

28 janvier 2018

Auteur : Lovis Thomas,

Professeur : Tièche François

Génération d’animations

Rapport Animaker

Table des matières

[1. Introduction 2](#_Toc504948157)

[1.1. Aperçu de code 2](#_Toc504948158)

[2. Architecture 4](#_Toc504948159)

[2.1. Analyse lexicale 4](#_Toc504948160)

[2.1.1. Résultats 5](#_Toc504948161)

[2.2. Analyse syntaxique 5](#_Toc504948162)

[2.2.1. Configurations 5](#_Toc504948163)

[Objets 5](#_Toc504948164)

[2.2.2. Les mouvements 6](#_Toc504948165)

[2.2.3. Résultats 6](#_Toc504948166)

[2.3. Interprétation et création du fichier 7](#_Toc504948167)

[3. Fonctionnalités / Prise en main 8](#_Toc504948168)

[3.1. Déclarations / Assignations 8](#_Toc504948169)

[3.1.1. Variables de nombre 8](#_Toc504948170)

[3.1.2. Variables objet 8](#_Toc504948171)

[3.2. Affichages 8](#_Toc504948172)

[3.2.1. Console 8](#_Toc504948173)

[3.3. Configuration 8](#_Toc504948174)

[3.3.1. Taille de la fenêtre 8](#_Toc504948175)

[3.3.2. Valeurs de tick 8](#_Toc504948176)

[3.4. Animations 9](#_Toc504948177)

[3.4.1. Mouvement 9](#_Toc504948178)

[3.4.2. Translations 9](#_Toc504948179)

[3.4.3. Rotations 9](#_Toc504948180)

[3.5. Boucles 9](#_Toc504948181)

[4. Problèmes rencontrés 10](#_Toc504948182)

[4.1. Erreurs de parsing 10](#_Toc504948183)

[4.2. Problèmes d’import 10](#_Toc504948184)

[4.3. Rotation 10](#_Toc504948185)

[5. Conclusion 10](#_Toc504948186)

# Introduction

Dans le cadre du cours « Compilateur », nous réalisons un programme qui simplifie la génération d’animation en utilisant la bibliothèque PLY de python.

Notre compilateur permet de générer des animations de formes basiques 2D, c’est-à-dire un carré/rectangle, un cercle et un triangle. Les animations possibles sont : la translation et la rotation. Le compilateur génère du code python utilisant la bibliothèque « pygame ».

## Aperçu de code

L’idée de base était de partir d’un code tel que celui exposé :

1. screen 800 600;
2. tick 80;
3. ball0 = Ball 750, 550, 20, [0, 255, 255];
4. rect0 = Rectangle 10, 400, 100, 25;
5. t = 10;
6. while(t){
7. print t;
8. translate ball0 -10 -10;
9. rotate rect0 1.5707;
10. t = t - 1;
11. };

*Code 1. Exemple de code*

Ce programme définit une fenêtre d’affichage pour l’animation 2D de 800 par 600 pixels et une fréquence de tick à 80 (sleep(80)).

Ensuite il initialise 2 objets (ball0 et rect0) et une variable t. L’animation commence dans la while avec l’affichage de la variable t en console, la translation de la balle à l’écran et la rotation du rectangle de PI/2.

Finalement la variable t est décrémentée afin d’avoir le comportement d’une boucle for (si t==0, le boucle prend fin).

