Rapport d’étude

Sujet : Technologies de modélisation 3D

# Introduction

Ce document est là pour rapporter l’étude d’une panoplie de technologies permettant la modélisation 3D. Framework, Moteur 3D, etc., tous les cas intéressants seront étudiés. Une seule contrainte va filtrer les technologies candidates à l’étude : elles doivent être basées sur du JavaScript.

Le but étant d’étudier les différentes offres, possibilités de chaque option, ce afin de déterminer la ou les technologies les plus intéressantes pour l’élaboration d’un synoptique prenant en charge des plans en 3D.

# La 3D avec WebGL

WebGL est une API de 3D dynamique JavaScript apparue en 2011 créée par le Khronos Group. Elle permet d’utiliser les standards d’OpenGL ES dans une page web et sur bien d’autres supports. Elle permet également d’utiliser les pilotes d’« Open Graphic Library for Embaded Systems » et d’utiliser la puissance que peuvent fournir les processeurs graphiques du terminal informatique.

C’est une technologie complète et très soutenue qui permets l’obtention de résultats très poussés. API JavaScript, la version 2.0 est sortie en 2017, elle deux grandes force pour le projet synoptique qui sont : la technologie basée sur JavaScript, et le soutien sans faille d’une communauté active.

WebGL 2.0 est vieille de 3 ans à l’écriture de ce rapport, les bugs du passage de la 1.0 à la 2.0 sont donc loin derrière et la version est très stable. Presque tous les navigateurs à jour sont compatibles avec cette version de WebGL : la fondation Mozilla estime qu’environ 75% des terminaux le sont.

Cette introduction faite, je peux passer aux technologies qui utilisent toutes WebGL pour leurs rendus 3D.

# Technologie n°1 : Babylon.js

Babylon.js est un moteur 3D temps réel qui se présente sous la forme d’une bibliothèque JavaScript.

Interprété nativement par un navigateur web supportant la norme HTML5. Le langage de programmation utilisé est le JavaScript, permettant les calculs et le rendu 3D par l'intermédiaire de l'interface de programmation WebGL.



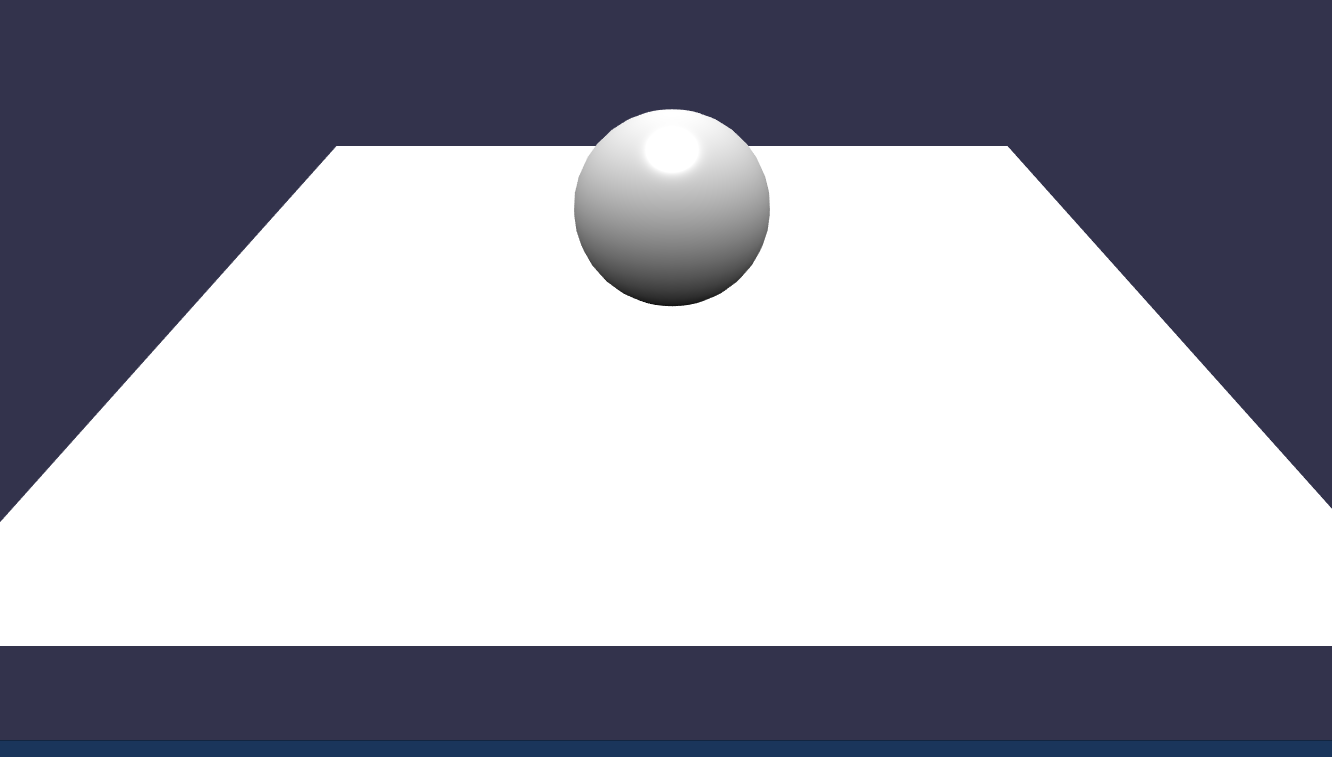
Babylon.js est complet, simple mais puissant. Il permet de modéliser presque tout, supporte tous les formats d’import.

Lors de mon test sur la mise en place d’un POC simple je suis arrivé en quelques lignes à avoir un rendu propre.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <meta http-equiv="Content-Type" content="text/html" charset="utf-8"/>      <title>Babylon - Getting Started</title>  *<!--- Link to the last version of BabylonJS --->*      <script src="https://cdn.babylonjs.com/babylon.js"></script>      <style>          html, body {              overflow: hidden;              width   : 100%;              height  : 100%;              margin  : 0;              padding : 0;          }          #renderCanvas {              width   : 100%;              height  : 100%;              touch-action: none;          }      </style>  </head>  <body>      <canvas id="renderCanvas"></canvas>      <script>          window.addEventListener('DOMContentLoaded', function(){  *// get the canvas DOM element*              var canvas = document.getElementById('renderCanvas');  *// load the 3D engine*              var engine = new BABYLON.Engine(canvas, true);  *// createScene function that creates and return the scene*              var createScene = function(){  *// create a basic BJS Scene object*                  var scene = new BABYLON.Scene(engine);  *// create a FreeCamera, and set its position to (x:0, y:5, z:-10)*                  var camera = new BABYLON.FreeCamera('camera1', new BABYLON.Vector3(0, 5,-10), scene);  *// target the camera to scene origin*                  camera.setTarget(BABYLON.Vector3.Zero());  *// attach the camera to the canvas*                  camera.attachControl(canvas, true);  *// create a basic light, aiming 0,1,0 - meaning, to the sky*                  var light = new BABYLON.HemisphericLight('light1', new BABYLON.Vector3(0,1,0), scene);  *// create a built-in "sphere" shape; its constructor takes 6 params: name, segment, diameter, scene, updatable, sideOrientation*                  var sphere = BABYLON.Mesh.CreateSphere('sphere1', 16, 2, scene);  *// move the sphere upward 1/2 of its height*                  sphere.position.y = 1;  *// create a built-in "ground" shape;*                  var ground = BABYLON.Mesh.CreateGround('ground1', 10, 10, 2, scene);  *// return the created scene*  *return* scene;              }  *// call the createScene function*              var scene = createScene();  *// run the render loop*              engine.runRenderLoop(function(){                  scene.render();              });  *// the canvas/window resize event handler*              window.addEventListener('resize', function(){                  engine.resize();              });          });      </script>  </body>  </html> |

