

Projet de BTS Système Numérique
– Informatique et Réseau

Éditeur de plan
de maison

Lycée Antoine Bourdelle -
Promotion 2018

Table des matières

I . Introduction.....	3
II . Solution envisagées.....	3
III . Éditeur matriciel.....	4
1 . Principe de fonctionnement.....	4
2 . Problèmes de ce modèle.....	5
3 . Tests.....	6
a . Initialisation.....	6
b . Création et destruction d'étage.....	6
c . Navigation dans les étages.....	8
d . Ciblage du curseur.....	9
IV . Éditeur vectoriel.....	10
1 . Principe de fonctionnement.....	10
2 . Raycast.....	12
3 . Tests.....	13
a . Initialisation.....	13
b . Création et destruction d'étage.....	13
c . Navigation dans les étages.....	14
d . Création des pièces.....	14
e . Sélection des pièces.....	17
f . Modification et destruction des pièces.....	18
g . Sauvegarde du plan.....	19
V . Travail restant.....	19
Annexes.....	20
I . Éditeur matriciel.....	20
II . Éditeur vectoriel.....	22

I . Introduction

Dans le projet de visite virtuelle, il nous faut pouvoir créer les visites sur tout les points. Un des points est qu'il faut pouvoir créer un éditeur de plan de maison, de manière schématique, et faire en sorte que cette éditeur soit facile d'utilisation.

II . Solution envisagées

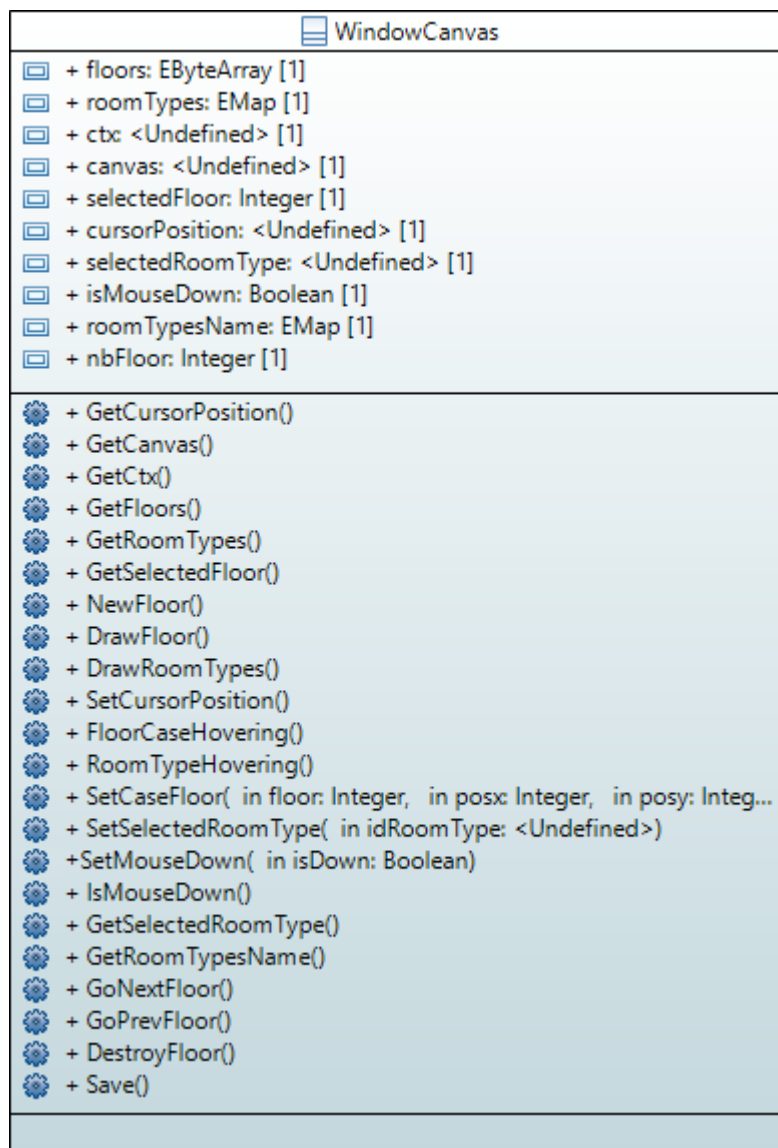
Il faut que l'éditeur soit réalisé avec du JavaScript pour qu'il soit directement utilisable par le site web. Le choix du JavaScript n'est pas anodin, car ce langage peut être interprété par n'importe quel navigateur, contrairement au flash, où il faut parfois installer un pilote puis l'activer au cas par cas, ce qui n'est pas pratique. Il doit aussi être simple d'utilisation, et que les données doivent être stocker de manière ordonné dans le programme, qui peut être fait facilement avec une approche objet. Et enfin l'application doit pouvoir sauvegarder le plan au format XML, pour que le plan puisse être réutiliser facilement.

Nous avons envisagé 2 méthodes. La première est avec une matrice, qui fut abandonné à cause des problèmes de sauvegarde, où les fichiers sont trop important, et une utilisation moins pratique.

La seconde méthode est avec des tracés vectoriels, ce qui simplifie sont utilisation et sa sauvegarde, et permet de faire des pièces plus complexe et plus facilement.

III . Éditeur matriciel

1 . Principe de fonctionnement



Le diagramme de classe suivant a été fait. Le problème de ce diagramme est qu'il n'y a qu'une seule classe où tout est regroupé. Ça a rendu le développement de cette version très difficile.

La première version de l'éditeur qui a été envisagé, et un éditeur matriciel, où nous sélectionnons un type de pièce grâce à un menu regroupant toutes les pièces différentes, et ensuite, une fois que le type de pièce est sélectionné, nous pouvons l'assigner à une case de la matrice. Nous pouvons créer plusieurs étages, et tout est sauvegardé en mémoire.

Voici le résultat où le développement s'est arrêté.