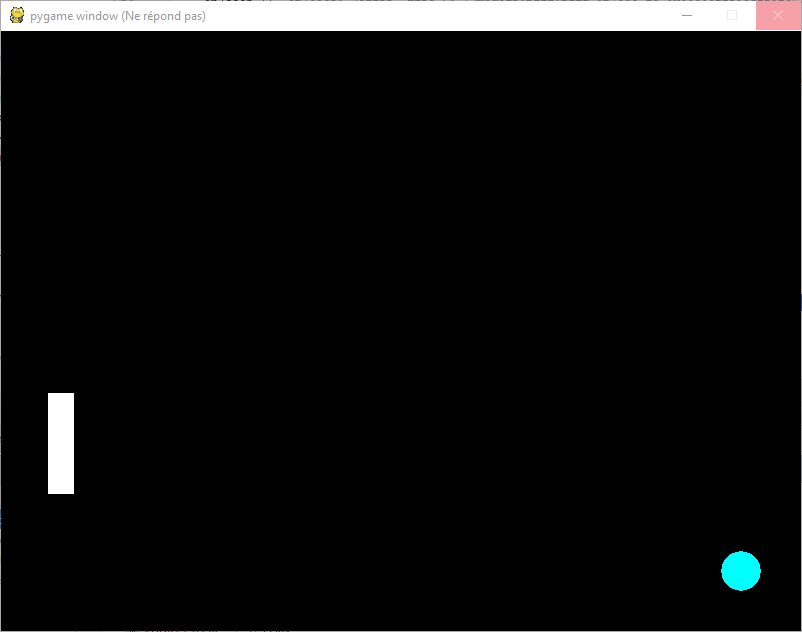
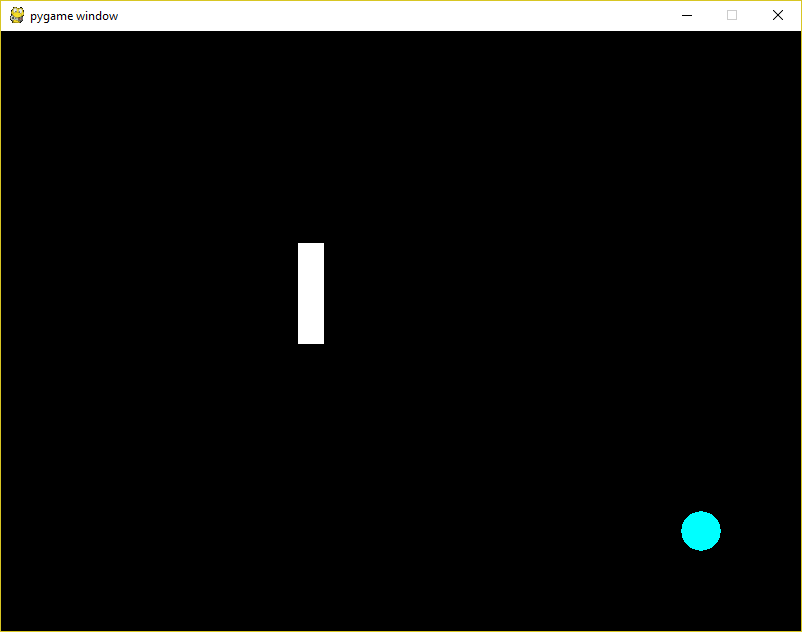
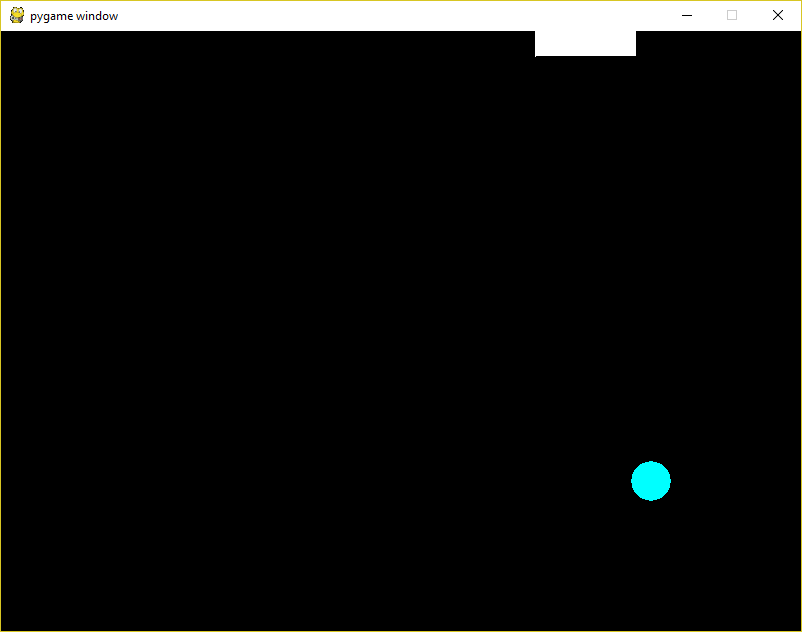
Le script résultant de l’interprétation du code 1 est le suivant :