J’ai très simplement suivi le tutoriel que le site propose pour faire ses premiers pas avec la technologie.

Et il suffit de quelques lignes de code, quelques minutes pour établir une scène avec un sol, un objet et une caméra avec laquelle il est possible de se déplacer. Le rendu de ce simple morceau de code peut être directement ouvert dans un navigateur sans serveur ni add-on.



Babylonjs, comme la majorité des solutions testées, utilise l’objet html « canvas » dans lequel la scène va être générée.

La dernière mise à jour de la bibliothèque date du 29/06/2020 (rédigé le 02/07/2020) et il s’agit de la 4.2.0-alpha.23.

Babylonjs sera retenu pour une étude plus poussée.

Ce qu’on aime chez Babylonjs :

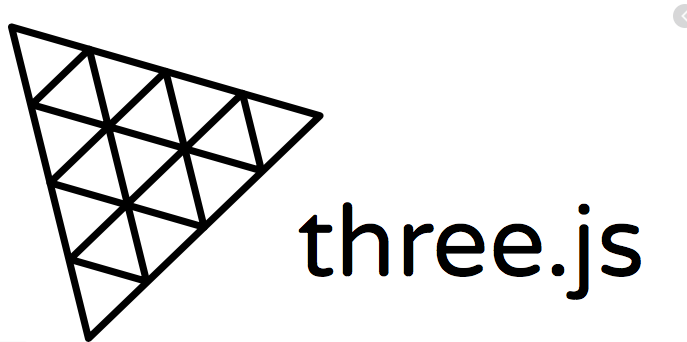
* Une communauté active, disponible et impliquée
* Un projet soutenu, « 12k likes » sur GitHub
* Une compatibilité JavaScript et TypeScript
* La simplicité d’utilisation
* Il embarque son propre moteur de jeu
* Pas besoin de plug-in navigateur pour le faire fonctionner
* Une chaine Youtube pleine de tutos et beaucoup de ressources
* Une documentation bien fournie
* Un projet OpenSource complètement gratuit
* Un GitHub à jour et complet
* Supporte tous les types d’import

Ce qu’on aime moins :

* C’est un moteur de jeu donc il est potentiellement moins verbeux sur certains points qui pourraient être importants.
* Un moteur de jeu peut se révéler gourmand en ressources.

# Technologie n°2 : Three.js

Threejs est un moteur 3D qui se présente sous la forme d’une bibliothèque JavaScript. Créée en 2010, elle est une des premières technologies qui a fait usage de WebGL pour le rendu 3D. Le langage de programmation est donc le JavaScript et c’est une des bibliothèques JS les plus suivie sur GitHub.



Plus verbeux que Babylonjs, ce n’en est pas spécialement plus compliqué. Le tutoriel est très (trop) simple et s’exécute directement dans une page web sans aucun serveur ni rien. Ce dernier n’étant pas assez complexe pour se rendre compte des possibilités de la technologie, j’ai voulu concevoir une scène semblable à celle de Babylonjs.

Plus de lignes, plus de technique, c’est un peu moins « beginner-friendly ». Après quelques heures dans la documentation (très complète), après avoir importé le projet entier dans mon espace de travail, mis en place un serveur Jetty, j’ai pu exécuter ce code ci-dessous et obtenir grossièrement le même résultat que lors de mon test de Babylonjs.

|  |
| --- |
| *import* \* *as* THREE *from* "../threejs/build/three.module.js"  *import* { FirstPersonControls } *from* '../threejs/examples/jsm/controls/FirstPersonControls.js';  *import* Stats *from* '../threejs/examples/jsm/libs/stats.module.js';  let container;  let camera;  let renderer;  let controls;  let scene;  let cube;  let stats;  let clock;  let ground;  function init() {      container = document.querySelector('#scene-container');      scene = new THREE.Scene();    scene.background = new THREE.Color('skyblue');    createCamera();    createControls();    createLights();    createMeshes();    createRenderer();    clock = new THREE.Clock();    stats = new Stats();    container.appendChild(stats.dom);    renderer.setAnimationLoop(() => {      update();      render();    })  }  function createCamera() {      camera = new THREE.PerspectiveCamera(      35, *//FOV*      container.clientWidth / container.clientHeight, *//aspect*      0.1, *//near clipping space*      100 *//far clipping space*    );    camera.position.set( 0, 0, 10 );  }  function createLights() {    const ambiantLight = new THREE.HemisphereLight(      0xddeeff,      0x202020,      3    )    const mainLight = new THREE.DirectionalLight(0xffffff, 5);    mainLight.position.set(10, 10, 10);    scene.add(ambiantLight, mainLight);  }  function createMeshes() {      let geometry = new THREE.BoxBufferGeometry(2,2,2);      let material = new THREE.MeshStandardMaterial( {        color: 0x808080,      transparent: false,      opacity:1      });      cube = new THREE.Mesh(geometry, material);    geometry = new THREE.PlaneBufferGeometry(20, 20, 20);    material = new THREE.MeshBasicMaterial( {        color: 0x808080,      side : THREE.DoubleSide    });    ground = new THREE.Mesh( geometry, material);    ground.position.y = -3;    ground.rotation.x = Math.PI/2;      scene.add( cube, ground );  }  function createRenderer() {      renderer = new THREE.WebGLRenderer({ antialias: true});      renderer.setSize(container.clientWidth, container.clientHeight);      renderer.setPixelRatio(window.devicePixelRatio);    renderer.physicallyCorrectLights = true;      container.appendChild(renderer.domElement);  }  function createControls() {      controls = new FirstPersonControls(camera, container);    controls.movementSpeed = 10;    controls.lookSpeed = 0.1;  }  function update() {    stats.update();  }  function render() {    controls.update( clock.getDelta() );    renderer.render(scene, camera);  }  function onWindowResize() {  *// set the aspect ratio to match the new browser window aspect ratio*    camera.aspect = container.clientWidth / container.clientHeight;  *// update the camera's frustum*    camera.updateProjectionMatrix();  *// update the size of the renderer AND the canvas*    renderer.setSize( container.clientWidth, container.clientHeight );    controls.handleResize();  }  window.addEventListener( 'resize', onWindowResize );  init(); |

Ceci est le contenu du fichier app.js que j’importe dans mon « index.html » qui lui-même contient un objet « canvas ».