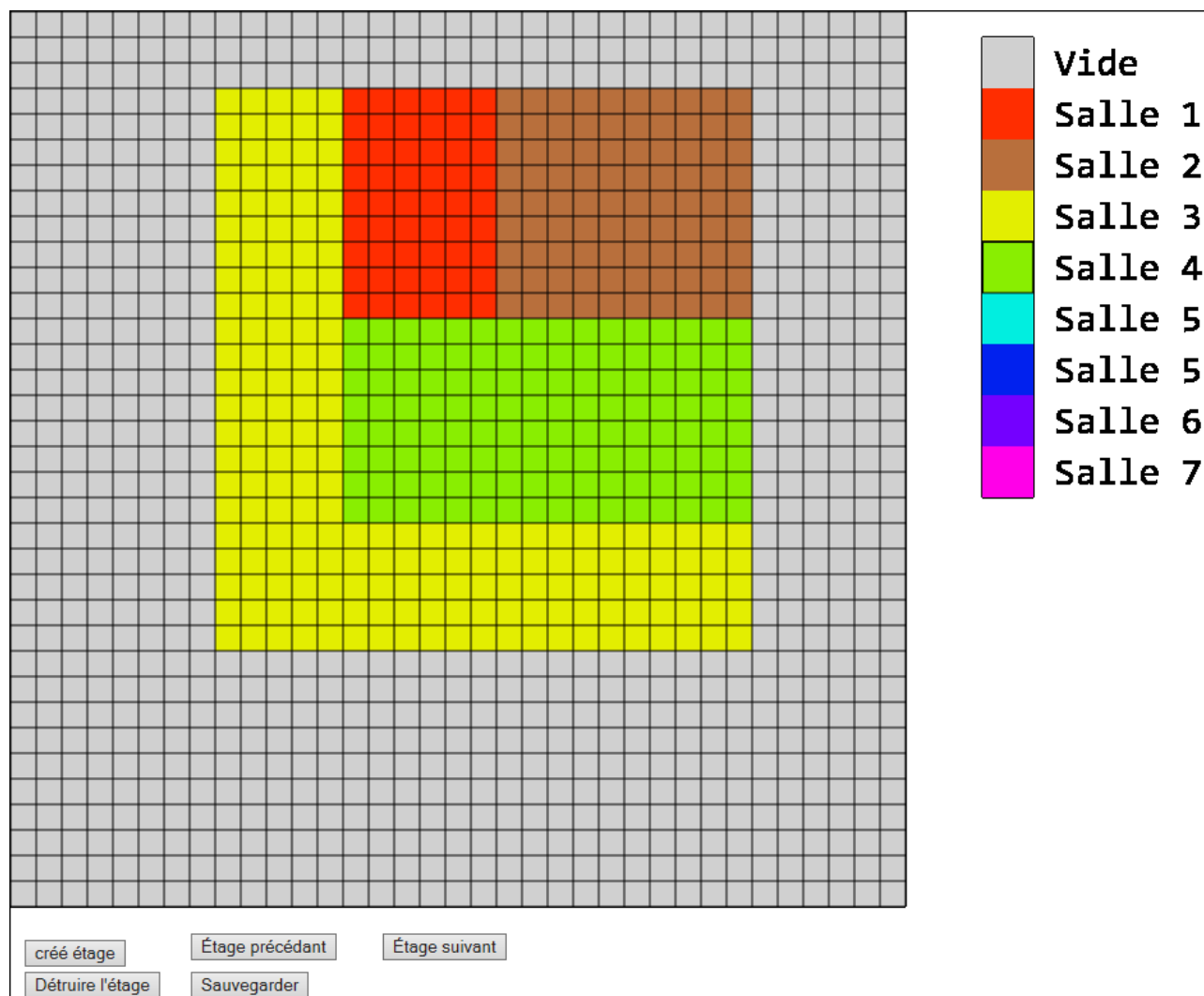
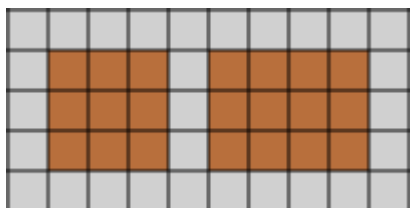


Illustration 1: Exemple de l'éditeur avec sélecteur de pièce

2 . Problèmes de ce modèle

Nous pouvions assigner des cases de manière fluide grâce à un clic puis glisser de la souris, mais cela était long et laborieux. Pour palier à ce problème, une sélection par zone a été envisagée, mais le développement s'est arrêté à ce moment. Le second problème est que nous pouvions avoir un type de pièce mais à plusieurs endroits, comme nous pouvons le voir ci-dessous.



Ceci est une pièce dissociée, ce qui n'a pas de sens.

La raison principale pour laquelle cette version a été abandonnée, est que l'enregistrement du plan générerait des fichiers trop volumineux à cause des matrices, où des informations inutiles seraient stockées.

3 . Tests

a . Initialisation

Au lancement de l'application, nous lui demandons de créer un étage automatiquement.

```
Étage créé : 0
editorTest.js (123,3)
  ▲ [object Array]      [Array[35]]
    editorTest.js (124,3)
      ▶ 0                [object Array] [...]
        length           1
```

Ici nous avons le résultat du test, avec la ligne en surbrillance bleue qui témoigne de l'étage créé, qui est un tableau en deux dimension initialisé à 0.

b . Création et destruction d'étage

La création d'un étage se fait en cliquant sur un bouton sur l'IHM Web.

Résultat de l'appel côté javascript :

```
Étage créé : 1
editorTest.js (123,3)
  ▲ [object Array]      [Array[35], Array[35]]
    editorTest.js (124,3)
      ▶ 0                [object Array] [...]
      ▶ 1                [object Array] [...]
        length           2
```

Avec la ligne en surbrillance bleue qui témoigne bien du nouvel étage créé.

