Manuel d'utilisation d'*algoGGB*

Présentation d'*algoGGB*

GeoGebra permet de créer (puis d'explorer interactivement) des figures mathématiques fort intéressantes, et ce de plusieurs façons :

- via une combinaison de commandes gestuelles et textuelles
  (textuelles : en utilisant son champ de saisie)
- via la création et l'utilisation d'outils fabriqués sur mesure
- via un langage de script, simple mais limité
- via un langage de programmation complet, puissant mais d'utilisation plutôt complexe (*JavaScript*).

Les ressources mises à notre disposition par *GeoGebra* sont tellement riches et variées qu'on peut se demander s'il sera jamais nécessaire d'avoir recours à la programmation *JavaScript* dans *GeoGebra*. Pourtant, il y a parfois des cas où ce type de programmation offre des avantages certains : par exemple

1. *Exploration d'autres types de géométries*
   GeoGebra est conçu pour l'étude et l'exploration de la géométrie euclidienne, que ce soit sous sa forme synthétique que sous sa présentation analytique. Mais on peut chercher à travailler dans le cadre d'autres géométries, comme celle de la **tortue**, popularisée par le mouvement Logo, et reprise par la suite dans d'autres contextes. Et, par sa nature même, la **géométrie de la tortue** nécessite une programmation dans un langage complet : pourquoi ne pas utiliser *GeoGebra* ?

2. *Exploration de figures tellement complexes qu'on doit les décrire par un programme*
   Certaines figures sont trop complexes pour être décrites manuellement, en énumérant les divers éléments qui les composent : les **fractals**, par exemple. En effet, plusieurs types de fractals sont de nature algorithmique, et on doit donc utiliser la programmation si on veut en obtenir des représentations. Ici encore : pourquoi ne pas utiliser *GeoGebra* ?

Si ces deux types d'utilisations font appel à une programmation, on peut se demander si *GeoGebra* est un contexte intéressant pour la réaliser. Après tout, il existe des environnements de programmation plus conviviaux que *GeoGebra*, et des langages plus efficaces et plus rapides que *JavaScript*, pour réaliser de telles activités. Mais il faut aussi voir que *GeoGebra* offre des avantages non négligeables :

1. *GeoGebra* offre directement des facilités de manipulation des figures, ne requérant aucune programmation supplémentaire :
   - on peut toujours déplacer, agrandir et réduire les figures
   - on peut souvent même modifier celles-ci
   - on peut utiliser directement boutons et glissières

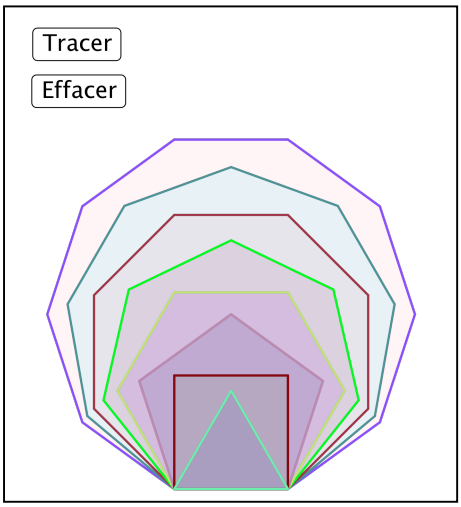
2. *GeoGebra* nous permet de placer aisément nos figures sur le web tout en préservant leur caractère interactif.

3. *GeoGebra* nous permet d'exporter nos figures sous différents formats, utilisables dans d'autres logiciels, notamment
   - le format vectoriel **SVG**, utilisable dans *Libre Office* et dans tous les navigateurs web modernes
   - le format matriciel **PNG**, avec divers choix de résolution, utilisable dans *Libre Office*, *Microsoft Office*, et dans tous les navigateurs web modernes.

Pour ces raisons, et conscient de la complexité de la programmation générale dans *GeoGebra*, j'ai écrit une bibliothèque de procédures permettant de simplifier l'utilisation de *JavaScript* dans *GeoGebra* : à court d'idées pour le nom, je l'ai nommée ***algoGGB***.

