme vous le voyez, une fois la page chargée, la fonction *init* est appelée. Nous la coderons donc dans *mon_appli.js*

index.html :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = 'utf-8'>
    <title>Ma première application BJS</title>
    <script src = 'babylon.js'></script>
    <script src = 'hand.js'></script>
    <script src = 'mon_appli.js'</script>
  </head>
  <body>
    <canvas id='renderCanvas'></canvas>
    <script>
      window.onload = init;
    </script>
  </body>
</html>
```

mon_appl.js : On code la fontion *init* qui récupère l'élément canvas du DOM et qui crée une instance de moteur BABYLON avec ce canvas. Le moteur 3D est lancé, il affiche une scène que nous coderons dans la fonction *createScene()* dans le même fichier.

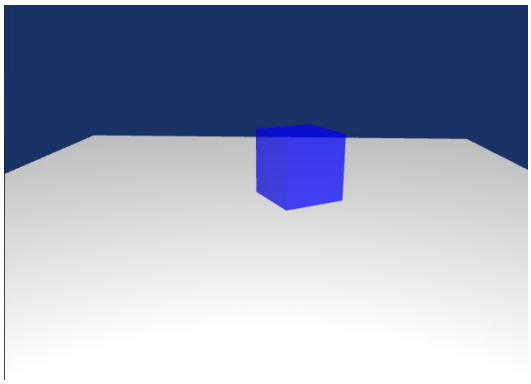
```
var init = function() {
  var canvas = document.querySelector('#renderCanvas');
  var engine = new BABYLON.Engine(canvas, true);
  var scene = createScene(canvas, engine);
  engine.runRenderLoop(function() { scene.render(); });
};
```

Ces seuls deux petits bouts de code réutilisables suffisent à mettre en place l'environnement d'exécution du moteur 3D. Il reste donc maintenant à créer la scène, son contenu et ajouter la logique applicative.
Commençons par un exemple simple c'est à dire une scène contenant un cube (*box*) posé sur un sol (*ground*) et éclairé depuis le haut. BabylonJS fournit de nombreuses fonctions permettant de créer en une seule ligne ces types d'objets.
Nous allons donc successivement créer un objet *scene*, un objet *camera* et lui attacher les contrôles afin que cette caméra puisse être déplacée avec la souris, le clavier ou au toucher sur un écran tactile et une source lumineuse dont on donnera la direction.
Ensuite nous créerons un objet *ground* et un objet *box* auxquels nous associerons à chacun un *material* pour leur donner une couleur et une opacité, par exemple.

```
var createScene = function(canvas, scene) {
  var scene = new BABYLON.Scene(engine);

  // camera
```

```
  var camera = new BABYLON.ArcRotateCamera('cam', 0, 0, 0,
BABYLON.Vector3.Zero(), scene);
  camera.setPosition(new BABYLON.Vector3(0, 10, -60));
                       //placement de la caméra dans la scène
  camera.attachControl(canvas);  // attache des contrôles
  // lumière
  var light = new BABYLON.HemisphericLight('light', new BABYLON.Vector3(0,
1, 0), scene);

  // sol
  var ground = BABYLON.Mesh.CreateGround('ground', 100, 100, 4, scene);
  ground.material = new BABYLON.StandardMaterial('groundMat', scene);
  ground.material.diffuseColor = new BABYLON.Color3(0.7, 0.7, 0.7);

  // cube
  var box = BABYLON.Mesh.CreateBox('box', 10, scene);
  box.material = new BABYLON.StandardMaterial('boxMat', scene);
  box.material.diffuseColor = BABYLON.Color3.Blue();
  box.material.alpha = 0.7;       // transparence
  box.position.x = 5;            // placement du cube dans la scène
  box.position.y = 5;
  box.rotation.y = Math.PI / 3;   // rotation du cube autour de son axe Y