(voir Code de création d'étage)

Maintenant que nous savons créer un étage il faut savoir le détruire, et le bon, selon l'étage sélectionné. Nous avons 7 étages, et nous sommes sur l'étage 2.

```
▲ [object Array] [Array[35], Array[35], Array[35], Array[35], Array[35], Array[35], Array[35]]
editorTest.js (124,3)
  ▶ 0 [object Array] [...]
  ▶ 1 [object Array] [...]
  ▶ 2 [object Array] [...]
  ▶ 3 [object Array] [...]
  ▶ 4 [object Array] [...]
  ▶ 5 [object Array] [...]
  ▶ 6 [object Array] [...]
  length 7
étage sélectionné : 3
```

En supprimant l'étage nous remarquons qu'il ne reste plus que 6 étages, et notre étage sélectionné est resté à 2.

```
étage détruit
editorTest.js (267,4)
▲ [object Array] [Array[35], Array[35], Array[35], Array[35], Array[35], Array[35]]
editorTest.js (268,4)
  ▶ 0 [object Array] [...]
  ▶ 1 [object Array] [...]
  ▶ 2 [object Array] [...]
  ▶ 3 [object Array] [...]
  ▶ 4 [object Array] [...]
  ▶ 5 [object Array] [...]
  length 6
étage sélectionné : 2
```

(voir Code de destruction d'étage)

Nous devons avoir au minimum 1 étage. Lorsque nous essayons de détruire ce dernier étage il nous est impossible de le supprimer.

```
impossible de détruire cet étage
editorTest.js (272,4)
```

c . Navigation dans les étages

Pour pouvoir utiliser le programme il nous faut pouvoir naviguer simplement dans les étages créés.

```
étage suivant : 1  
editorTest.js (290,4)  
étage suivant : 2  
editorTest.js (290,4)  
impossible d'accéder a l'étage suivant  
editorTest.js (292,4)
```

Nous voyons bien que l'incrémentation s'est bien déroulée, jusqu'au moment où nous sommes arrivés au dernier étage, et l'incrémentation s'est bloquée.

```
étage précédant : 1  
editorTest.js (280,4)  
étage précédant : 0  
editorTest.js (280,4)  
impossible d'accéder a l'étage précédant  
editorTest.js (282,4)
```

Même chose pour passer aux étages inférieurs.

(voir Code de navigation entre les étages)

d . Ciblage du curseur

Pour la sélection des carrés, il nous faut savoir si nous nous trouvons dans la zone d'édition, quand nous sommes à l'extérieur de la zone nous obtenons ceci.

Résultat côté javascript en dehors de la grille	▷ [object Object] {x: null, y: null}
Résultat côté javascript sur la case 7;18	▷ [object Object] {x: 7, y: 18}

Ce qui est le résultat attendu.

A l'intérieur de la zone, nous obtenons bien les coordonnées de la bonne case.

Nous avons la même chose pour le sélecteur de pièces, avec les valeurs attendues.

Résultat côté javascript en dehors du sélecteur	▷ [object Object] {x: null, y: null}
Résultat côté javascript sur le sélecteur pièce 4	▷ [object Object] {x: 0, y: 4}

(voir Exemple de l'éditeur avec sélecteur de pièce page 5)

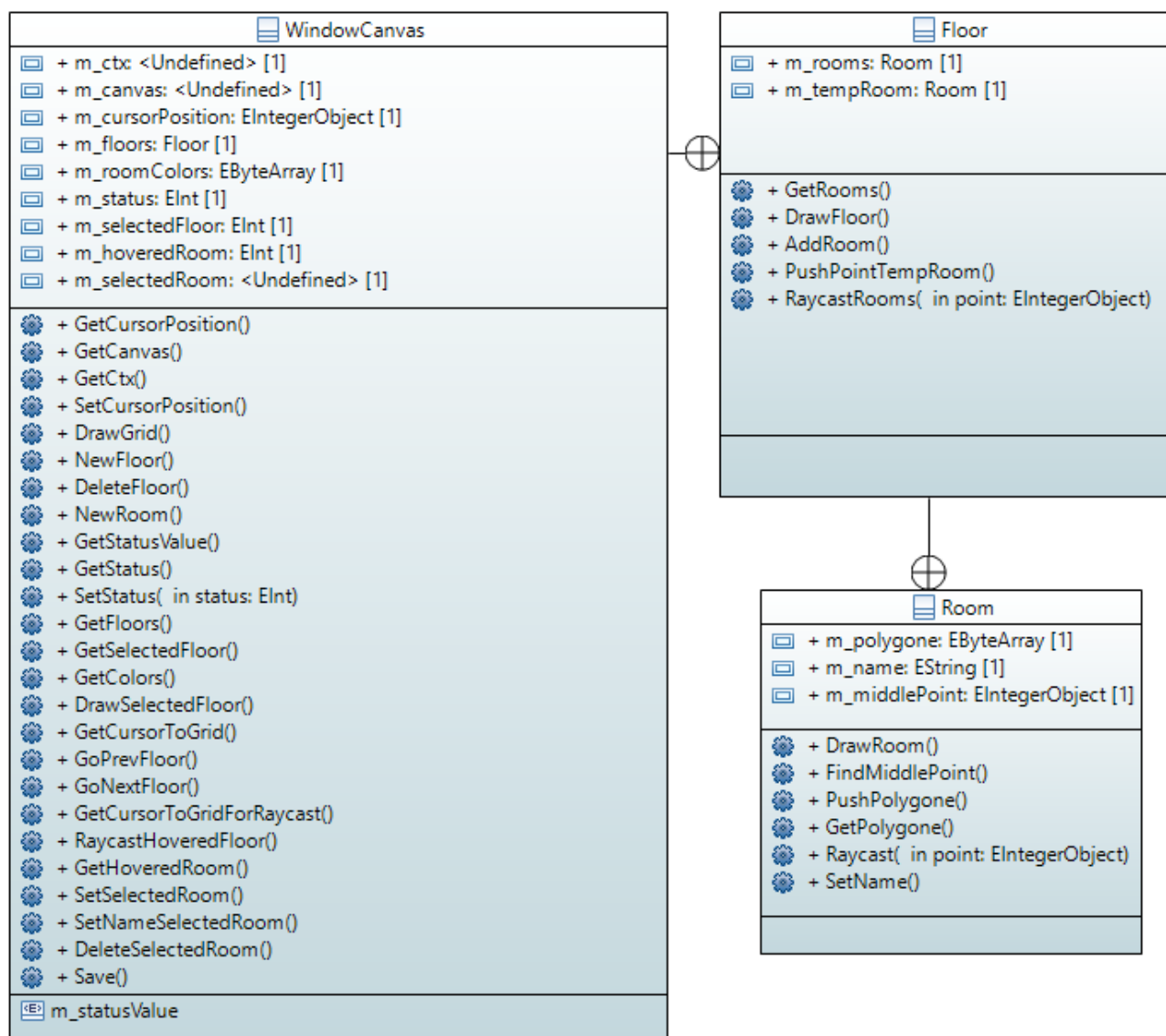
Le tableau du sélecteur étant en deux dimension, seule la coordonnée y est prise en compte. Mais il a été envisagé de le passer à deux dimension, c'est pourquoi la coordonnée x a été conservée, avec une valeur nulle.