Un premier exemple, pas à pas

Je vais maintenant décrire comment utiliser ***algoGGB*** pour réaliser la figure suivante, en utilisant la géométrie de la tortue :

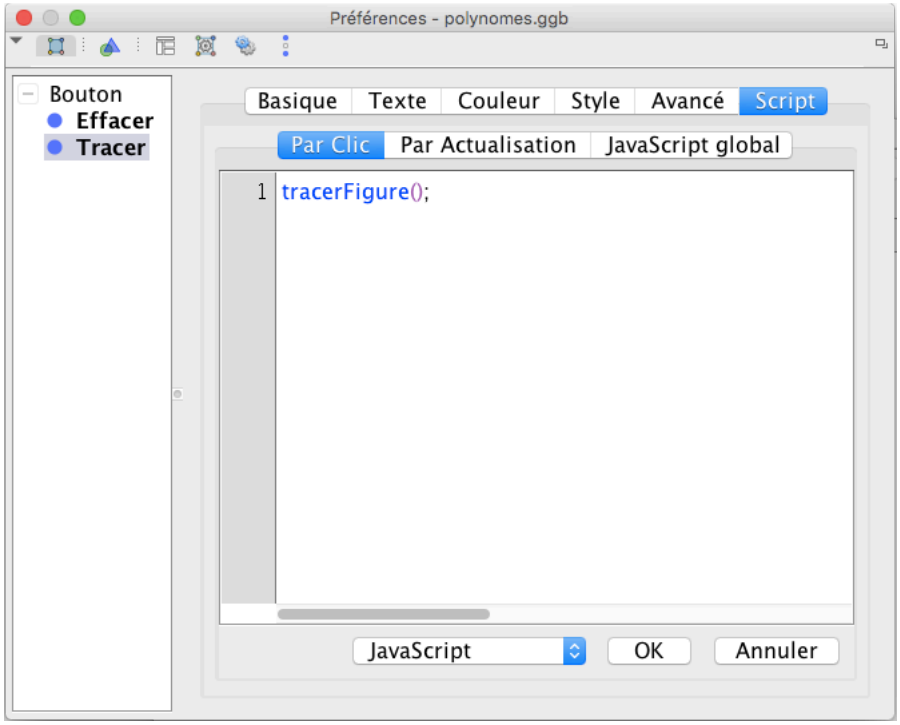


Mais tout d'abord, un mot d'avertissement : mon but, dans ce manuel, est exclusivement de décrire comment utiliser ***algoGGB***. Je vais donc supposer que le lecteur a des connaissances raisonnables de *GeoGebra*, du langage *JavaScript* et des domaines mathématiques (**géométrie de la tortue**, **fractals**) auxquels je vais faire référence.

Une remarque avant de commencer : ce premier exemple, même après une amélioration ultérieure, reste très facile à réaliser sans avoir recours à la programmation. Il a été choisi pour sa simplicité et pour illustrer les rudiments de la géométrie de la tortue. Par contre, les exemples subséquents ne seront pas facile à réaliser dans GeoGebra sans utiliser algoGGB...

L'environnement ***algoGGB*** se présente sous la forme d'une figure *GeoGebra* (appelée, vous l'avez deviné, ***algoGGB.ggb***) comportant deux boutons (**Tracer** et **Effacer**) et dotée d'une bibliothèque de procédures *JavaScript*. La première étape de notre démarche consistera donc à dupliquer cette figure *GeoGebra*, et à donner à sa copie le nom de ***polygones.ggb***.

Examinons maintenant les propriétés du bouton **Tracer**, que l'on atteint par un clic droit sur ledit bouton, suivi du choix de l'item « Propriétés… » dans le menu local qui apparaît alors. Un clic sur l'onglet « Script », puis sur l'onglet « Par Clic » nous conduit à la situation décrite dans la figure ci-dessous.



On peut interpréter ceci comme suit : un clic sur le bouton **Tracer** exécutera la procédure *JavaScript* **tracerFigure**. De même, un clic sur **Effacer** dans la colonne de gauche nous apprendra qu'un clic sur ce bouton exécutera la procédure *JavaScript* **initialiser**.

Un clic sur l'onglet « JavaScript global » fera apparaître la bibliothèque de procédures ***algoGGB***. Mais, ne vous laissez pas impressionner par la quantité d'informations que vous y trouverez. Rendez-vous plutôt vers la fin du texte, pour retrouver la section « Procédures utilisateur » suivante :

```
// ********************************************************************************
// ****************** Procédures utilisateur (ci-dessous) ************************
// ********************************************************************************

var nbInitialObjets = 2; // Nombre d'objets au départ, à ne pas effacer
var valeursAsurveiller = [];  // Liste des noms des valeurs à surveiller
var pointsAsurveiller = []; // Liste des noms des points à surveiller

function tracerFigure(){
        initialiser() ;

}
```

À moins de devenir un expert d'***algoGGB***, vous vous limiterez à travailler dans la section « Procédures utilisateur ». Pour l'instant, on peut faire trois remarques :

- La variable **nbInitialObjets** vaut deux, ce qui correspond à la présence de nos deux boutons **Tracer** et **Effacer** qu'on veut préserver quand on efface. Si on désirer ajouter d'autres éléments « permanents », il faudra mettre cette variable à jour.
- Pour l'instant, la procédure **tracerFigure** ne comporte que l'instruction **initialiser**. C'est ce qui explique qu'il ne se passe rien de visible quand on clique sur le bouton **Tracer**. Ce sera à nous d'ajouter les instructions pour tracer notre figure.
- La procédure **initialiser** n'est pas décrite dans la section « Procédures utilisateur » : c'est donc ***algoGGB*** qui gèrera cela pour nous, en se basant notamment sur la variable **nbInitialObjets**.