**import** pygame, time, sys  
**from** pygame.locals **import** \*  
**from** pygame.locals **import** QUIT, MOUSEBUTTONDOWN, KEYDOWN, K\_RETURN  
**from** models.ball **import** Ball  
**from** models.triangle **import** Triangle  
**from** models.rectangle **import** Rectangle  
  
**def** refresh\_screen(objects, pygame, screen):  
 **for** obj **in** objects:  
 obj.draw(pygame, screen)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 pygame.init()  
 objects=[]  
 screen = pygame.display.set\_mode((1200, 800))  
 tick=60  
 ball0 = Ball(750, 550, 20, [0, 255, 255])  
  
 objects.append(ball0)  
 rect0 = Rectangle(10, 400, 100, 25, [255, 255, 255])  
  
 objects.append(rect0)  
 t = 10  
 **while**(t):  
 screen.fill(0)  
 print(t)  
 ball0.translate(-50, -50)  
 ball0.draw(pygame, screen)  
 rect0.rotate(3.1415926535 / 2)  
 rect0.draw(pygame, screen)  
 t = t - 1  
 refresh\_screen(objects, pygame, screen)  
 pygame.display.update()  
 time.sleep(tick/1000)  
 **for** event **in** pygame.event.get():  
 **if** event.type == QUIT:  
 sys.exit(0)  
 pygame.display.flip()  
 running = 1  
 **while** running:  
 **for** event **in** pygame.event.get():  
 **if** event.type == QUIT:  
 running = 0  
 **for** obj **in** objects:  
 obj.move()  
 obj.draw(pygame, screen)  
 refresh\_screen(objects, pygame, screen)  
 pygame.display.update()  
 time.sleep(tick/1000)

*Code 2. Exemple interprété*

La 1ère boucle while est celle qu’on a utilisé dans notre programme mais la 2ème boucle est destinée à garder un affichage lorsque qu’il n’y a plus d’animation et attend que l’utilisateur termine le programme au lieu de fermer bêtement la fenêtre une fois les instructions terminées. L’exéction du code 2 produit l’affichage suivant:

**Time**

*Fig. 1. Exécution de l’exemple*

Il s’agit seulement d’un petit exemple des possibilités de ce programme.

# Architecture

Tout code soumis au programme Animaker va être traité analyser et interprété de la façon suivante :

* Le code est analysé par des expressions régulières qui vont définir la nature de chaque mot ou caractère. Il s’agit de l’analyseur lexical ;
* Les mots de lexiques (qu’on appelle tokens) sont ensuite filtrer par l’analyseur syntaxique qui va vérifier la grammaire du texte et créer un arbre syntaxique abstrait. Cet arbre va servir à héberger la suite logique des instructions du futur programme animé.
* L’interpréteur va alors pouvoir utiliser l’arbre crée pour écrire le programme dans une fichier annexe et transformer chaque instruction en instruction adaptée utilisant la bibliothèque pygame.

## Analyse lexicale

Les principales choses qui ont été ajoutées par rapport à l’analyseur du cours sont des mots réservés (move, rotate, Ball, Rectangle, …). le caractère littéral « , » est aussi ajouté pour la syntaxe d’assignation des objets.

reserved\_words = (  
 …,  
 'move',  
 'rotate',  
 'Ball',  
 'Rectangle',

'Triangle',  
 'PI',  
 'tick',  
 'screen',  
 'translate')

tokens = (  
 …,   
 'COLOR'   
)

literals += ','

Le seul lexique reconnu par une expression régulière sont les variables définissant la couleur. L’expression détécte 3 chiffres séparées par une virgule. Par exemple : « 255, 255, 255 » pour du blanc.

def t\_COLOR(t):  
 r'\[\d+,\s+\d+,\s+\d+\]'  
 t.value = t.value  
 return t

### Résultats

Les résultats de sorties de l’analyseur lexical sont disponibles en exécutant le fichier « lexAnim.py ».

line 1: SCREEN(screen)

line 1: INTEGER(800)

line 1: INTEGER(600)

line 1: ;(;)

line 2: TICK(tick)

line 2: INTEGER(80)

line 2: ;(;)

line 3: IDENTIFIER(ball0)

line 3: =(=)

line 3: BALL(Ball)

line 3: INTEGER(750)

line 3: ,(,)

line 3: INTEGER(550)

line 3: ,(,)

line 3: INTEGER(20)

line 3: ,(,)

line 3: COLOR([0, 255, 255])

line 3: ;(;)

…

## Analyse syntaxique

L’analyse de la syntaxe engendre la production de l’arbre syntaxique abstrait qui va nous permettre, dans le code suivant qu’est l’interpréteur, de jouer avec les nœuds.