Plus verbeux que le moteur de jeux, le moteur 3D tend à laisser le codeur gérer une plus grande partie des paramètres.

Tenue à jour très régulièrement, la dernière révision date du 27 juin 2020, il s’agit de la r118.

Three.js sera retenu pour une étude plus poussée.

Ce qu’on aime chez Threejs :

* C’est une technologie aboutie, mûre, très complète
* C’est une 5 plus grosses bibliothèque JavaScript de GitHub
* Un soutien et un suivi sans faille : « 60k likes », +1300 contributeurs sur GitHub, des tutos, un livre…
* Une bibliothèque complètement libre et open source
* Des exemples et des ressources en grande quantité
* Des mises à jour régulières
* Support de tous les types d’import
* Une utilisation de JavaScript qui est tout de même assez intuitive
* Se contente de n’importe quel petit serveur web

Ce qu’on aime moins chez Threejs :

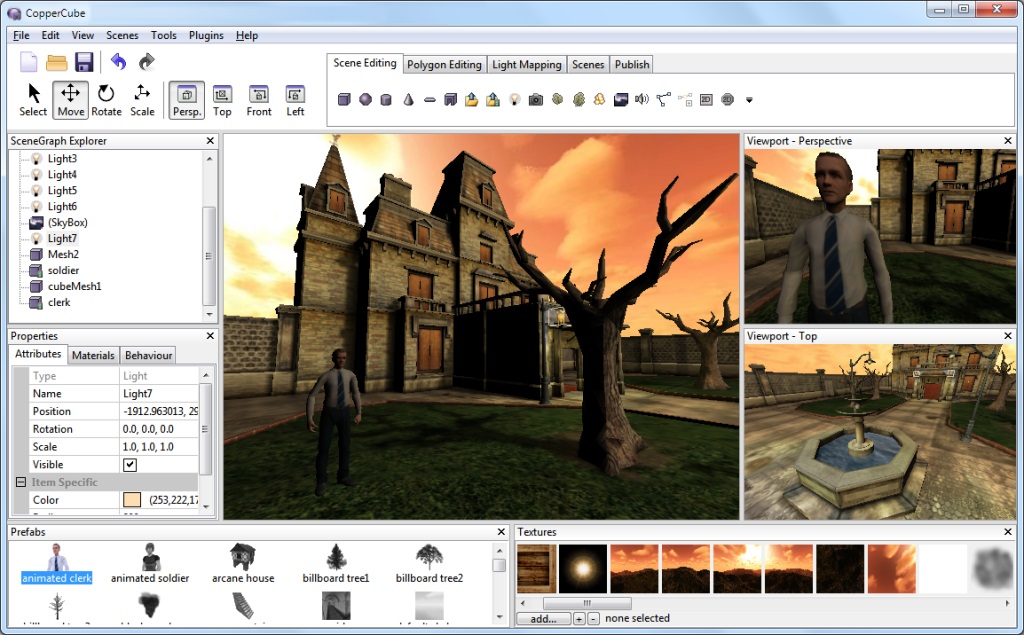
* Plus compliqué à mettre en place
* Plus de lignes de code
* C’est un moteur 3D, donc face à un moteur de jeu comme Babylonjs, il a ses forces et faiblesses, notamment le fait que l’on doive toucher à des notions de mathématiques ou de 3D parfois assez avancées pour parvenir à programmer ce que le moteur de jeu implémente seul à travers une fonction toute prête.
* Les développeurs qui arrivent en 2020 sur la 3D Web s’accordent globalement à dire que Threejs donne parfois l’impression d’être un peu dépassé par son temps.

# Technologie n°3 : CopperCube

CopperCube est un moteur de jeu qui n’est composé que d’un éditeur graphique. Développé par Ambierra, il est destiné en priorité aux non-programmeurs, de par le fait qu’on ne trouve pas de code dans l’édition du jeu ou du rendu 3D. Il peut générer le rendu pour l’exécuter dans un navigateur, en .exe et sous d’autres formats.



On peut tout de même éditer des scripts qui vont faire interagir les composants entre eux mais cela se limite à relativement peu. Basé sur le « drag and drop » et un système orienté logique d’action, l’interaction avec l’interface et la création de contenu est véritablement rapide et simple. Il est possible d’importer des modèles et ses propres conceptions dans l’éditeur. Une fois le rendu souhaité obtenu, il est possible de lancer le résultat dans un navigateur via WebGL. Il est même possible de demander un rendu directement dans Windows avec un « .exe ».



Globalement, l’interface est bien pensée et j’ai pu créer une scène « complexe » (composée de multiples éléments eux-mêmes complexes comme des bâtiments ou des personnages, parfois même animés) facilement et en très peu de temps. En revanche la technologie n’est pas du tout adaptée à l’élaboration d’un synoptique. En effet le fait qu’on ne code rien et que l’on créé la scène rend l’élaboration d’un synoptique impossible.

Par conséquent cette technologie ne sera pas retenu pour une étude plus poussée.

# Technologie n°4 : WRLD3D

Wold3D est un ensemble de plusieurs technologies qui ont pour but de vouloir modéliser le monde en 3D. Chaque bâtiment, et l’intérieur de chacun d’entre eux, pouvoir afficher les horaires des bureaux de manière dynamique, etc. Et à terme être capable de faire quelque chose de semblable à ce que le projet Google Lens est déjà en train de mettre au point. WRLD3D voudrait pouvoir offrir un monde modéliser entièrement en 3D avec de la réalité augmentée dans cette modélisation pour afficher des informations, itinéraire, etc. C’est une technologie qui utilise JavaScript.