Voyons maintenant comment faire tracer notre figure. Commençons par faire tracer un seul polygone, en se servant de nos connaissances de la géométrie de la tortue et de *JavaScript* :

```
function polygone(nbCotes, longueurCote) {
        for (var k=0 ; k < nbCotes ; k=k+1) {
                avance(longueurCote) ;
                gauche(360/nbCotes) ;
        }
}
```

Si l'on ajoute la procédure **polygone** dans la section « Procédures utilisateur », et si on ajoute une instruction comme « polygone(5,10) ; » dans la fonction **tracerFigure**, on risque d'être déçu par le résultat obtenu : figure trop grande ou trop petite, mal placée et mal orientée, etc.

On peut améliorer les choses en remarquant que la tortue utilise le système d'axes de GeoGebra : elle se trouve initialement à l'origine et « se dirige » vers le haut. Vous pouvez mieux voir la situation en rendant (temporairement ?) les axes visibles…

Autre remarque : n'effacez pas la commande **initialiser**, et laissez-la comme première instruction de la fonction **tracerFigure** : ceci vous assurera d'effacer les tracés précédents et de réinitialiser les paramètres d'***algoGGB*** (dont l'état de la tortue).

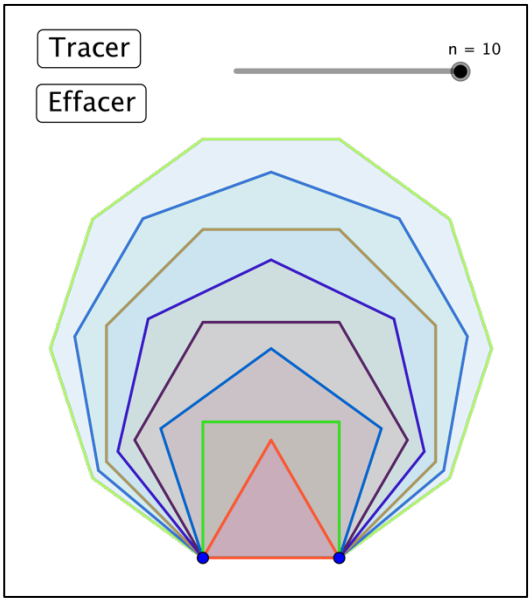
Dans mon cas, avec mon choix de fenêtre et de système d'axes, et après avoir ajouté une boucle pour tracer plusieurs polygones, la fonction **tracerFigure**, pourrait ressembler à ceci :

```
function tracerFigure() {
        initialiser() ;
        droite(90) ; sautePos( -1,-3) ;
        for (var k=10 ; k > 2; k=k-1) {
                couleurCrayon(hasard(0,255), hasard(0,255), hasard(0,255));
                couleurRemplissage(hasard(0,255), hasard(0,255), hasard(0,255), 0.1);
                debutRemplir();
                polygone(k,2) ;
                finRemplir() ;
        }

}
```

Mentionnons simplement que l'instruction tortue **sautePos** déplace la tortue aux coordonnées spécifiées, sans que son orientation soit modifiée, et sans que son crayon laisse une trace. Quant aux instructions **debutRemplir** et **finRemplir**, elles commandent le remplissage d'un polygone décrit à l'aide d'instructions **avance**, **recule**, **segment** ou **fixePos**. Et on utilise les instructions **couleurCrayon** et **couleurRemplissage** pour spécifier les composantes rouge, verte et bleue (via un nombre entre 0 et 255) des couleurs à utiliser : dans le cas présent, ces composantes sont choisies au hasard. Pour **couleur Remplissage**, un quatrième paramètre spécifie le degré de transparence : il peut varier de 0 (complètement transparent) à 1 (complètement opaque).

Amélioration de notre premier exemple

Notre premier exemple trace bien la figure désirée, mais le tout n'est pas très dynamique. On va améliorer la situation par l'ajout de 3 éléments initiaux supplémentaires : deux points (qui détermineront le côté commun à tous les polygones) et une glissière (qui déterminera le nombre de polygones à tracer).



Mais avant d'ajouter ces 3 éléments, il est bien important de s'assurer qu'on a bien effacé tous les tracés précédents. Voici comment procéder :

- On clique sur le bouton **Effacer**
- On ajoute deux points, nommés 'P' et 'Q'
- On ajoute une glissière, nommée 'n', allant de 3 à 10 par sauts de 1
- On modifie immédiatement la valeur de la variable **nbInitialObjets** pour la porter à 5 (les deux boutons initiaux, plus les 3 éléments qu'on vient d'ajouter).

