Geneva University

Master Thesis

Computer Science department

Une approche déclarative pour la programmation graphique

Patrick Sardinha

Year 2020 - 2021

Table des matières

Table des matières **1**

1 Introduction **4**

2 Travaux connexes **8**

2.1 OpenGL, DirectX et Gator 8

2.2 Outils pour la programmation graphique 10

2.3 Langages déclaratifs et React 12

3 Modèle **15**

3.1 Rendery et abstractions 15

3.2 Programmation déclarative et par blocs 17

3.3 PATL 22

3.3.1 Sémantique statique de typage de PATL 22

3.3.2 Sémantique opérationnelle de PATL 24

4 Implémentation **29**

4.1 Rendery et React 29

4.2 PATL 30

4.3 Cas pratique : Système Solaire 35

4.4 Didacticiel 36

5 Evaluation **38**

6 Conclusion et travaux futurs **40**

Remerciements **42**

Références **43**

Annexes **45**

Résumé

La programmation graphique permet de créer des applications 3D et est notamment utilisée pour la conception de jeux vidéo ainsi que pour la modélisation. A l'heure actuelle, la spécification OpenGL est l'une des plus utilisées pour la création d'applications graphiques et offre un ensemble de fonctions donnant la possibilité aux programmeurs de déclarer et de manipuler des objets 3D ainsi que des images. Cependant, l'utilisation d'une telle spécification demande de coder à relativement bas niveau et de bonnes connaissances dans la programmation graphique. Il est de ce fait assez compliqué de concevoir ce type d'applications. L'utilisation d'un langage de programmation haut niveau incluant des abstractions sur les types est un moyen de faciliter la conception de ces dernières. Le but de ce travail est de créer un nouveau langage nommé PATL, ayant pour but d'être complètement déclaratif, de proposer une implémentation par blocs et de permettre l'utilisation d'abstractions pour la création des applications et la représentation des objets 3D. Il s'agit d'un *Domain-Specific Language* (DSL) orienté Swift inspiré d'un langage déclaratif : React. Ce nouveau langage de programmation a pour but de permettre la création et l'utilisation de types abstraits et d'ajouter une couche déclarative à Rendery, un moteur de rendu 2D/3D moderne qui est écrit en Swift et basé sur OpenGL. L'utilisation de PATL avec Rendery vise à ne pas avoir à manipuler les fonctions OpenGL et ainsi de créer des applications graphiques de manière simplifiée tout en utilisant le langage Swift.

Mots-clés Programmation graphique ; Programmation déclarative ; OpenGL ; Swift ; Rendery ; React ; PATL.

1 Introduction

Le développement d'applications graphiques est essentiel pour les domaines scientifiques, industriels mais aussi artistiques notamment depuis l'expansion à grande échelle du domaine des jeux vidéo. De nombreuses *Application Programming Interface* (API) ont ainsi été créées dans le but de promulguer des bibliothèques et de faciliter la création d'applications graphiques. Parmi ces APIs, on retrouve notamment OpenGL (Open Graphics Library) [1] développé par Silicon Graphics [2] et Kronos Groups [3], DirectX [4] développé par Microsoft [5] et Vulkan [6] également développé par Kronos Groups visant à remplacer OpenGL.

De nombreux programmes sont basés sur OpenGL dont la plupart sont liés à la photographie/vidéo, à la modélisation, à la visualisation ainsi qu'aux jeux vidéo. Parmi ces programmes, les plus populaires sont *Adobe Photoshop*, un logiciel de retouche photo et graphique, *Blender* et *Google SketchUp* qui sont respectivement un moteur d'animation et de jeu ainsi qu'un modeleur 3D, *Google Earth*, un logiciel de cartographie de la Terre et *Minecraft* ou *Doom*, deux jeux vidéo de type bac à sable et FPS [7].

Une application graphique OpenGL se base sur un certain nombre d'éléments principaux, notamment sur la notion de contexte ainsi que de fenêtre dans laquelle la scène, composée d'objets, est affichée. Une boucle de rendu permet d'afficher de manière continue des images sur cette fenêtre, de vérifier si des événements sont survenus (déclenchement de touches clavier, clics de souris, etc.) et de mettre à jour l'état de la fenêtre si nécessaire. Pour l'affichage des objets, il est nécessaire de fournir des données d'entrées à la première étape du pipeline graphique, le processus principal permettant le rendu d'objets, puis de configurer la façon dont ces données doivent être interprétées.