(voir Code de ciblage de la grille)

IV . Éditeur vectoriel

1 . Principe de fonctionnement

Ci-dessous le diagramme de classe de l'éditeur de plan utilisant des vecteurs. Le précédent éditeur a été pris comme base, mais il a été réorganisé pour être amélioré.



Le but de cet éditeur est de s'affranchir des limites générées par l'utilisation de carrés juxtaposés pour la création de formes de pièces. Avec cette nouvelle méthode, il est possible de dessiner des pièces avec des pans coupés. (les pièces sont des polygones créées par des clics de souris, et les points sélectionnés viennent se coller à une grille.)

Il permet aussi un nombre de pièces illimité.

La création de pièces devient plus simple et efficace.

Voici le résultat final :

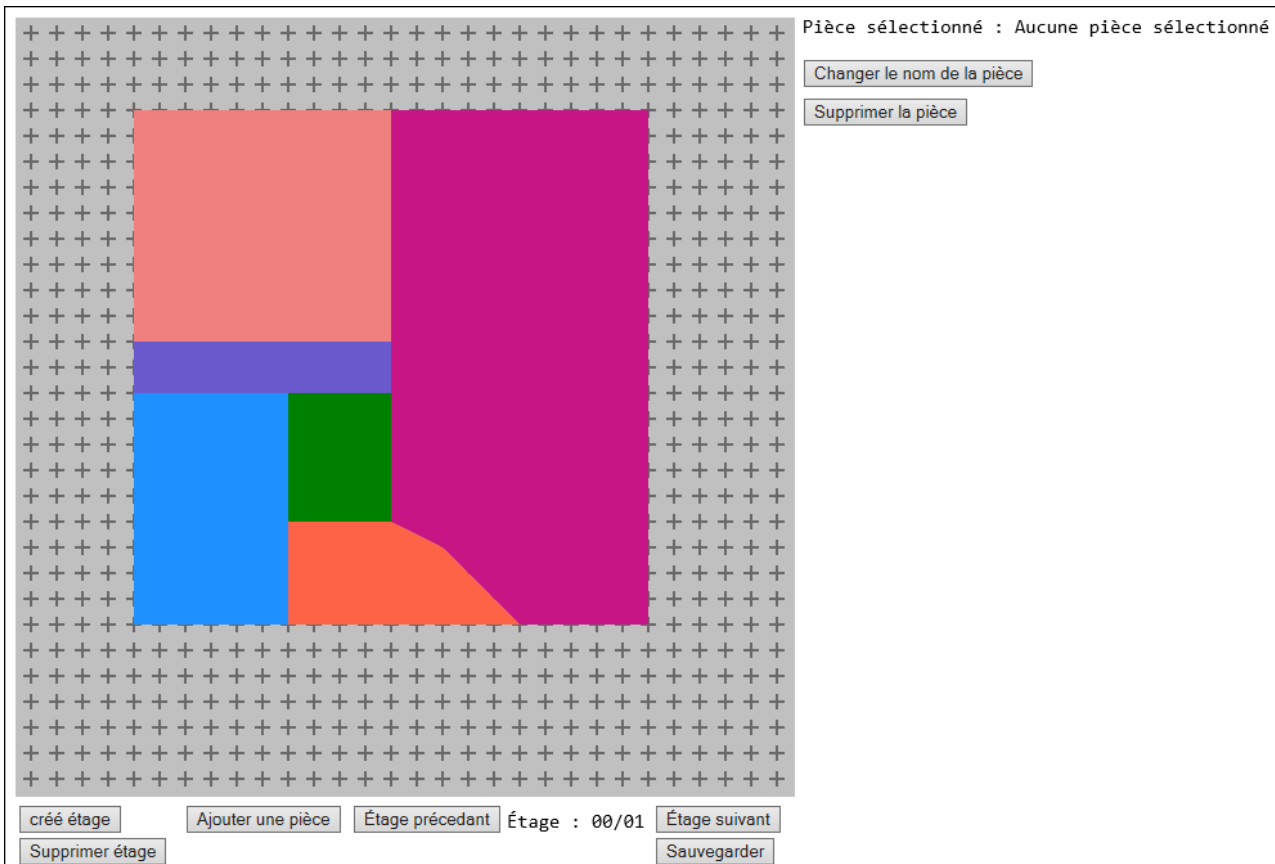
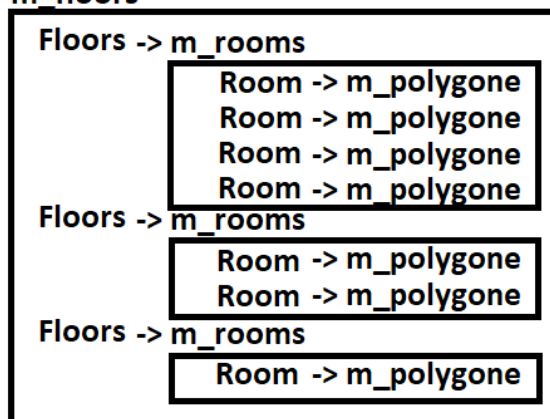


Illustration 2: Éditeur vectoriel

Le stockage des informations a été imaginé de cette manière.