Il nous faut ensuite tenir compte de ces trois nouveaux éléments dans le tracé de notre figure : ceci donne une nouvelle version de **tracerFigure**, où les commandes ajoutées ou modifiées sont en rouge.

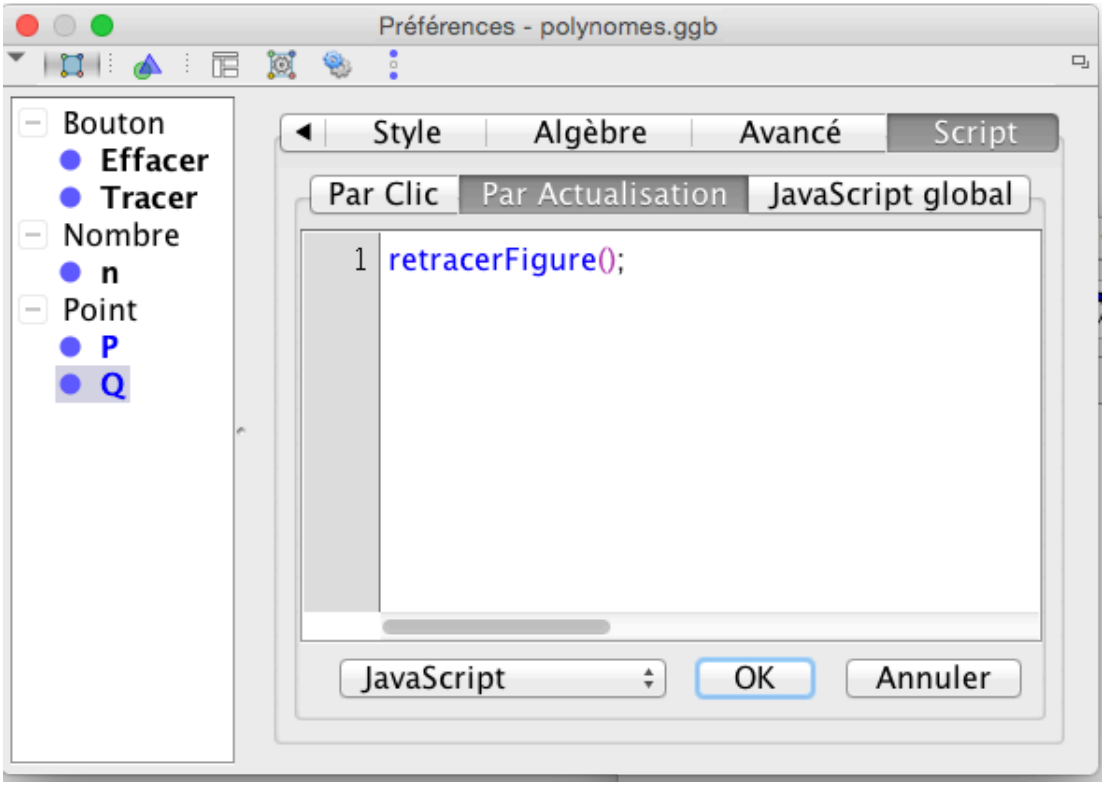
```
function tracerFigure() {
        initialiser() ;
        var n = valeur('n') ;
        var x1 = coordX('P'), y1 = coordY('P') ;
        var x2 = coordX('Q'), y2 = coordY('Q') ;
        var d = racine((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)) ;
        droite(90) ; sautePos( x1,y1) ; fixeCap(vers(x2,y2)) ;
        for (var k=n ; k > 2; k=k-1) {
                couleurCrayon(hasard(0,255), hasard(0,255), hasard(0,255));
                couleurRemplissage(hasard(0,255), hasard(0,255), hasard(0,255), 0.1);
                debutRemplir();
                polygone(k,d) ;
                finRemplir() ;
        }

}
```

Examinons de plus près la première commande ajoutée : **n =valeur('n')**. On peut interpréter ceci comme suit : on stocke dans une variable *JavaScript* nommée **n** la valeur d'un objet *GeoGebra*, soit une glissière nommée *n*. Notez qu'on aurait pu utiliser une autre variable *JavaScript*, par exemple **w =valeur('n')** : on stocke dans une variable *JavaScript* nommée **w** la valeur de la glissière *GeoGebra* nommée *n*. En général, ce sera ainsi que notre programme ***algoGGB*** communiquera avec **GeoGebra** : via le nom de ses objets.