### Configurations

Les expressions de configurations définissent leur propres nœuds tick et screen avec les valeurs internes appropriées :

**def** p\_expression\_tick(p):  
 *"""statement : TICK int"""* p[0] = AST.TickNode(p[2])  
   
**def** p\_expression\_screen(p):  
 *"""statement : SCREEN int int"""* p[0] = AST.ScreenNode([p[2], p[3]])

### Objets

Les expressions d’objet sont définies comme cela. Par exemple la balle :

def p\_expression\_ball(p):  
*"""expression : BALL int ',' int ',' int  
| BALL int ',' int ',' int ',' color"""***if** len(p) == 7:  
 p[0] = AST.BallNode([AST.TokenNode(p[1]), p[2], p[4], p[6]])  
**elif** len(p) == 9:  
 p[0] = AST.BallNode([AST.TokenNode(p[1]), p[2], p[4], p[6], p[8]])

Ici l’analyse de la syntaxe :

BALL int ',' int ',' int ',' color

va engendrer la création d’un nœud BallNode lui-même composé d’un TokenNode (p[1]), de IntNode(p[2], p[4] et p[6]) et d’une ColorNode (p[8]).

* Le TokenNode héberge le type ;
* Les IntNode héberge les coordonées ;
* Le ColorNode (si spécifié 🡪 len(p) == 9) héberge la couleur de l’objet.

La IntNode, comme les autres nodes de terminales, définit sa propre valeur :

**class** IntNode(Node):  
 type = **'integer'  
 def** \_\_init\_\_(self, value):  
 Node.\_\_init\_\_(self)  
 self.value = int(value)

### Les mouvements

Le mouvement utilise des TokenNode et des IntNode pour modifier les valeurs de positions des objets

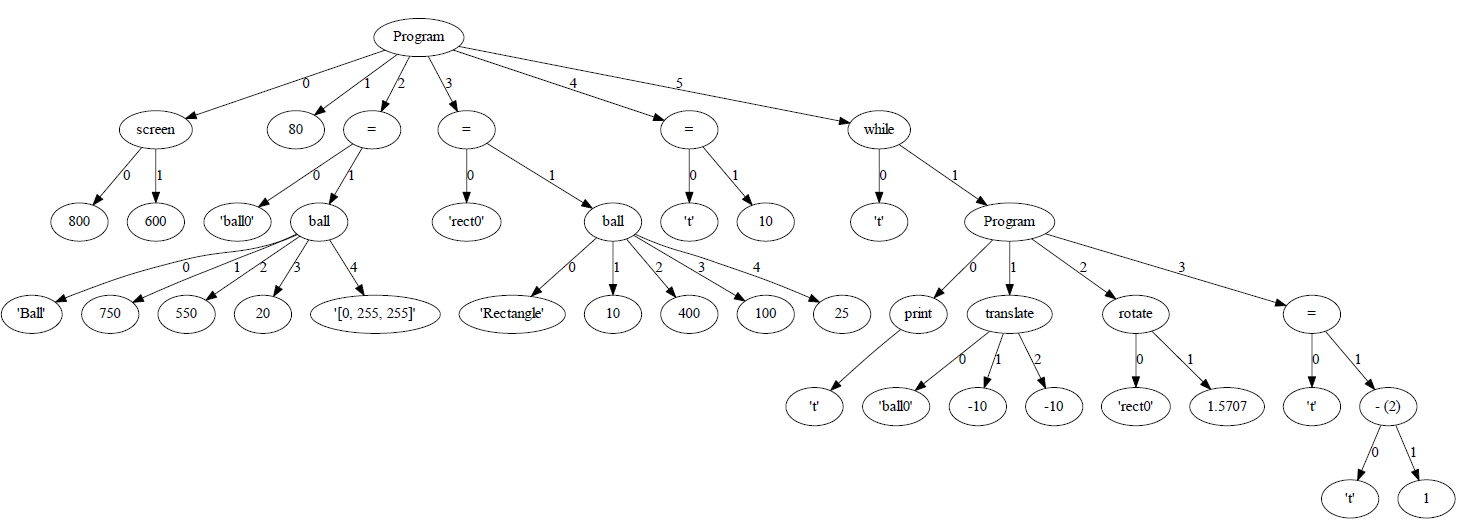
**def** p\_expression\_move(p):  
 *"""statement : MOVE IDENTIFIER int int"""* p[0] = AST.MoveNode([AST.TokenNode(p[2]), p[3], p[4]])