m_floors

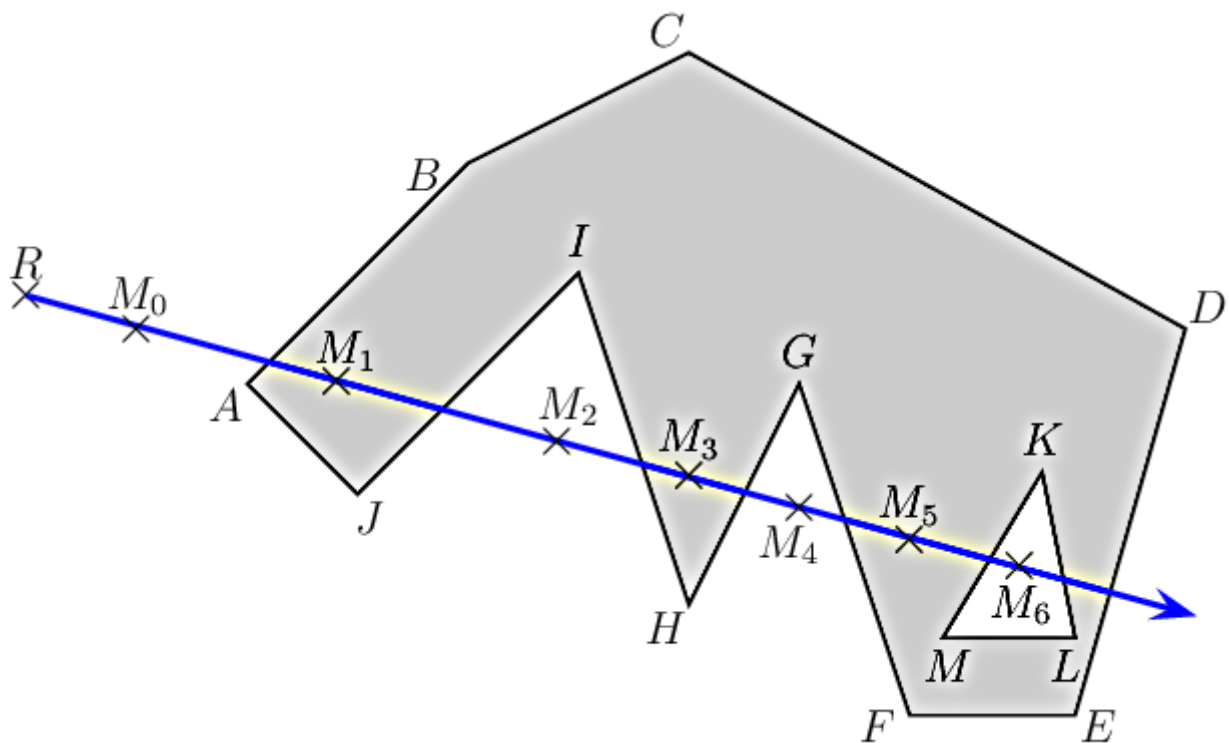


L'attribut `m_floors` contient les étages (floor) ayant l'attribut `m_rooms` contenant des pièces (room) contenant l'attribut `m_polygone` contenant des points et qui sert à afficher les pièces sur l'éditeur.

2 . Raycast

Pour pouvoir modifier les pièces créées, il faut pouvoir les sélectionner. Le plus simple est de cliquer sur la pièce en question, puis de modifier ses paramètres. Pour savoir si le curseur de la souris se trouve dans un polygone, nous avons choisi d'utiliser la méthode du raycast.

Le principe est le suivant : nous traçons une droite passant par deux points (R, et M qui est le pointeur), puis nous comptons le nombre d'intersections entre le segment [RM] et les côtés du polygone. Si ce nombre est impair, le point M est dans le polygone, et si il est pair, le point M se trouve à l'extérieur du polygone.



La mise en place d'un raycast a été une tâche très difficile, puisqu'il a fallu déterminer quand notre segment [RM] et un côté du polygone sont sécants.

3 . Tests

a . Initialisation

Lors du lancement de l'application nous devons créer un étage vide.

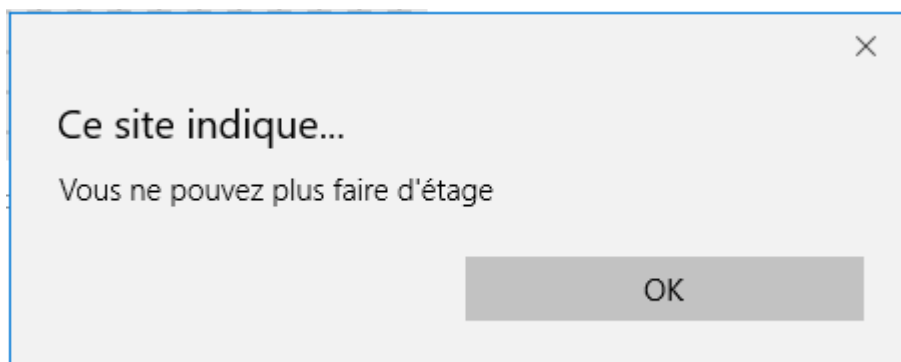
```
étage créé : 0
editorVectorTest.js (222,3)
└─ [Object Array]      [Object {...}]
   editorVectorTest.js (223,3)
   └─ 0                 [object Object] {...}
      length            1
```

Nous voyons que l'étage a bien été créé, et que celui-ci est vide.

b . Création et destruction d'étage

Nous pouvons utiliser le bouton qui crée un étage, et incrémente l'étage sélectionné.

La création d'étages est bloqué à 100, lorsque celui-ci y arrive il fait apparaître une pop-up indiquant l'impossibilité d'en créer un de plus.

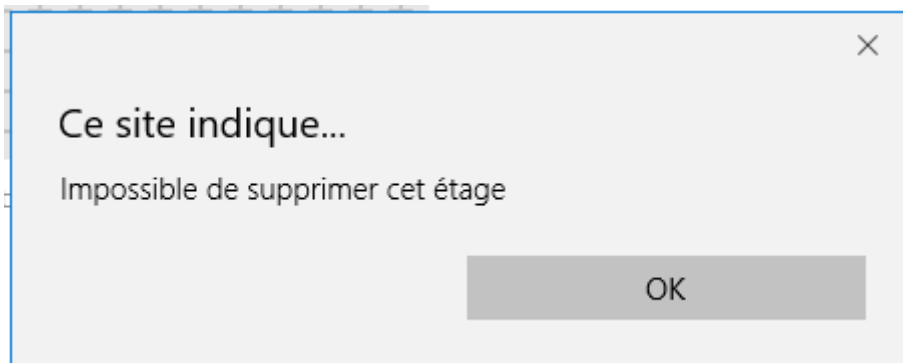


Lors de la suppression d'un étage, il décrémente l'étage sélectionné et supprime l'étage.

```
étage créé : 1
editorVectorTest.js (222,3)
└─ [Object Array]      [Object {...}, Object {...}]
   editorVectorTest.js (223,3)
   étage détruit, étage sélectionné : 0
   editorVectorTest.js (236,3)
   └─ [Object Array]      [Object {...}]
```