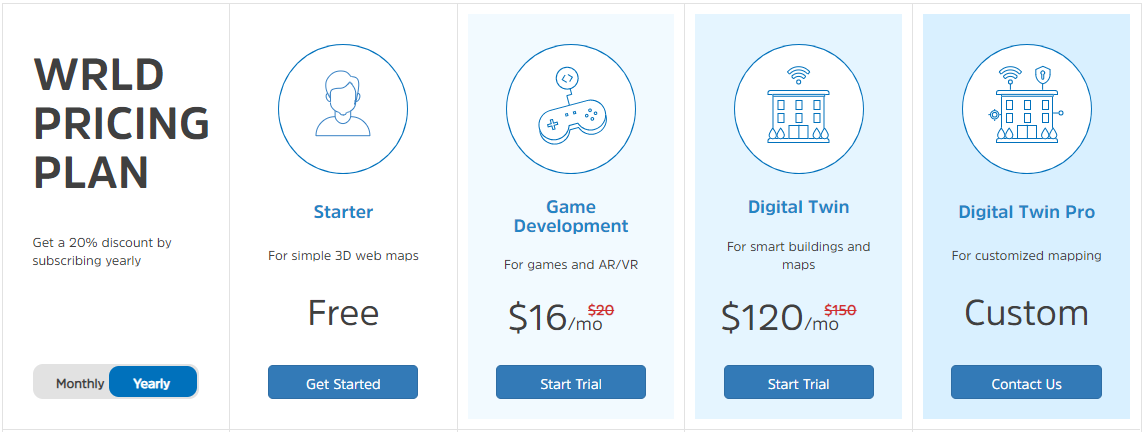
Quand je suis tombé pour la première fois sur cette technologie je savais déjà qu’elle ne serait pas adaptée aux besoins du projet. Mais il y avait une fonction qui m’intéressait particulièrement et c’était l’élaboration d’une maquette 3D d’un bâtiment à partir de plans 2D. Il est en effet possible d’implémenter l’intérieur des bâtiments générés en 3D sur les cartes du logiciel.

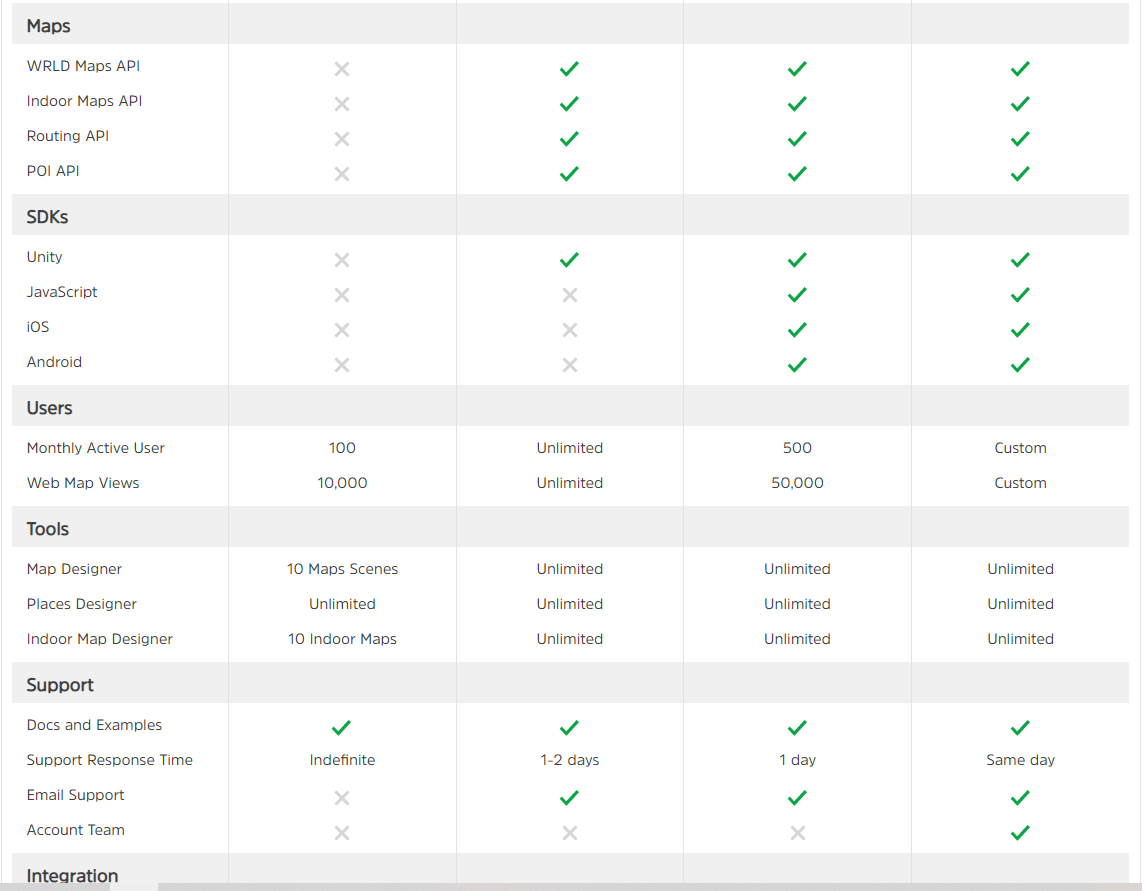


Le problème c’est que WRLD ne fournis aucun logiciel qui est capable que faire l’édition des plans 2D compatibles avec leur solution. Ils ne servent qu’à importer le plan spécial dans leur carte pour ensuite afficher l’intérieur du bâtiment en 3D.



Dans cette technologie, rien n’est gratuit à par l’utilisation des cartes que l’on peut importer et utiliser dans des projets divers.





Rien dans cette solution n’est exploitable pour notre synoptique, pour cette raison WRLD ne sera pas dans la liste des sélectionnés pour l’étude en profondeur de la technologie.

# Technologie n°5 : Whs.js

WHSJS ou Whitestormjs est une bibliothèque JavaScript qui encapsule le moteur de threejs en se ventant de mieux gérer les physiques et de simplifier la syntaxe de threejs.



C’est un projet entièrement JavaScript commencé en 2015 et qui aurait pu être intéressant de par la simplicité qu’il se dit apporter à threejs qui parfois manque un peu d’abstraction.

Cependant c’est tout de même un risque car l’encapsulation d’une technologie aussi complexe qu’un moteur 3D doit être mise à jour à l’unisson des mises à jour du moteur, etc.