De la même façon, on récupère les coordonnées des deux points *GeoGebra P* et *Q*, et on s'en sert pour calculer la distance entre les points et pour fixer la position de départ (le point *P*). L'orientation de départ est déterminée, non plus en tournant à droite de 90°, mais en se tournant vers le second point (le point *Q*).

Les boutons **Tracer** et **Effacer** tiennent maintenant compte des changements que nous avons apportés, mais il nous reste à informer ***algoGGB*** que nous voulons rendre interactifs la glissière et les points. Ceci se fait en deux temps :

- On doit en informer les variables **valeursAsurveiller** et **pointsAsurveiller**. Dans notre cas, ceci prend la forme
  var valeursAsurveiller = ['n'] ;
  var pointsAsurveiller = ['P', 'Q'];
- Dans le sous-onglet « Par Actualisation » de l'onglet « Script » de chacun de nos trois objets, il faut faire un appel à la procédure **retracerFigure**, comme suit

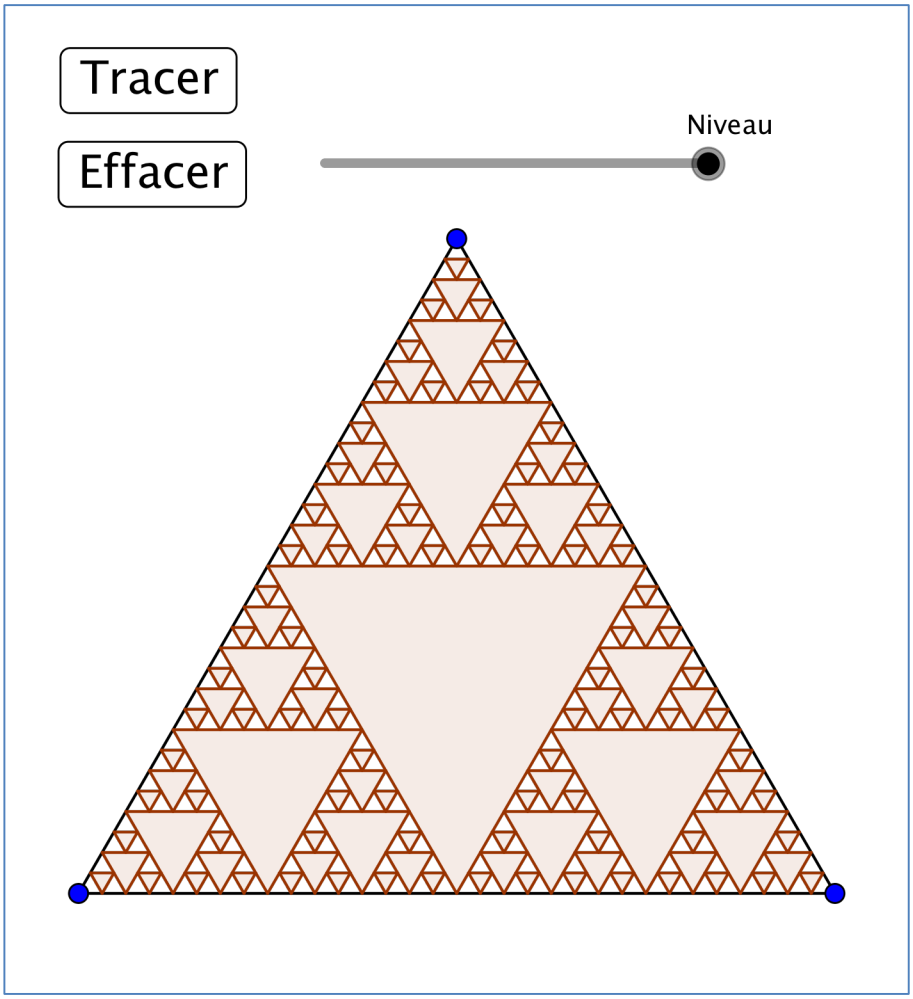


Le résultat : une figure qui « suit » interactivement le déplacement de vos points, ainsi que les changements de votre glissière.

Une remarque pour terminer cet exemple. Au moment où j'écris ces lignes, l'éditeur *JavaScript* de *GeoGebra* est plutôt primitif : entre autres faiblesses, il ne possède même pas d'outil de recherche-remplacement. Donc, si vos programmes deviennent un peu complexes, il vaut peut-être mieux les écrire dans un éditeur spécialisé pour la programmation, comme *Brackets* (voir http://brackets.io/). De plus, les messages émis en cas d'erreur par *GeoGebra* ne sont pas toujours précis. Espérons que le tout va s'améliorer avec le temps.