1. https ://www.opengl.org
2. https ://en.wikipedia.org/wiki/Silicon_Graphic
3. https ://www.khronos.org
4. https ://docs.microsoft.com/en-us/windows/win32/direct
5. https ://www.microsoft.com/en-us
6. https ://www.khronos.org/vulkan
7. https ://en.wikipedia.org/wiki/List_of_OpenGL_application

Les bibliothèques proposées, qui ont été évoquées auparavant, sont basées sur différents langages de programmation tels que le C pour OpenGL et le C++ pour DirectX. Ces APIs utilisent chacune leur propre langage de programmation de *Shader* [8] qui sont respectivement GLSL (OpenGL Shading Language) [1], HLSL (DirectX High-Level Shader Language) [2] et aussi GLSL pour Vulkan. Les shaders sont des programmes informatiques exécutés par le processeur graphique (GPU) et permettent de paramétrer une partie du processus de rendu.

Cependant, l'utilisation de ces APIs demande un bon niveau de connaissances sur les langages utilisés par les librairies ainsi que sur la façon d'implémenter les fonctions proposées. Il en est de même concernant les langages de programmation de *shaders*. De plus, il est aussi indispensable de comprendre le fonctionnement du *pipeline* graphique [3] ainsi que la façon dont les données nécessaires pour la création d'objets 3D sont gérées en mémoire et configurées pour être par la suite traitées de la bonne manière. Ces spécifications obligent les programmeurs à manipuler les données concernant les différents objets 3D à un niveau d'abstraction relativement bas nécessitant ainsi l'utilisation de structures de données telles que vecteurs ou matrices.

```
1   float vertices[] = {
2        // positions          // colors
3        0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,
4       -0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,
5        0.0f,  0.5f, 0.0f,   0.0f, 0.0f, 1.0f
6   };
```

FIGURE 1 – Représentation des données pour un triangle 2D

A titre d'exemple, la Figure 1 représente la structure de données nécessaires pour créer un triangle 2D. Chaque sommet du triangle est ainsi représenté à l'aide de six valeurs dont les trois premières spécifient la position du sommet alors que les trois dernières précisent sa couleur. Cet exemple reste relativement simple pour un objet en deux dimensions mais pour une application impliquant de nombreux objets de dimensions supérieures, par exemple une application 3D, manipuler les données devient rapidement complexe et fastidieux.

8. https ://fr.wikipedia.org/wiki/Shader

Comme la création d'une application graphique nécessite la mise en place en général d'un grand nombre d'éléments, cela se traduit ainsi souvent en de longs codes et assez complexes. De plus, le redondance de codes concernant par exemple l'initialisation, la mise en place ou les opérations sur les objets est très fréquente dans ce genre d'application.

Ainsi, l'utilisation d'un niveau d'abstraction supplémentaire pour créer et manipuler des objets, passer par le paradigme de programmation déclaratif pour décrire le problème de l'application et entre autre réduire la taille du code et mettre en place un système de programmation par blocs dans le but de décomposer le code en composants indépendants et imbricables, semblent être de bonnes solutions pour faciliter le développement d'applications graphiques. Cependant, dans la littérature il n'existe pas, à notre connaissance, de proposition de langage haut niveau proposant ces solutions.

Ce travail propose ainsi de créer un nouveau langage, nommé PATL [4]. Il s'agit d'un DSL orienté Swift s'inspirant de React. Swift est un langage de programmation haut niveau, simple, performant et sûr, nous permettant de mettre en place aisément des abstractions sur les types. A la base, React est une bibliothèque JavaScript pour créer des interfaces utilisateurs. Pour le nouveau langage proposé, l'idée est d'utiliser le principe déclaratif ainsi que la notion de composants de React. Les différents composants définis dans une application auront pour but d'être combinés les uns avec les autres et affichés afin d'obtenir le rendu d'une scène 3D complexe. PATL se base sur Rendery qui est un moteur de rendu 2D/3D écrit en Swift dont son but est de fournir une interface de programmation simple et intuitive pour écrire des applications graphiques en se basant sur la spécification d'OpenGL.

Notre nouveau langage va réutiliser les abstractions proposés par Rendery concernant notamment la mise en place d'une application ainsi que la création et la manipulation des objets. Il va de ce fait, ajouter une couche déclarative à Rendery, réutiliser un des principes fondamentaux de React qui est la décomposition du code en blocs et permettre à l'aide du système de structure de Swift de créer de nouvelles abstractions sur les types.