J’ai donc eu l’occasion d’essayer whsjs qui se présente d’une autre manière que ce que j’ai pu voir jusque-là :

|  |
| --- |
| *import* \* *as* WHS *from* '../whs.js-dev/build/whs.module.js';  *import* \* *as* THREE *from* '../lib/three.js';  const app = new WHS.App([      new WHS.app.ElementModule(),      new WHS.app.SceneModule(),      new WHS.app.CameraModule({        position: new THREE.Vector3(0,0,50)      }),      WHS.app.RenderingModule({bgColor: 0x162129}),      new WHS.app.ResizeModule()    ]);  const sphere = new WHS.Sphere({ *// Create sphere component.*      geometry: {        radius: 3,        widthSegments: 32,        heightSegments: 32      },        material: new THREE.MeshBasicMaterial({        color: 0xF2F2F2      }),        position: [0, 10, 0]    });      sphere.addTo(app); *// Add sphere to world.*    new WHS.Plane({      geometry: {        width: 100,        height: 100      },        material: new THREE.MeshBasicMaterial({        color: 0x447F8B      }),        rotation: {        x: - Math.PI / 2      }    }).addTo(app);      app.start(); *// Start animations and physics simulation.* |

En effet ici on déclare « app » et on utilise cette dernière pour contenir l’ensemble de notre scène et de ses composants. L’idée n’est pas mauvaise, et le pari est tenu dans le sens que cette syntaxe est tout de même plus courte que celle de threejs. Surtout les dernières lignes, avec le « app.start() » qui permet de générer la scène sans faire appel aux méthodes « update », « renderloop »…

Cependant ce code ne démarre pas et quoi que je puisse modifier j’obtiens toujours la même erreur :



Que ce soit dans les exemple censés fonctionner que j’importe depuis le web directement ou depuis mes fichiers les plus simples et les plus proches du tuto, je n’arrive pas à dépasser cette exception.

Il s’agit d’une erreur dans le fichier source de whitestorm « whs.js » que je n’arrive pas à résoudre avec mes connaissances assez limitées en JavaScript pour le moment, et avec cette bibliothèque.

La dernière mise à jour de la technologie a été mise en ligne il y a 3 ans le 8 août 2017 et il s’agissait de la « release 2.1.6. ». Cette dernière information me fait dire que l’erreur relève peut-être d’une obsolescence de la technologie.

Whs a beau proposer un projet et une syntaxe intéressante, la communauté n’y est presque plus active et s’il est si difficile de créer une scène basique sans obtenir une erreur, alors cela risque de compromettre la mise en place d’un synoptique. Plus les révisions de threejs avancent et sortent et plus whs va avoir du mal à rattraper son retard.

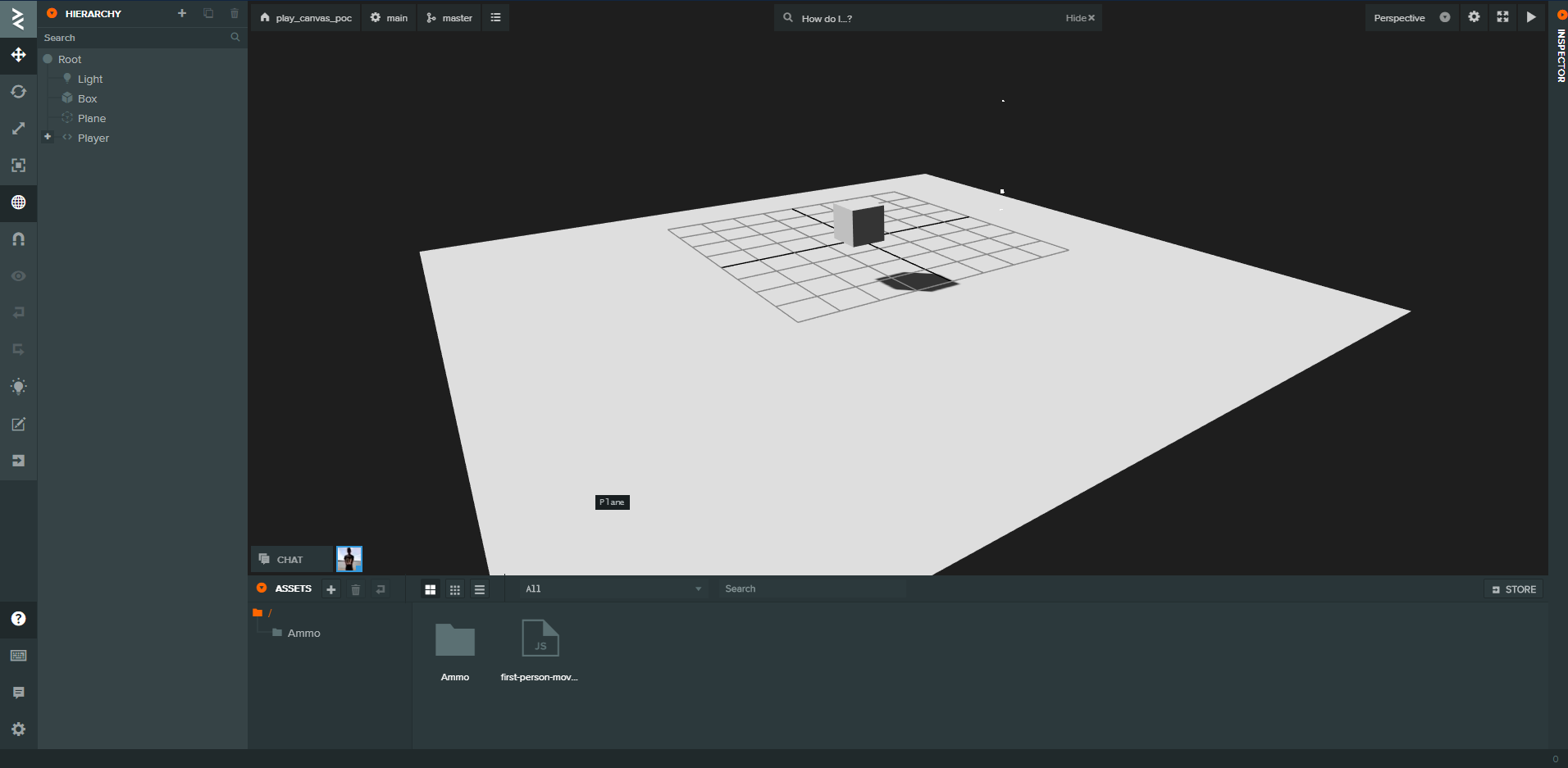
WhitestormJS ne fera pas partie des sélectionnés pour l’étude plus poussée.