Un second exemple

Je vais maintenant décrire comment utiliser ***algoGGB*** pour réaliser la figure suivante, qui est une représentation d'un fractal, le célèbre triangle de Sierpinski :



Mais, alors que dans l'exemple précédent le programme *JavaScript* gérait lui-même tous les tracés de la figure transformée, nous laisserons ce soin à *GeoGebra* dans la seconde figure. Ceci implique qu'il nous faudra passer d'une description numérique de la figure (exemple : on trace un segment de longueur et d'inclinaison données) à une description plus conceptuelle (exemple : on crée dans *GeoGebra* un segment reliant deux points connus de *GeoGebra*). Ce qui nécessitera qu'il faudra non seulement demander des informations à *GeoGebra* (comme dans l'exemple précédent), mais aussi et surtout demander à *GeoGebra* de construire de nouveaux objets. Ceci se fera à l'aide de la fonction **commandeG**, dont le fonctionnement est un peu subtil : prière de rester concentré !

Commençons à apprivoiser **commandeG** dans un cas simple : si on peut utiliser une commande, par exemple « s=Segment[P,Q] », dans le champ de saisie de *GeoGebra*, alors on peut utiliser **commandeG('s=Segment[P,Q]')** dans nos programmes ***algoGGB***. Mais, dans

un tel cas, il faut avouer qu'un programme ***algoGGB*** n'a aucun avantage sur l'action directe. D'autant plus qu'il nous faudra utiliser les **versions anglaises des commandes** dans tous nos programmes *JavaScript* de *GeoGebra*.

Mais si nous avons à tracer une multitude de segments, comme c'est le cas pour le triangle de Sierpinski, nous serons amenés à gérer automatiquement les noms des divers segments, notamment en plaçant ces noms dans des variables *JavaScript*.

Ainsi, si le nom du premier point est placé dans une variable *point1*, que le nom du second point se trouve dans la variable *point2*, et que le nom du segment qu'on veut créer se trouve dans une troisième variable *seg*, il serait incorrect d'utiliser la commande

commandeG('seg=Segment[point1,point2]')

puisque, par exemple, le nom du premier point n'est pas *point1* : il est contenu dans la variable *point1* (par exemple, il pourrait être *P237*). On peut résoudre ce problème en faisant appel aux capacités de manipulation des chaînes de caractères dans *JavaScript*. On obtient alors la commande

commandeG(seg + '=Segment[' + point1 + ', ' + point2 + ']')

qui est correcte mais qui semble aussi inutilement compliquée. D'autant plus qu'il s'agit ici d'une commandes très simple : imaginez pour les commandes plus complexes !

algoGGB propose un mécanisme pour simplifier ce type de problème : dans la situation ci-dessus, ceci prend la forme

commandeG('@1=Segment[@2,@3]', seg, point1, point2)

Voici comment interpréter cette nouvelle forme de **commandeG** : avant d'exécuter la commande **Segment**, ***algoGGB*** remplacera

- l'expression spéciale **@1** par la valeur de la variable *JavaScript* **seg**
- l'expression spéciale **@2** par la valeur de la variable *JavaScript* **point1**
- l'expression spéciale **@3** par la valeur de la variable *JavaScript* **point2**.

Voyons comment appliquer tout ceci à la construction du triangle de Sierpinski. Comme d'habitude, faisons une copie de ***algoGGB.ggb***, que nous nommerons **sierpinski.ggb**. Ce fichier comporte déjà deux objets initiaux (les boutons **Tracer** et **Effacer**), auxquels nous ajouterons une glissière (**Niveau** – allant de 1 à 5 par sauts de 1) et trois points (**A**, **B**, **C** – dont on cachera les étiquettes), ce qui portera **nbInitialObjets** à 6 .

Dans tracerFigure, nous commencerons par tracer les segments reliant nos trois points (tout en cachant leurs étiquettes), puis nous ferons un appel à la procédure récursive sierpinski : ceci pourra ressembler au fragment suivant.

```
function tracerFigure() {
        initialiser() ;
        var n = valeur('Niveau') ;
        commandeG( 'sAB=Segment[A,B] ') ; cacheEtiquette('sAB') ;
        commandeG( 'sAC=Segment[A,C] ') ; cacheEtiquette('sAC') ;
        commandeG( 'sBC=Segment[B,C] ') ; cacheEtiquette('sBC') ;
        sierpinski(n, 'A', 'B', 'C') ;
}
```

Veuillez noter que ce programme ***algoGGB*** ne manipule pas directement les objets *GeoGebra* : il travaille plutôt avec leurs ***noms***. C'est ainsi que ce premier appel à la fonction sierpinski transmet comme paramètres trois chaînes de caractères, soient les noms de nos trois points initiaux : 'A', 'B' et 'C'.