Si l'étage qui était sélectionné était l'étage 0, il ne décrémente pas et supprime l'étage.
(voir Code de création d'étage et Code de destruction d'étage)

De plus, si il ne reste qu'un seul étage, il est impossible de le supprimer, et un pop-up avertit l'utilisateur.



c . Navigation dans les étages

Nous pouvons passer d'un étage à l'autre en cliquant sur les boutons, le programme incrémentera ou décrémentera alors l'étage sélectionné, mais ne dépassera jamais le nombre d'étages présent.

```
étage inférieur : 0
editorVectorTest.js (286,3)
impossible d'aller a l'étage inférieur
editorVectorTest.js (280,4)
étage supérieur : 1
editorVectorTest.js (298,3)
étage supérieur : 2
editorVectorTest.js (298,3)
impossible d'aller a l'étage supérieur
```

(voir Code de sélection d'étage)

d . Création des pièces

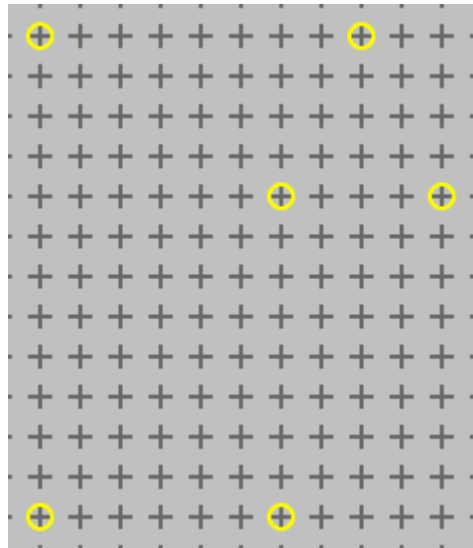
Pour créer une pièce il faut signaler à l'éditeur que nous passons en mode **création de pièce**. Nous modifions l'attribut "m_status" de mainWindow, qui est là pour signaler en quel mode nous sommes, puis nous sommes capable de sélectionner des points sur la grille.

(voir Code de changement d'état pour la création de pièce)

Ici nous voyons que le mode a été changé.

```
changement de l'état de l'éditeur vers le mode création de pièce
editorVectorTest.js (243,3)
```

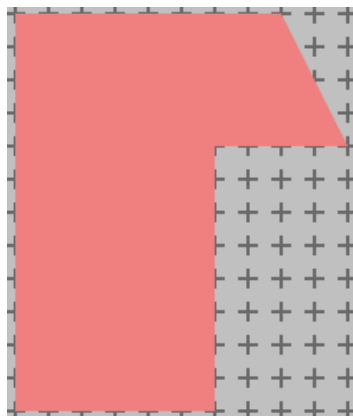
Ci-dessous nous voyons les points sélectionnés avec l'éditeur.



Lors de la validation des points, en faisant un clic droit, l'éditeur nous demande d'entrer le nom de la pièce.

A light grey dialog box with a blue border and a close button (X) in the top right corner. Inside the box, the text "Entrez le nom de la pièce" is displayed above a text input field. The input field contains the word "Chambre" and has a small close button (X) on its right side. Below the input field are two buttons: "OK" and "Annuler".

Une fois le nom saisi, la pièce s'affiche correctement.



Ci-dessous nous voyons l'attribut "m_rooms" de la classe floor, qui témoigne de la création de la pièce.

```

pièce créée
editorVectorTest.js (87,3)
└─ [object Array] [Object {...]]
    └─ editorVectorTest.js (88,3)
        └─ 0 [object Object] {...]
            length 1

```

Ci-dessous nous voyons que 4 pièces ont été créées

```

└─ [object Array] [Object {...], Object {...], Object {...], Object {...]]
    editorVectorTest.js (88,3)
    └─ 0 [object Object] {...]
        1 [object Object] {...]
        2 [object Object] {...]
        3 [object Object] {...]
        length 4

```

De plus l'attribut "m_status" passe en mode **attente**, ce qui permet d'avoir le contrôle sur les autres fonctionnalités de l'application.

```

changement de l'état de l'application en attente
editorVectorTest.js (412,6)

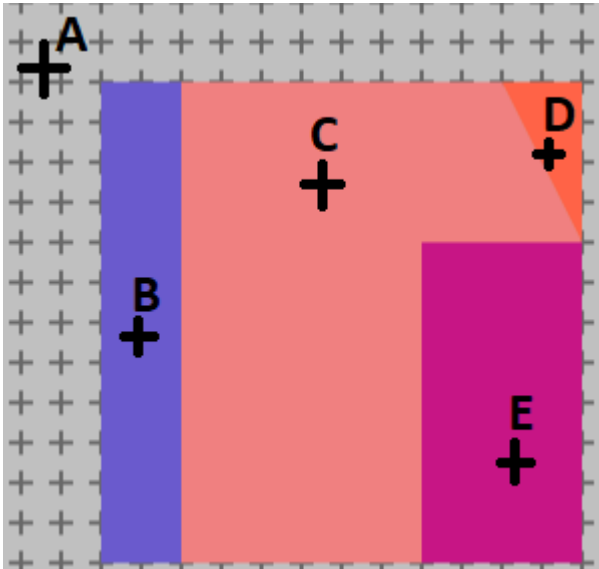
```

Si l'éditeur est en création de pièces, toutes les autres fonctionnalités de l'éditeur sont verrouillées, comme la création d'un étage, d'une pièce, le changement d'étage ou sa suppression, et la modification ou la suppression d'une pièce.

(voir Code d'ajout d'une pièce)

e . Sélection des pièces

Pour sélectionner une pièce il nous faut utiliser notre raycast, pour le tester nous allons pointer différents endroits de l'éditeur contenant plusieurs pièces.