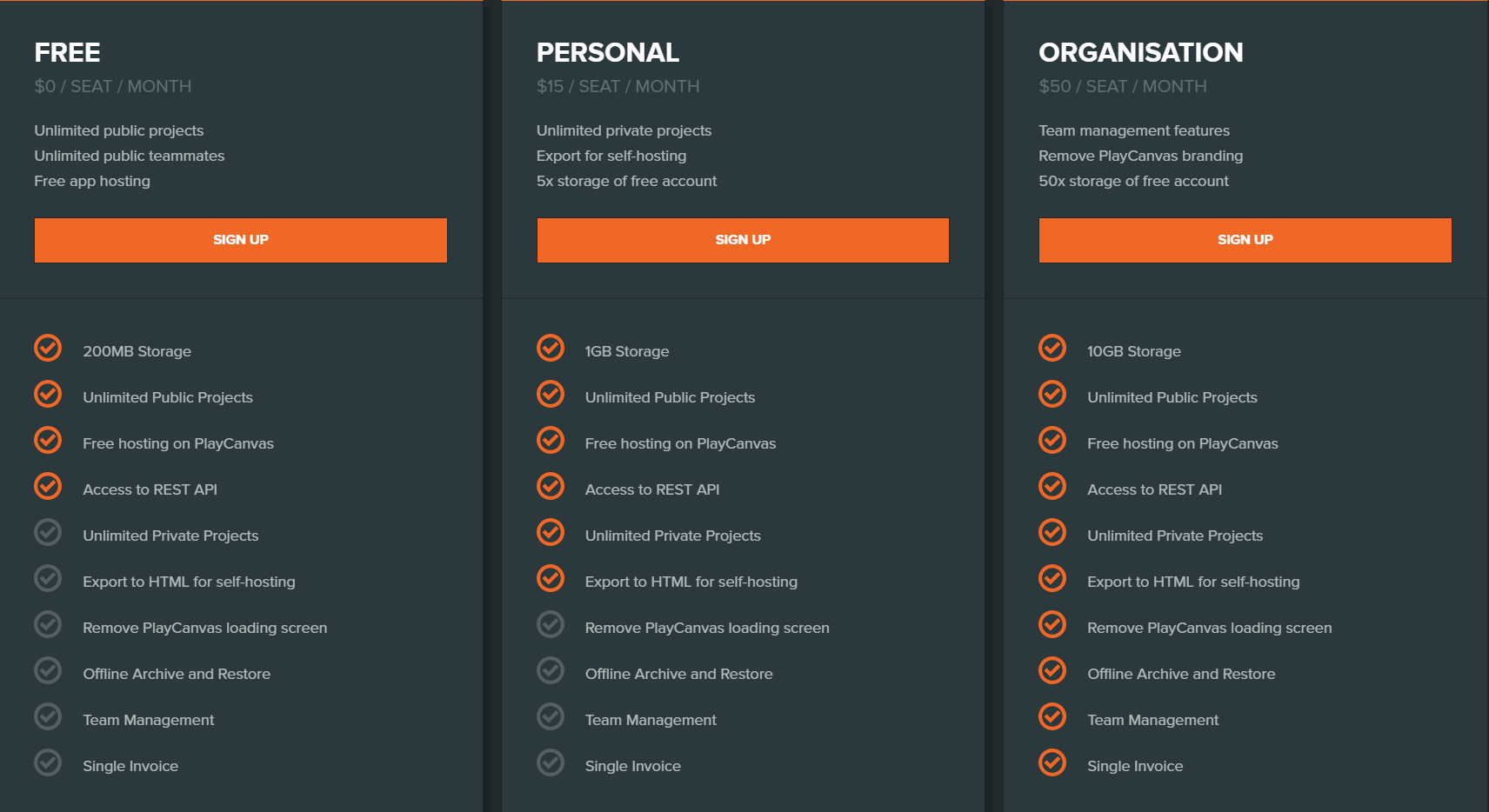
# Technologie n°6 : PlayCanvas

PlayCanvas est un moteur de jeu OpenSource qui supporte WebGL pour assurer l’exécution des projets dans tous les navigateurs récents. Il permet de créer rapidement et facilement des scènes et des objets en les faisant interagir avec des scripts JS. License MIT, il a été fondé en 2014 dans le même but que CopperCube : faciliter le développement de jeux sur navigateur.



Arborant un puissant mais simple éditeur à même le navigateur, il est possible en très peu de temps, d’arriver à des résultats assez cohérents en terme de « game-design ».



PlayCanvas est disponible en version gratuite et décliné en plusieurs autres versions : 

L’interaction entre les différents éléments du décor se fait avec des script JS comme le suivant qui sert à associer des contrôles clavier à une caméra :

|  |
| --- |
| var FirstPersonMovement = pc.createScript('firstPersonMovement');  FirstPersonMovement.attributes.add('camera', {      type: 'entity',      description: 'Optional, assign a camera entity, otherwise one is created'  });  FirstPersonMovement.attributes.add('power', {      type: 'number',      default: 2500,      description: 'Adjusts the speed of player movement'  });  FirstPersonMovement.attributes.add('lookSpeed', {      type: 'number',      default: 0.25,      description: 'Adjusts the sensitivity of looking'  });  *// initialize code called once per entity*  FirstPersonMovement.prototype.initialize = function() {      this.force = new pc.Vec3();      this.eulers = new pc.Vec3();        var app = this.app;    *// Listen for mouse move events*      app.mouse.on("mousemove", this.\_onMouseMove, this);  *// when the mouse is clicked hide the cursor*      app.mouse.on("mousedown", function () {          app.mouse.enablePointerLock();      }, this);  *// Check for required components*  *if* (!this.entity.collision) {          console.error("First Person Movement script needs to have a 'collision' component");      }  *if* (!this.entity.rigidbody || this.entity.rigidbody.type !== pc.BODYTYPE\_DYNAMIC) {          console.error("First Person Movement script needs to have a DYNAMIC 'rigidbody' component");      }  };  *// update code called every frame*  FirstPersonMovement.prototype.update = function(dt) {  *// If a camera isn't assigned from the Editor, create one*  *if* (!this.camera) {          this.\_createCamera();      }        var force = this.force;      var app = this.app;  *// Get camera directions to determine movement directions*      var forward = this.camera.forward;      var right = this.camera.right;    *// movement*      var x = 0;      var z = 0;  *// Use W-A-S-D keys to move player*  *// Check for key presses*  *if* (app.keyboard.isPressed(pc.KEY\_Q)) {          x -= right.x;          z -= right.z;      }  *if* (app.keyboard.isPressed(pc.KEY\_D)) {          x += right.x;          z += right.z;      }  *if* (app.keyboard.isPressed(pc.KEY\_Z)) {          x += forward.x;          z += forward.z;      }  *if* (app.keyboard.isPressed(pc.KEY\_S)) {          x -= forward.x;          z -= forward.z;      }  *// use direction from keypresses to apply a force to the character*  *if* (x !== 0 && z !== 0) {          force.set(x, 0, z).normalize().scale(this.power);          this.entity.rigidbody.applyForce(force);      }  *// update camera angle from mouse events*      this.camera.setLocalEulerAngles(this.eulers.y, this.eulers.x, 0);  };  FirstPersonMovement.prototype.\_onMouseMove = function (e) {  *// If pointer is disabled*  *// If the left mouse button is down update the camera from mouse movement*  *if* (pc.Mouse.isPointerLocked() || e.buttons[0]) {          this.eulers.x -= this.lookSpeed \* e.dx;          this.eulers.y -= this.lookSpeed \* e.dy;      }  };  FirstPersonMovement.prototype.\_createCamera = function () {  *// If user hasn't assigned a camera, create a new one*      this.camera = new pc.Entity();      this.camera.setName("First Person Camera");      this.camera.addComponent("camera");      this.entity.addChild(this.camera);      this.camera.translateLocal(0, 0.5, 0);  }; |

On peut constater la longueur de ce morceau de code, non négligeable. De plus il s’agit d’une fonctionnalité que les moteurs 3D et le moteur de jeu testé avant ont sous forme d’une simple fonction ou objet, donc c’est tout de même regrettable de ne pas avoir une simple case à cocher pour faire bénéficier notre caméra de ce genre de comportement.