C’est également le cas pour translate. Rotate, quant à lui, utilise un float.

**def** p\_expression\_rotate(p):  
 *"""statement : ROTATE IDENTIFIER float"""* p[0] = AST.RotateNode([AST.TokenNode(p[2]), p[3]])

### Résultats

Voici un aperçu graphique des données qui sortent de l’analyseur syntaxique.



*Fig. 2. Sortie de l’analyseur lexical*

La forme logique de cet arbre va être utilisé par l’interpréteur. C’est ce qui va être expliqué au point suivant.

## Interprétation et création du fichier

L’interprétation est faite avec la récursivité. Pour faire ce dernier, nous nous sommes basés sur les différents travaux pratiques vus en cours.

Toutes les interprétations faites sont écrites dans un fichier Python utilisant la bibliothèque pygame. Pour cette dernière, il est impératif d’ajouter au tout début du fichier son initialisation et ses imports.

file.write("import pygame, time, sys\n")

file.write("from pygame.locals import \*\n")

file.write("from pygame.locals import QUIT, MOUSEBUTTONDOWN, KEYDOWN, K\_RETURN\n")

file.write("from models.ball import Ball\n")

file.write("from models.triangle import Triangle\n")

file.write("from models.rectangle import Rectangle\n\n")

Ensuite, on va traiter les nœuds par rapport à leur actions futures dans le code. Comme dit précédemment, tous les nœuds sont être écrits dans un fichier Python. Par exemple, pour un nœud de type BallNode, l’interpréteur va écrire le code pour créer un tel objet en python dans le fichier.

Il est également important, d’écrire à la fin du fichier python, une boucle infinie permettant à pygame de garder la fenêtre active. Dans le cas contraire, la fenêtre se fermerait automatiquement. Seul, le clique sur le bouton de fermeture pourra arrêter le programme.

running = 1

while running:

for event in pygame.event.get():

if event.type == QUIT:

running = 0

Voici une liste expliquant les différents nœuds :

|  |  |
| --- | --- |
| **Nœud** | **Description** |
| BallNode, RectangleNode, TriangleNode | Ces nœuds correspondent à un objet afficher dans pygame (un cercle, un rectangle, un triangle).  Ce nœud écrit dans le fichier la string de création d’un tel objet dans pygame |
| RotateNode, TranslateNode | Ces nœuds permettent de faire des rotations ou des translations aux objets créés.  Ce nœud écrit l’appel à la fonction concernée dans le fichier |
| IntNode,  FloatNode,  ColorNode | Ces nœuds renvoient le type de données que c’est (nombre entier, nombre flottant ou une couleur) |
| WhileNode | Ce nœud représente une boucle while, il va écrire dans le fichier tous les autres nœuds contenus exécuté dans la boucle. Il ne faut pas oublier de clear et rafraichir la fenêtre de pygame pour pouvoir voir les animations |
| AssignNode | Ce nœud est appelé lorsqu’il y a une assignation (=). On ajoute en tant que clé d’un dictionnaire le nom de la variable avec comme valeur, la valeur de la variable. On distingue avec 2 dictionnaires différents si ce sont des variables objets ou non. Cela permet par la suite de faire un rafraichissement de tous les objets dans pygame |
| OpNode | Ce nœud correspond à une opération arithmétiques basique (+, -, /, \*). Elles ne sont pas directement calculées par l’interpréteur mais écrit telles quelles dans le fichier python. |

## Gérer l’indentation Python

En Python, l’indentation est importante, il fallait donc gérer cette dernière lors de l’écriture de nos différentes fonctions. Pour pallier à ce problème, il suffit d’ajouter un paramètre dans toutes les fonctions « execute » donnant le nombre d’espaces d’indentation actuelle. Ainsi lorsque l’on est dans un nœud « while » on augment de 4 l’indentation pour les futurs nœuds à l’intérieur de cette « while ».