algoGGB propose en outre une façon aisée d'engendrer de nouveaux noms. Par exemple, l'appel **nouveau('Pt')** va retourner un nouveau nom constitué de la chaîne *Pt* suivie d'un nombre, par exemple *78*, choisi de telle sorte que le nom *Pt78* n'existe pas précédemment dans *GeoGebra*. Voyons maintenant à quoi va ressembler notre fonction **sierpinski**.

```
function sierpinski(niveau, point1, point2, point3) {
        if (niveau == 0) {return ;}
        var point12, point13, point23 ;
        point12=nouveau('P') ; point13=nouveau('P') ; point23=nouveau('P') ;
        commandeG('@1=Midpoint[@2,@3]',point12,point1,point2);
        commandeG('@1=Midpoint[@2,@3]',point13,point1,point3);
        commandeG('@1=Midpoint[@2,@3]',point23,point2,point3);
        cacheObjet(point12); cacheObjet(point13); cacheObjet(point23);
        commandeG('@1=Polygon[@2,@3,@4]',
                        nouveau('tri'), point12, point13, point23);
        sierpinski(niveau-1, point1, point12, point13) ;
        sierpinski(niveau-1, point2, point12, point23) ;
        sierpinski(niveau-1, point3, point13, point23) ;
}
```

On remarque tout d'abord l'emploi des versions anglaises des commandes : **Midpoint** et **Polygon**. La stratégie utilisée pour tracer le triangle de Sierpinski demeure cependant très simple :

- on calcule les milieux des trois côtés du triangle, et on trace le triangle (un polygone) déterminé par ces trois points [Note : il faut d'abord trouver des noms pour les trois points, puis les cacher après leur création.]
- on applique récursivement la même procédure aux trois autres petits triangles.

Finalement, on peut ajouter une dernière touche d'interactivité

- on fait la modification suivante : **var valeursAsurveiller = ['Niveau'];**
- dans le script *Par Actualisation* de la glissière **Niveau**, on place **retracerFigure() ;**

Après ce deuxième exemple, nous sommes prêts à examiner plus systématiquement les ressources mises à notre disposition par algoGGB : ce sera l'objet de la section « Référence ***algoGGB*** ».

Référence *algoGGB*

Géométrie de la tortue

Commande	Description
initialiser()	Pas une commande tortue, à proprement parler, mais place la tortue à l'origine, pointant vers l'axe des y positifs. Doit normalement se retrouver au début de la fonction tracerFigure(). Efface tous les objets de GeoGebra sauf les **nbInitialObjets** premiers.
avance(distance)	Fait avancer la tortue d'une certaine distance. Les unités de longueur sont celles de GeoGebra. Pour reculer, utiliser une « distance » négative. Retourne le nom du segment créé (si le crayon est baissé).
droite(angle)	Fait pivoter la tortue vers la droite d'un certain nombre de degrés. Pour pivoter à gauche, utiliser un angle négatif.
posX()	Retourne la coordonnée en X de la position de la tortue
posY()	Retourne la coordonnée en Y de la position de la tortue
fixePos(x,y)	Amène la tortue en position (x,y). Laisse une trace si le crayon est baissé. Retourne le nom du segment créé (si le crayon est baissé).
sautePos(x,y)	Amène la tortue en position (x,y). Ne laisse pas de trace,. même si le crayon est baissé.
cap()	Retourne le cap de la tortue (en degrés)
fixeCap(angle)	Tourne la tortue dans la direction indiquée (en degrés).
vers(x,y)	Donne le cap qu'il faudrait avoir pour aller de la position actuelle de la tortue vers le point (x,y).
baisseCrayon()	Une tortue avec crayon baissé laisse une trace en se déplaçant.
leveCrayon()	Une tortue avec crayon levé ne laisse pas de trace en se déplaçant.
fixeTailleCrayon(taille)	Fixe l'épaisseur du trait.
couleurCrayon(r,v,b)	Les paramètres rouge, vert et bleu varient entre 0 et 255.
couleurRemplissage(r,v,b,a)	Les paramètres rouge, vert et bleu varient entre 0 et 255. Le paramètre a (variant entre 0 et 1) gère le degré de transparence.
debutRemplir()	Après cette commande, toutes les commandes traçant des segments (avance, recule, fixePos, segment) contribueront à définir un polygone.
finRemplir()	Signale la fin de la définition du polygone en cours de définition. Celui-ci est alors tracé, en utilisant la couleur du crayon et de remplissage.