Points sélectionnés	Résultats attendus
A	"vous ne survolez pas de pièce"
B	"corridor : 0"
C	"salle a manger : 1"
D	"chambre : 2"
E	"Salon : 3"

vous ne survolez pas de pièce

`editorVectorTest.js (105,3)`

corridor : 0

`editorVectorTest.js (101,5)`

salle a manger : 1

`editorVectorTest.js (101,5)`

chambre : 2

`editorVectorTest.js (101,5)`

Salon : 3

`editorVectorTest.js (101,5)`

Sur la console nous obtenons ceci, ce qui montre bien que notre raycast fonctionne.

(voir Code du raycast)

Lorsque nous cliquons sur une pièce il faut pouvoir retenir sur laquelle nous avons cliqué, l'indice de la pièce sera donc stocké dans l'attribut "m_selectedRoom", et s'affiche sur l'éditeur.

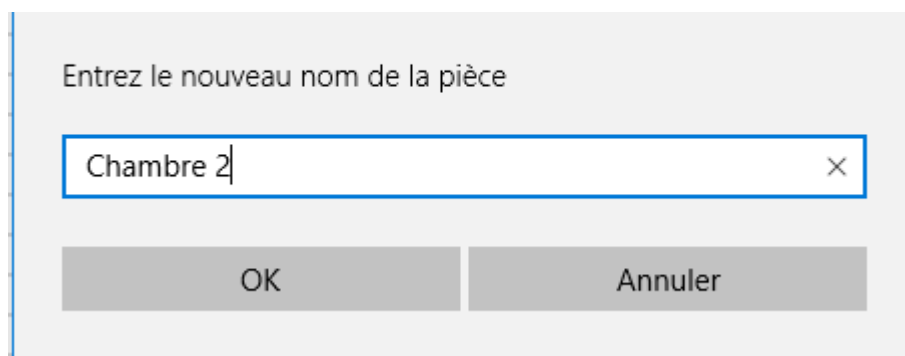
Pièce sélectionnée : corridor

Changer le nom de la pièce

Supprimer la pièce

f . Modification et destruction des pièces

Maintenant que nous pouvons sélectionner notre pièce, nous pouvons changer son nom, en cliquant sur le bouton le programme fait apparaître une pop-up pour que nous puissions faire la modification.



Entrez le nouveau nom de la pièce

Chambre 2

OK Annuler

Après la validation le nom change, et s'affiche.

Pièce sélectionnée : Chambre 2

Changer le nom de la pièce

Supprimer la pièce

Si nous cliquons sur le bouton supprimer, la pièce est détruite et le message « Aucune pièce sélectionnée » s'affiche à la place du nom.

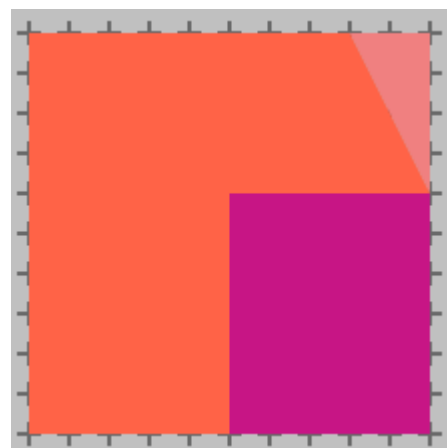
Ici nous voyons qu'il ne reste que trois pièces, que la pièce sélectionnée a été supprimée, et qu'il n'y a plus de pièce sélectionnée.

```
▲ [object Array]      [Object {...}, Object {...},
  editorVectorTest.js (315,3)
  ▶ 0                  [object Object] {...}
  ▶ 1                  [object Object] {...}
  ▶ 2                  [object Object] {...}
  length               3
```

Pièce sélectionnée : Aucune pièce sélectionnée

Changer le nom de la pièce

Supprimer la pièce



g . Sauvegarde du plan

Lorsque nous sauvegardons, le programme génère une sauvegarde au format XML.

```
- <blueprint>
  - <floor>
    - <room>
      <name>pièce 1 étage 0</name>
      - <point>
        <x>9</x>
        <y>9</y>
      </point>
      - <point>
        <x>9</x>
        <y>15</y>
      </point>
      - <point>
        <x>14</x>
        <y>15</y>
      </point>
      - <point>
        <x>14</x>
        <y>9</y>
      </point>
    </room>
  </floor>
  - <floor>
    - <room>
      <name>pièce 1 étage 1</name>
      - <point>
        <x>9</x>
        <y>7</y>
      </point>
      - <point>
        <x>8</x>
        <y>17</y>
      </point>
      - <point>
        <x>12</x>
        <y>18</y>
      </point>
      - <point>
        <x>13</x>
        <y>13</y>
      </point>
      - <point>
        <x>17</x>
        <y>13</y>
      </point>
      - <point>
        <x>17</x>
        <y>7</y>
      </point>
    </room>
  </floor>
</blueprint>
```

Ici nous avons un exemple de sauvegarde avec

- deux étages,
 - à chaque étage une pièce,
 - dans chaque pièce
 - le nom de la pièce
 - les points stockés

(voir Code de sauvegarde XML)

V . Travail restant

À ce stade, il ne reste plus qu'à envoyer le fichier XML à la base de données du serveur, puis à associer les noms des pièces déjà existants de la base de données à l'éditeur.

Annexes

I. Éditeur matriciel

```

this.NewFloor = function() {
    var stage = new Array();
    floors.splice(selectedFloor+1, 0, stage); // nouvelle étage
    for ( var x = 0; x < NB_COL; x++)
        stage.push(new Array());
    for ( var i = 0; i < NB_COL; i++) {
        for ( var j = 0; j < NB_ROW; j++) {
            stage[i].push(STAGE_INIT_VALUE);
        }
    }
    roomTypesName.push(new Map());
    roomTypesName[roomTypesName.length-1].set(0, "Vide");// Gris (Vide)
    roomTypesName[roomTypesName.length-1].set(1, "Salle 1");// Rouge
    roomTypesName[roomTypesName.length-1].set(2, "Salle 2");// Marron
    roomTypesName[roomTypesName.length-1].set(3, "Salle 3");// Jaune
    roomTypesName[roomTypesName.length-1].set(4, "Salle 4");// Vert
    roomTypesName[roomTypesName.length-1].set(5, "Salle 5");// Cyan
    roomTypesName[roomTypesName.length-1].set(6, "Salle 5");// Bleu
    roomTypesName[roomTypesName.length-1].set(7, "Salle 6");// Mauve
    roomTypesName[roomTypesName.length-1].set(8, "Salle 7");// Magenta
    if(floors.length>1){
        selectedFloor++;
        this.DrawFloor();
    }
    console.log("Étage créé !" + floors.length);
};