PATL a ainsi pour but de faciliter le développement d'applications graphiques en Swift en ne se souciant pas de la spécification d'OpenGL.

Nous évaluons expérimentalement notre langage avec différents cas pratiques d'implémentation. De plus, un tutoriel est développé dans le but de fournir une première documentation du langage, visant à expliquer ses concepts ainsi que son utilisation.

La structure du papier va suivre le plan suivant : dans un premier temps, la section 2 abordera les travaux en lien avec notre sujet. Puis dans un second temps, la section 3 étudiera la méthodologie proposée. La section 4 clarifiera l'implémentation de celle-ci et proposera l'étude d'un cas pratique. La section 5 évaluera la méthode proposée. Dans un dernier temps, la section 6 évoquera les conclusions de ce travail et de possibles travaux futurs.

2 Travaux connexes

2.1 OpenGL, DirectX et Gator

OpenGL est l'une des APIs les plus utilisées pour la programmation d'applications graphiques. On peut plus exactement parler de bibliothèque logicielle ou de spécification qui regroupe un grand nombre de fonctions permettant l'affichage de scènes 3D. La première version d'OpenGL a été développée en 1994 et par la suite de nouvelles versions sont sorties. OpenGL s'est imposée du fait de son accessibilité au travers des différentes plate-formes. Le plus important concurrent d'OpenGL est DirectX, une collection de bibliothèques pour la programmation d'applications graphiques développée par Microsoft et pour les plate-formes Microsoft, dont le lancement s'est fait en septembre 1995. Tout comme OpenGL, il s'agit d'une API bas niveau et nous pouvons d'ailleurs remarquer une grande similitude sur la façon d'implémenter des applications graphiques entre les deux APIs.

Dans la littérature, une étude propose de comparer OpenGL et DirectX dans l'objectif d'explorer les similitudes ainsi que les différences entre ces deux APIs [5]. Le papier aborde tout d'abord la progression des principales versions d'OpenGL et de DirectX puis compare les spécifications. Une grande différence peut ici déjà être observée concernant les spécifications puisque celles d'OpenGL sont publiques et gratuites ce qui n'est pas le cas de celles de DirectX. Les deux APIs peuvent intégrer des *frameworks* pour faciliter leur utilisation. Il en existe de nombreux pour chacune de ces deux APIs dont par exemple, Freeglut ou GLFW pour OpenGL et DXUT pour DirectX. OpenGL et DirectX utilisent chacun leur propre langage de *shaders*, respectivement GLSL et HLSL et partagent les mêmes types de *shaders* ainsi que les mêmes équivalences de types de données (exemple : int, double, vec, mat, etc.). La structure générale des *vertex shaders* ainsi que des *fragment shaders* est similaire entre GLSL et HLSL. Après avoir écrit un *shader*, ceux-ci sont configurés de la même façon, à quelques différences près (encapsulation des programmes *shaders* du côté d'OpenGL pour l'optimisation), pour être ensuite exécutés sur le GPU. Finalement, pour passer des données au GPU, OpenGL tout comme DirectX utilise le système de *buffer* où l'idée est de pouvoir leur associer des noms, de les lier ainsi que de leur passer des données.

En résumé, OpenGL et DirectX sont deux APIs bas niveau obligeant ainsi les programmeurs à manipuler des objets, configurer des structures et implémenter des algorithmes de rendu avec précision.

Comme évoqué auparavant, ces deux APIs utilisent leur propre langage de *shader*, il s'agit de GLSL pour OpenGL et de HLSL pour DirectX, tous les deux ayant une syntaxe basée sur le langage C. On distingue principalement deux types de *shaders*, les *vertex shaders*, calculant la position des objets et les *fragment shaders* pour la couleur des pixels. Il s'agit de programmes exécutables par la carte graphique dont le but est de gérer une partie du processus de rendu. Ces deux langages, GLSL et HLSL sont des langages de plus haut niveau par rapport à la façon dont étaient implémentés les *shaders* auparavant (syntaxe assembleur), cependant développer des *shaders* reste une tâche relativement compliquée, pas forcément intuitive et surtout très répétitive.