En résumé, cette technologie ne fera pas partie des sélectionnées de par le fait qu’elle ne peut pas coller au projet du synoptique 3D à la même manière que CopperCube. Pour tester des scènes rapidement, oui, pour le développement d’un synoptique, non.

# Technologie n°7 : XeoGL

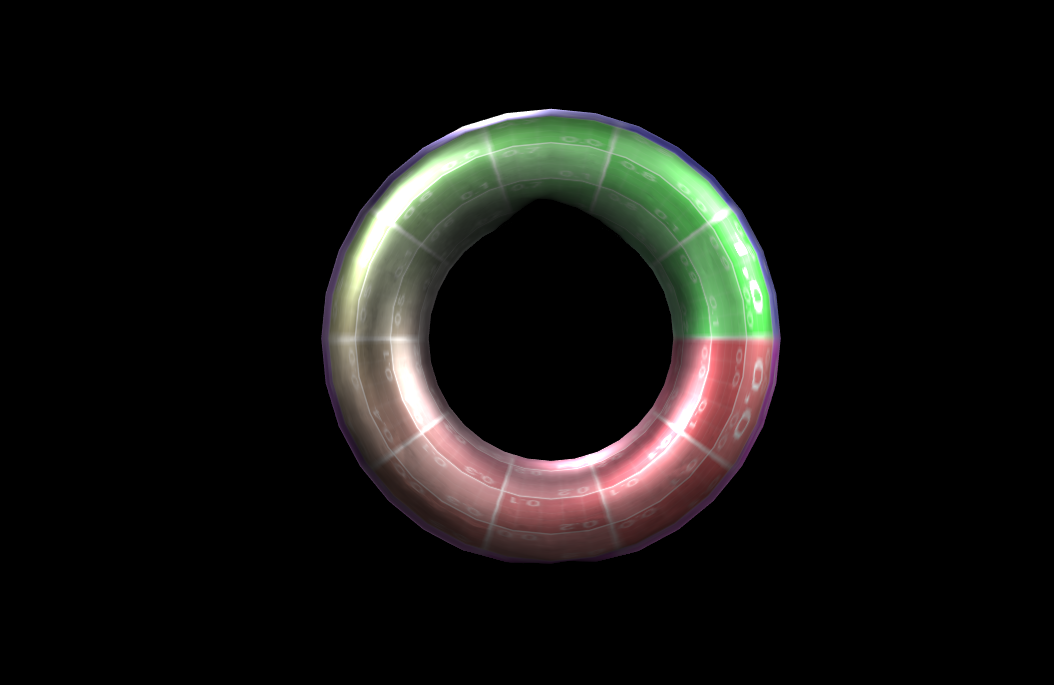
Bibliothèque JavaScript orientée visualisation 3D via WebGL, XeoGL embarque son propre moteur 3D. Avec un large nombre de supports d’import pris en charge, XeoGL est un projet qui a commencé en 2016 et dont la dernière version date de 2018.



Le tutoriel est orienté import et visualisation de matériaux ou formes faisant déjà partis du projet XeoGL.

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <script src="./js/xeogl.js"></script>      <img src="./js/goldRoughness.jpg"></img>      <img src="./js/uvGrid2.jpg"></img>      <title>xeogl - poc</title>  </head>  <body>      <script>          var geometry = new xeogl.TorusGeometry({          radius: 1.0,          tube: 0.3      });      var material = new xeogl.MetallicMaterial({          baseColorMap: new xeogl.Texture({          src:"./js/goldRoughness.jpg"      }),          roughnessMap: new xeogl.Texture({          src:"./js/uvGrid2.jpg"      })      });      var mesh = new xeogl.Mesh({          geometry: geometry,          material: material,          position: [0, 0, 10]      });      </script>  </body>  </html> |

Le code ci-dessus permet l’obtention de ce résultat :



La technologie est prometteuse, si elle facilite véritablement les imports et la visualisation de modèles dans l’espace, alors elle doit faire partie des sélectionnées.

Ce qu’on aime chez XeoGL :

* Syntaxe claire et simple
* Gratuit, OpenSource
* Adapté/Développé pour la visualisation de modèle ce qui colle avec le projet
* Support un grand nombre de formats d’import

Ce qu’on aime moins chez XeoGL :

* Plus de mise à jour d’un gros moment
* Pas beaucoup de suivis sur GitHub (800 stars)

# Technologie n°8 : Cesium

Cesium n’est en fait pas vraiment une simple bibliothèque 3D, il s’agit en fait d’un ensemble de solutions dont une bibliothèque 3D JS qui ont pour but de faciliter la mise en place et la visualisation de cartes 3D et de globes terrestres 3D.