```

Illustration 3 Code de création d'étage

```

this.DestroyFloor = function(){
    if(floors.length>1)
    {
        floors.splice(selectedFloor,1);
        roomTypesName.splice(selectedFloor,1);
        if(selectedFloor>0) selectedFloor--;
        this.DrawFloor();
        console.log("étage détruit");
    }
    else{
        console.log("impossible de détruire cette étage");
    }
}

```

Illustration 4 Code de destruction d'étage

```
this.GoPrevFloor=function() {
    if(selectedFloor>0)
    {
        selectedFloor--;
        this.DrawFloor();
        console.log("étage précédant");
    }else{
        console.log("impossible d'accéder a l'étage précédant");
    }
}
this.GoNextFloor=function() {
    if(selectedFloor<floors.length-1)
    {
        selectedFloor++;
        this.DrawFloor();
        console.log("étage suivant");
    }else{
        console.log("impossible d'accéder a l'étage suivant");
    }
}
}
```

Illustration 5 Code de navigation entre les étages

```
this.FloorCaseHovering = function() {
    var posx = Math.floor((cursorPosition.x - MATRIX_OFFSET.x)
        / SIZE_SQUARE);
    var posy = Math.floor((cursorPosition.y - MATRIX_OFFSET.y)
        / SIZE_SQUARE);
    if ((posx >= NB_COL) || (posx < 0) || (posy >= NB_ROW) || (posy < 0))
        return {
            x : null,
            y : null
        };
    return {
        x : posx,
        y : posy
    };
};
```

Illustration 6 Code de ciblage de la grille

II . Éditeur vectoriel

```
this.NewFloor = function(){
    if(m_status == m_statusValue.DRAWING_ROOM) return;
    if(m_floors.length>99){
        alert("Vous ne pouvez plus faire d'étage")
        return;
    }
    m_selectedRoom=-1;
    m_floors.splice(++m_selectedFloor, 0, new Floor());
    this.DrawSelectedFloor();
    console.log("étage créé : " + m_selectedFloor);
    console.log(m_floors);
};
```

Illustration 7: Code de création d'étage

```
this.DeleteFloor = function(){
    if(m_status == m_statusValue.DRAWING_ROOM) return;
    if(m_floors.length==1){
        alert("Impossible de supprimer cet étage");
        return;
    }
    m_selectedRoom=-1;
    m_floors.splice(m_selectedFloor, 1);
    if(m_selectedFloor) m_selectedFloor--;
    this.DrawSelectedFloor();
    console.log("étage détruit, étage sélectionné : " + m_selectedFloor);
    console.log(m_floors);
};
```

Illustration 8: Code de destruction d'étage

```

this.GoPrevFloor = function() {
    if(m_status == m_statusValue.DRAWING_ROOM) return;
    if(m_selectedFloor==0)
    {
        console.log("impossible d'aller a l'étage inférieur");
        return;
    }
    m_selectedRoom=-1;
    m_selectedFloor--;
    this.DrawSelectedFloor();
    console.log("étage inférieur : " + m_selectedFloor);
};

this.GoNextFloor = function() {
    if(m_status == m_statusValue.DRAWING_ROOM) return;
    if(m_selectedFloor==m_floors.length-1)
    {
        console.log("impossible d'aller a l'étage supérieur");
        return;
    }
    m_selectedRoom=-1;
    m_selectedFloor++;
    this.DrawSelectedFloor();
    console.log("étage supérieur : " + m_selectedFloor);
};

```

Illustration 9: Code de sélection d'étage

```

this.NewRoom = function() {
    if(m_status == m_statusValue.DRAWING_ROOM) return;
    m_status = m_statusValue.DRAWING_ROOM;
    console.log("changement de l'état de l'éditeur vers le mode création de pièce");
}

```

Illustration 10: Code de changement d'état pour la création de pièce

```

this.AddRoom = function() {
    if(m_tempRoom.GetPolygone().length<=2) {
        alert("La Pièce doit contenir plus de points !");
        return false;
    }
    m_tempRoom.SetName(prompt('Entrez le nom de la pièce'));
    m_rooms.push(m_tempRoom);
    m_tempRoom = new Room();
    console.log("pièce créée");
    console.log(m_rooms);
    return true;
};

```

Illustration 11: Code d'ajout d'une pièce

```

this.Raycast = function(point) {
    var x = point.x, y = point.y;

    var inside = false;

    for (var i = 0, j = m_polygone.length - 1; i < m_polygone.length; j = i++) {
        var xi = m_polygone[i].x, yi = m_polygone[i].y;
        var xj = m_polygone[j].x, yj = m_polygone[j].y;

        var intersect = ((yi > y) != (yj > y))
            && (point.x < (xj - xi) * (point.y - yi) / (yj - yi) + xi);
        if (intersect) inside = !inside;
    }
    return inside;
};

```

Illustration 12: Code du raycast

```

this.Save = function() {
    var xmlString = "<blueprint>"; //début de la balise blueprint

    for(var iFloor=0; iFloor<m_floors.length; iFloor++){
        xmlString += "<floor>"; //début de la balise floor

        var tempRooms = m_floors[iFloor].GetRooms();
        for(var iRoom=0; iRoom<tempRooms.length; iRoom++){
            var tempRoom = tempRooms[iRoom];

            xmlString += "<room>" + //début balise room
                "<name>" + tempRoom.GetName() + "</name>"; //récupération du nom
            for(var j=0; j < tempRoom.GetPolygone().length; j++){

                xmlString += "<point>" + //début balise point
                    "<x>" + tempRoom.GetPolygone()[j].x + "</x>" + //balise x
                    "<y>" + tempRoom.GetPolygone()[j].y + "</y>" + //balise y
                    "</point>"; //fin balise point
            }
            xmlString += "</room>"; //fin balise room
        }
        xmlString += "</floor>"; //fin balise floor
    }
    xmlString += "</blueprint>"; //fin de la balise blueprint
    console.log(xmlString);
};

```

Illustration 13: Code de sauvegarde XML