# Fonctionnalités / Prise en main

## Déclarations / Assignations

### Variables de nombre

Les variables de nombre peuvent servir à faire des calculs ou à servir de condition pour la boucle d’affichage :

a = 10;

b = 5.2;

c = -24;

d = a – b;

while(a){

…

};

### Variables objet

Avec les variables d’objets, on peut déclarer nos formes et garder des références. Cela va nous permettre de jouer avec leur position et de les afficher :

ball0 = Ball 1020, 20, 50;  
rect0 = Rectangle 0, 1050, 100, 100;

triangle0 = Triangle 0, 0, 100, 100, 0, 100;

## Affichages

### Console

L’affichage console est surtout utile pour afficher des indications au programmeur et éventuellement débugger.

a = 10;

print a;

## Configuration

### Taille de la fenêtre

On peut configurer la taille de la fenêtre d’affichage 2D pygame à l’aide de cette commande :

screen 800 600;

### Valeurs de tick

Toutes les variables d’objet peuvent être afficher à l’aide de la fonction show. Cela a pour effet de dessiner la forme dans la fenêtre d’affichage 2D pygame. Dans cet exemple, un rond :

ball0 = Ball 20, 20, 50;  
show ball0;

## Animations

### Mouvements

Un mouvement ne peut se faire uniquement à la fin de toutes les autres animations. Un mouvement permet de faire bouger pixel par pixel (en x comme en y) sur une longueur x et y donnée. A noter que tous les mouvements des objets en lieu "en même temps"

ball0 = Ball 20, 20, 50;

move ball0 -10 -10;

### Translations

Elle permet de translater un objet une seule fois du nombre de pixel en x et en y que l’on souhaite. Pour translater plusieurs fois un objet, il faut passer par une boucle while.

ball0 = Ball 20, 20, 50;

translate ball0 -10 -10;

### Rotations

On peut faire tourner un objet à l’aide de l’entrée suivante. Elle se base sur le centre de l’objet :

rect0 = Rectangle 0, 50, 100, 100;

rotate rect0 1.5707;

La fonctionnalité n’est malheureusement pas complétement opérationnelle dû à des erreurs mathématiques.

## Boucles

La boucle while va servir à modifier les valeurs de positions et réafficher les formes. C’est pourquoi on va l’utiliser alégrement. Le test de la variable t renvoi la valeur « False » uniquement si t==0.

while(t){  
 …

t = t – 1;  
};

L’exemple démontre l’application de la condition t pour la while avec un compteur mais rien n’empêche d’établir des algorithmes plus compliqués avec d’autres modifications de la valeur t.

# Problèmes rencontrés

## Erreurs de parsing

Le développement a connu quelques difficultés dans la réalisation de l’analyse syntaxique. Il a fallu bien prendre en main le système de nœud AST pour bien structurer les informations.

## Problèmes d’import

Le mot clé « parser » utilisé comme nom de fichier pour l’analyseur syntaxique engendre des difficultés d’import. On n’en connait pas la source.

## Rotation

La rotation est compliquée, il faut traiter et déplacer chaque point séparément en tenant compte du centre estimé de l’objet.

# Conclusion

Les objectifs sont atteints :

* L’analyse lexicale fonctionne bien mais on pourrait séparer les types de classe par une expression régulière d’une suite de lettre commençant par une majuscule par exemple. On privilégie davantage la sécurité en réservant chaque mot mais on perd en flexibilité.
* L’analyseur syntaxique est fonctionnel et l’arbre syntaxique est très propre.
* L’interprétation est indirecte dans le sens où on aurait pu créer un contexte interne. L’interpréteur aurait eu un contexte pygame et aurait par exemple utilisé la while pour afficher. Néanmoins, on a préféré mettre en avant la portabilité en générant un script utilisant pygame qui lui peut être lancé de manière autonome. De cette manière, le fichier n’a également pas besoin d’être réinterprété pour être exécuté.