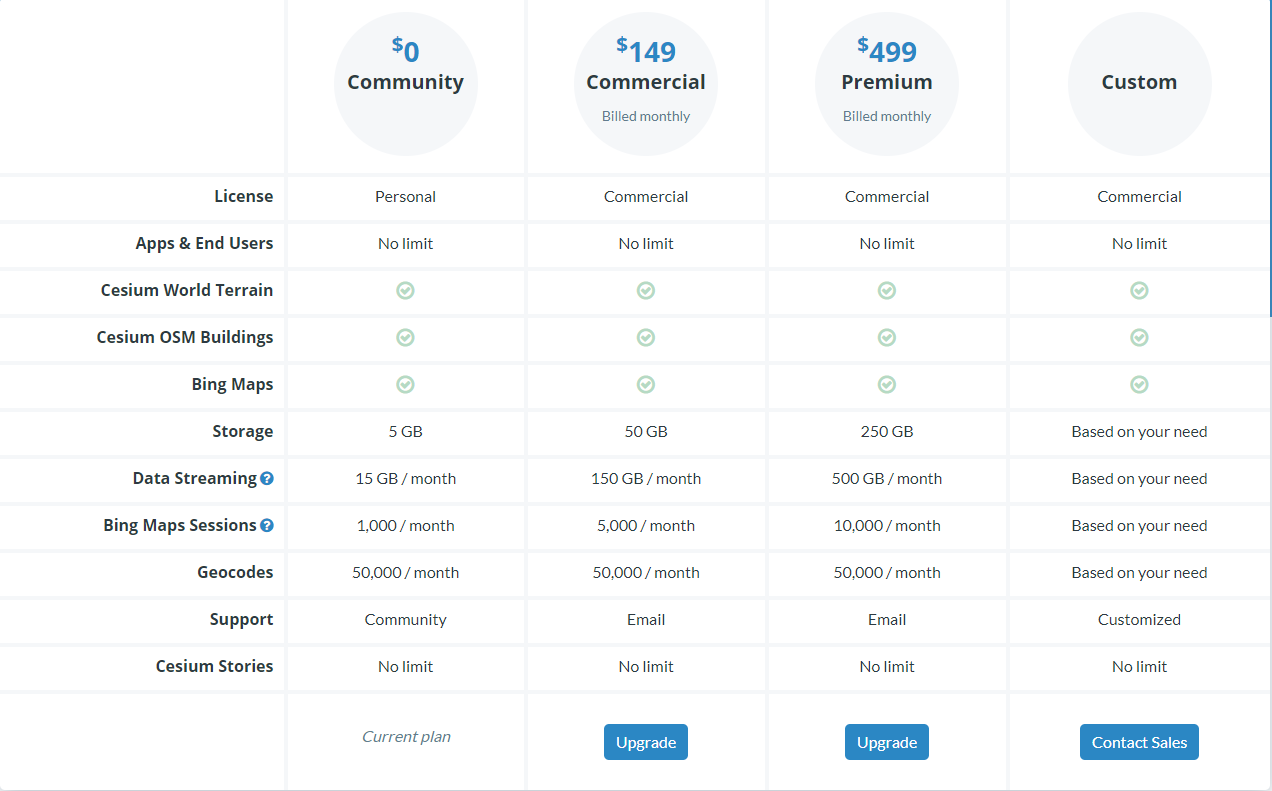
Cela ressemble fortement à la solution de WRLD à première vue et c’est d’ailleurs pour la même raison que je l’ai mis dans la liste des tests.

Malheureusement, la première des choses qui est certaines est que nous n’allons pas pousser la recherche plus loin. Bien qu’intéressante, la technologie de Cesium ne correspond pas vraiment au projet du synoptique 3D.

Je pensais qu’ils avaient un produit capable de transformer des plans 2D en objet 3D ou scène, mais rien de cela sans un travail d’édition long et fastidieux des plans 2D avant, ce qui n’est pas le but recherché.

De plus leur moteur JS 3D n’est fait que pour faciliter la mise en place de cartes ou globes terrestres, ce qui n’est pas vraiment intéressant pour les utilisations que l’on aura du moteur 3D.





Leurs solutions peuvent devenir payantes en fonction de l’activité, et encore une fois, nous pouvons constater que cette solution n’est destinée qu’à des entreprise ou particuliers qui veulent se servir de cartes de monde et de globes.

Elle ne fera donc pas partie des sélectionnées pour aller plus loin dans la recherche.

# Technologie annexe n°1 : 3D Builder

Les technologies annexes sont là pour rapporter quelques informations sur des solutions qui pourraient être utiles dans le développement ou la mise en service du synoptique 3D.

3D Builder est un outil gratuit développé par Microsoft souvent fournis de base avec Windows 10. Il s’agit d’un modeleur 3D pour l’usage d’imprimante 3D.

S’il n’est pas spécialement conçu pour cela, il peut aussi très bien créer à partir d’une image, un objet 3D prêt à être enregistré au format STL, OBJ… Cela peut se révéler utile pour l’intégration de plans dans le synoptique 3D.

# Conclusion : Choix des technologies

Les technologies qui seront expérimentées avec plus d’approfondissement sont Threejs, Babylonjs et XeoGL.

Le tableau suivant liste les fonctionnalités ou critères sur lesquels est basée la sélection. Le tableau ne mentionnera pas WRLD3D et Cesium car les technologies sont tout simplement trop loin de ce qui nous recherchons.

Je n’ai pas obtenu de résultats concluants avec les essais sur WHSJS donc ce tableau se basera sur l’expérience lors du codage et sur la documentation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Critère\Technologie | Babylonjs | Threejs | CopperCube | Whsjs | PlayCanvas | XeoGL |
| Technologie JavaScript | Oui | Oui | Oui | Oui | Oui | Oui |
| Propose une bibliothèque | Oui | Oui | Non | Oui | Non | Oui |
| Embarque son propre moteur | Oui | Oui | Oui | Non | Oui | Oui |
| Création POC simple facile | Oui | Moyen | Oui | Non (Pas à jour ?) | Oui | Oui |
| Camera contrôlable facilement | Oui (Type de caméra) | Oui (Type de contrôles à appliquer à une caméra) | Oui (de base) | Oui | Oui (il faut écrire un script assez complexe pour gérer le tout) | Oui (type de caméra) |
| Permet l’import de fichiers | Oui (tous types de fichiers) | Oui (tous types de fichiers) | Oui (tous types de fichiers) | Oui (mais certainement pas plus récents) | Oui (tous types de fichiers) | Oui (tous types de fichiers) |
| Maintenu/A jour régulièrement | Oui | Oui | 1 fois tous les ans/2 ans | Non | Oui | Non |
| Documentation présente et complète | Oui | Oui | Oui | Confuse (plusieurs sites…) | Oui | Oui mais confuse (erreur dans les redirection) |
| Support réactif | Oui | Oui | Pas certain | Non | Oui | Pas certain |
| Implémentation du « Focus » simple | Oui | Oui | Pas certain | Oui | Certainement via un script | Oui |

# Sources

## WebGL :

<https://caniuse.com/#search=WebGL>

<https://github.com/KhronosGroup/WebGL>

<https://www.wikiwand.com/en/WebGL>

<https://www.khronos.org/webgl/>

## Babylonjs :

<https://www.babylonjs.com/>

<https://github.com/BabylonJS/Babylon.js>

<https://en.wikipedia.org/wiki/Babylon.js>

## Threejs :

<https://threejs.org/>

<https://github.com/mrdoob/three.js/>

<https://www.wikiwand.com/en/Three.js>

<https://medium.com/@hiteshkrsahu/player-2-has-joined-the-game-three-js-vs-babylon-dea49bf00466>

## CopperCube :

<https://www.ambiera.com/coppercube/>

<https://www.wikiwand.com/en/CopperCube>

## WHS.JS

<https://github.com/WhitestormJS/whs.js/releases?after=v0.0.6>

<https://whs.io/>

## PlayCanvas

<https://playcanvas.com/>

<https://www.wikiwand.com/en/PlayCanvas>

## XeoGL

<https://xeogl.org/>

<https://github.com/xeolabs/xeogl>

## Cesium

<https://cesium.com/why-cesium/>