  return scene;
};
```

Ce qui nous permet d'obtenir la scène de la figure suivante qui, à part la simplicité de son code, ne présente pas grand intérêt. En général, les utilisateurs qui souhaitent construire des scènes sophistiquées conçoivent leurs objets dans des logiciels de modélisation 3D comme Blender, 3ds Max ou Unity avant de les importer dans BabylonJS.

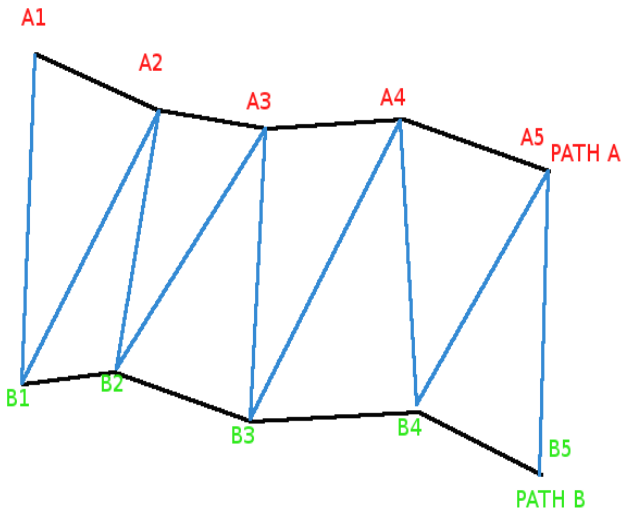


Cependant l'utilisation des formes de mesh basiques (cube, anneau, sphere, tube, etc) proposées par BJS peut être suffisante pour la représentation de données. Pour ceci nous allons nous intéresser à un type de formes particulier : les formes paramétriques.

Les formes paramétriques : le ruban

Les formes paramétriques n'ont pas de forme prédéfinie, contrairement par exemple à une sphère pour laquelle on s'attend à voir une forme sphèrique. Leur forme va dépendre d'un ensemble de données qui leur seront passées en paramètre à la construction du mesh. On a donc ici un moyen simple de transformer des jeux de données en visualisation graphique 3D.
Les formes paramétriques de BJS présentent de plus la particularité d'être toutes dynamiquement modifiables, c'est à dire que la forme initialement construite pourra être ensuite modifiée si le jeu de données passé en paramètre est modifié.

Examinons la forme paramétrique de base de BJS, à savoir le ruban (*ribbon*). Un ruban représente la surface entre deux (ou plus) chemins, un chemin n'étant qu'une suite de points consécutifs dans l'espace.

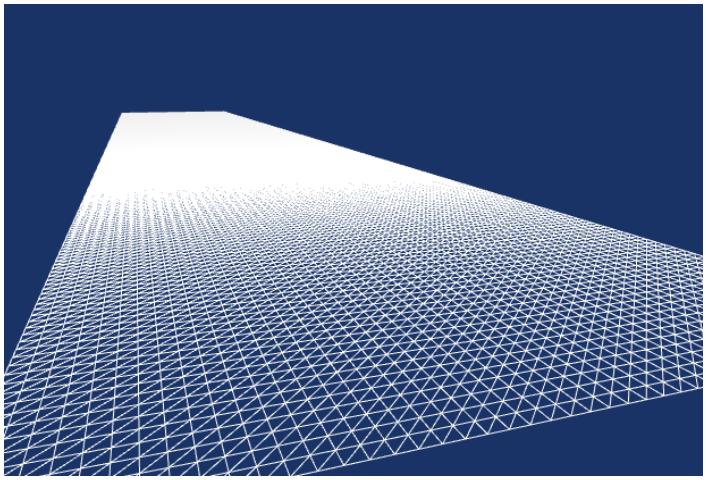


Si on dispose de plusieurs de jeux de données (des mesures selon des intervalles de temps par exemple), il est assez facile de transformer chaque jeu en coordonnées de points, constituant alors chacun un chemin.

Prenons un exemple concret : toutes les 5 secondes, je reçois du serveur (soit par une requête xhr, soit par une websocket) dans un tableau au format JSON un lot de 50 valeurs numériques qui peuvent des mesures de métrologie, des valeurs boursières, etc. Je reçois simplement 50 nouvelles valeurs de ce qu'on mesure toutes les 5 s et je décide d'afficher constamment 5 minutes de données, soit 5 x 60 = 300 jeux de données.

Dans le code précédent, supprimons tout ce qui concerne le sol et la box et créons un simple ruban plat pour commencer de 300 chemins, chacun contenant 50 points.

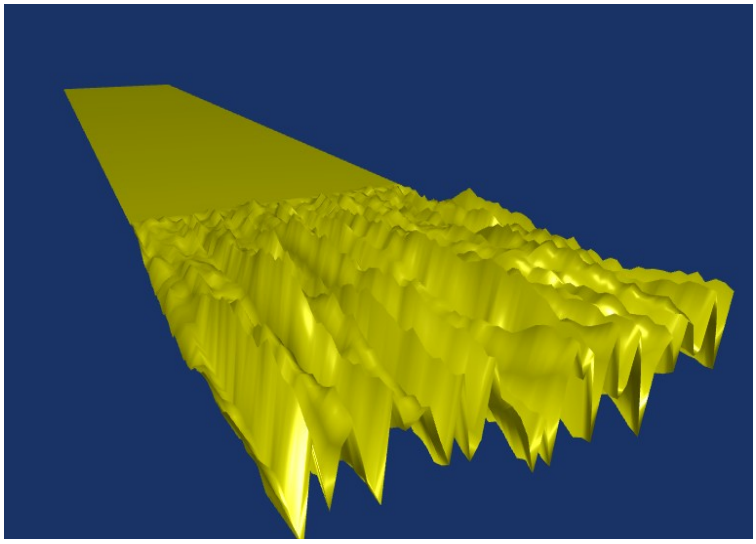
```
var paths = [];
for (var i = 0; i < 300; i++) {
  var path = [];
  for (var j = 0; j < 50; j++) {
    path.push(new BABYLON.Vector3(25 - j, 0, i - 30));
    }
  paths.push(path);
}