Géométrie analytique

Commande	Description
segment(x1,y1,x2,y2)	Trace le segment reliant les points (x1,y1) et (x2,y2). Retourne le nom du segment ainsi créé (si crayon baissé).
cercle(x,y,r)	Trace le cercle de centre (x,y) et de rayon r. Retourne le nom du cercle ainsi créé.
disque(x,y,r)	Trace le disque de centre (x,y) et de rayon r. Retourne le nom du disque ainsi créé.
arc(x,y,r,a1,a2)	Trace un arc sur le cercle de centre (x,y) et de rayon r. L'arc va de l'angle a1 vers l'angle a2, dans la direction positive. Retourne le nom de l'arc ainsi créé.
ellipse(x,y,a,b)	Trace l'ellipse centrée en (x,y) et de demi-axes a et b. Retourne le nom de l'ellipse ainsi créée.
ellipseRemplie(x,y,a,b)	Trace l'ellipse remplie centrée en (x,y) et de demi-axes a et b. . Retourne le nom de l'ellipse ainsi créée.
ecris(x,y,ligne1, ligne2, …)	Écris un texte, pouvant comporter plusieurs ligne, à partir de la position (x,y). Le texte doit comporter au moins une ligne.

Communication avec GeoGebra

Commande	Description
commandeG(modele,param1,param2, …)	Le "modèle" est une commande GeoGebra (en anglais) comportant des expressions @1, @2, … Avant d'être exécuté, ces expressions seront remplacées par les valeurs de param1, param2, …
valeur(nom)	Valeur d'un objet GeoGebra identifié par son nom.
coordX(nom)	Coordonnée en X d'un point GeoGebra identifié par son nom.
coordY(nom)	Coordonnée en Y d'un point GeoGebra identifié par son nom.
fixeValeur(nom,valeur)	Fixe la valeur d'un objet GeoGebra identifié par son nom.
fixeCoords(nom,x,y)	Fixe les coordonnées d'un objet GeoGebra identifié par son nom.
cacheObjet(nom)	Cache un objet GeoGebra identifié par son nom.
cacheEtiquette(nom)	Cache l'étiquette d'un objet GeoGebra identifié par son nom.
fixeObjet(nom)	Rend fixe un objet GeoGebra identifié par son nom.
montreObjet(nom)	Montre un objet GeoGebra identifié par son nom.
montreEtiquette(nom)	Montre l'étiquette d'un objet GeoGebra identifié par son nom.
libereObjet(nom)	Rend mobile un objet GeoGebra identifié par son nom.
colorier(nom,R,V,B)	Fixe la couleur d'un objet GeoGebra identifié par son nom.
remplir(nom,R,V,B,A)	Fixe le remplissage d'un objet GeoGebra identifié par son nom.
fixeEpaisseur(nom,taille)	Fixe l'épaisseur d'un objet GeoGebra identifié par son nom.
obtenirNomsDerniersObjets(n)	Retoure une liste contenant les noms des *n* derniers objets créés (du plus ancien au plus récent, soit le dernier créé). Utile quand une commande (exemples : **Polygon** ou un outil défini par l'utilisateur) créé plusieurs objets dont on veut modifier les propriétés.

Autres commandes

Commande	Description
tracerFigure()	**Doit être définie par l'utilisateur.**
retracerFigure()	Retrace la figure si une des valeurs de la liste **valeursAsurveiller** ou un des points de la liste **pointsAsurveiller** a été modifié.
nouveau(debut)	Engendre un nouveau nom commençant par la valeur de l'expression *debut* et suivi d'un nombre. Exemple : nouveau('poly') peut produire le nom 'poly137'. Noter que l'objet nommé n'est pas créé par cette commande.
alerte(ligne1, ligne2, …)	Affiche une fenêtre contenant le message décrit dans les diverses lignes. Le message doit comporter au moins une ligne.
demande(ligne1, ligne2, … , parDéfaut)	Affiche une fenêtre contenant le message décrit dans les diverses lignes. Le message doit comporter au moins une ligne. Affiche aussi une zone de texte, comportant initialement la chaîne **parDéfaut**, mais que l'utilisateur peut modifier.
degres(angle)	Convertis en degrés un angle exprimé en radians.
radians(angle)	Convertis en radians un angle exprimé en degrés.
hasard(min, max)	Génère un nombre entier choisi au hasard entre min et max.

Abbréviations

Commande	Description
av(distance)	avance(distance)
recule(distance)	avance(- distance)
re(distance)	recule(distance)
dr(angle)	droite(angle)
gauche(angle)	droite(- angle)
ga(angle)	gauche(angle)
lc()	leveCrayon()
bc()	baisseCrayon()
sin(angle)	Math.sin(angle)
cos(angle)	Math.cos(angle)
tan(angle)	Math.tan(angle)
sinD(angle)	sin(radians(angle))
cosD(angle)	cos(radians(angle))
tanD(angle)	tan(radians(angle))
asin(x)	Math.asin(x)
acos(x)	Math.acos(x)
atan(x)	Math.atan(x)
abs(x)	Math.abs(x)
racine(x)	Math.sqrt(x)