var ribbon = BABYLON.Mesh.CreateRibbon('ribbon', paths, null, null, 0,
scene, true);
ribbon.material = new BABYLON.StandardMaterial('mat', scene);
ribbon.material.wireframe = true;
```



Nous allons maintenant modifier ce ruban toutes les 5 s avec les données reçues. Pour ceci, nous allons simplement modifier les valeurs contenues dans chaque *path* du tableau *paths* à l'aide des nouvelles données. Nous rappellerons alors la méthode *createRibbon()* en lui donnant uniquement les nouvelles données et l'instance de *ribbon* en paramètre (le reste sera *null*) ce qui provoquera la mise à jour de la forme.

```
var delai = 5000;
window.setInterval(function() {
  var data = recupereDonnees(); // récupère le tableau de 50 données
```

```
  var path =[];
  // on construit un nouveau premier chemin
  // la donnée reçue est l'ordonnée y du point courant
  for (var j = 0; j < 50; j++) {
   path.push(new BABYLON.Vector3(j, data[j], 0));
  }
  // on décale tous les chemins existant en actualisant le z de chaque point
  for (var i = 299; i > 0; i--) {
   paths[i] = paths[i -1];
    var courant = paths[i];
    for (var k = 0; k < 50; k++) {
      courant[k].z = i;
    }
  }
  paths[0] = path;          // ajoute le nouveau chemin au début
  // on actualise enfin la forme
  BABYLON.Mesh.CreateRibbon(null, paths, null, null, null, null, null, null,
ribbon);
}, delai);
```



L'essentiel du code traite ici des changements de coordonnées des points de chaque chemin. La mise à jour de la forme se code ensuite en une seule ligne.

Conclusion

La 3D sur le Web procure de nombreux apports quels que soient les domaines applicatifs, y compris la simple visualisation de données numériques.
Les avantages bien connus du déploiement Web (unicité de la plate-forme cible : le navigateur, du langage, pas d'installation, etc) pourraient être contre-balancés par la complexité et la charge des calculs à mettre en œuvre sur le client. Heureusement l'usage d'un framework WebGL comme BabylonJS remédie à cette difficulté en permettant à l'utilisateur final de faire abstraction de la couche du moteur 3D et de se concentrer uniquement sur la logique applicative de son besoin.

Auteur

Jérôme Bousquié est ingénieur de recherche à l'IUT de Rodez et contributeur au projet BabylonJS.