Dans la littérature, un nouveau système de typage et un langage nommé Gator [6] ont été proposés afin d'éviter les bogues de géométrie, des bogues difficiles à résoudre et pouvant facilement être générés en utilisant des langages comme GLSL par exemple. L'idée générale est de refléter le système de coordonnées de chaque objet géométrique et ainsi de générer des erreurs pour des opérations géométriquement incorrectes. Pour ce faire, les auteurs ont mis en place un "type de géométrie" impliquant trois composants :

- le *coordinate scheme* (le type de coordonnées : cartésiennes, etc.)
- le *reference frame* (la vue : locale, monde, etc.)
- et le *geometric object* (le type d'objet : point, ligne, etc.)

définissant ainsi les opérations pouvant ou, au contraire, ne pouvant être effectuées. Ce nouveau type donne ainsi plus d'informations sur les objets qu'un simple vecteur et sa syntaxe est la suivante : *scheme* ⟨*frame*⟩ *.object*.

Par exemple, pour représenter le type d'un point dans l'espace monde, représenté lui-même avec des coordonnées cartésiennes 3D, on aura : *cart3* ⟨*world*⟩ *.point*. A partir de là, des opérations sur des objets ayant le même type mais un type de géométrie différent impliquent des erreurs et évitent ainsi la génération de bogues.

Dans un langage de *shaders* comme GLSL ou basé sur GLSL, ce nouveau type prend tout son sens puisqu'il est fréquent de manipuler des objets appartenant à différents systèmes de coordonnées. Cependant, la création des *shaders* est déjà une tâche compliquée et l'utilisation de ce type alourdi fortement le code et le rend de ce fait moins maintenable.

Pour atténuer le problème de la tâche répétitive concernant le développement des *shaders*, différents outils ont été créés tels que par exemple, Shadertoy BETA [7] ou Geeks3D [8] permettant de pouvoir écrire des *vertex* et des *fragment shaders* sur *browser*, d'observer en temps réel le rendu de ceux-ci puis de les exporter afin de les réutiliser dans d'autres applications. Ces outils mettent également à disposition de nombreux *shaders* déjà implémentés dans le but de les importer facilement.

2.2 Outils pour la programmation graphique

Le langage Swift [9] est un langage de programmation objet compilé, développé en open source par Apple. Il s'agit d'un langage relativement jeune (2014) dont l'idée principale est d'être simple et performant. De plus, la documentation fournie est très complète et aide ainsi à prendre facilement en main le langage. Ce sont pour ces raisons que ce langage devient de plus en plus populaire chaque année et ne cesse de monter dans le classement TIOBE [10] des langages de programmation les plus utilisés.

Apple a créé différents outils pour la programmation graphique dont SpriteKit [9], pour la création de jeux 2D et propose une classe d'objets SKShader [11] afin de créer des *fragments shaders* personnalisés. Similairement, il existe un autre outil nommé SceneKit [10] pour donner la possibilité de développer des applications 3D avec le langage Swift. Avec SceneKit, il est ainsi possible de créer, par exemple, des scène 3D complexes pour les jeux vidéo ou alors des simulations physiques. Le principe de SceneKit est de promouvoir une API relativement haut niveau pour permettre de manipuler et d'afficher des objets 3D voulus.

9. https ://swift.org/documentation
10. https ://www.tiobe.com/tiobe-index/
11. https ://developer.apple.com/documentation/spritekit/skshader

Cependant, du fait que Swift soit un langage relativement récent, les outils proposés à l'heure actuelle dans ce langage ne permettent pas de développer une application graphique, tel qu'un jeu vidéo par exemple, aussi facilement qu'avec un autre langage ayant plus de bibliothèques à sa disposition.

Une proposition pour palier au problème décrit précédemment est Rendery [11], un moteur de rendu 2D/3D écrit en Swift, permettant d'écrire des applications graphiques de manière simple et intuitive. Rendery a pour but de minimiser le nombre de ses dépendances externes et a aussi été développé afin d'être une alternative multiplateforme à SpriteKit et SceneKit. Il se base sur la spécification OpenGL (3.30) tout en apportant un niveau d'abstraction supérieur sur les procédés techniques permettant d'obtenir le rendu de scènes tridimensionnelles. Le fait que Rendery soit une librairie autonome permet d'importer facilement la librairie dans un nouveau projet et facilite alors la création d'applications graphiques en Swift sans avoir à se soucier de la spécification d'OpenGL.

Concernant les langages dédiés, *Domain-Specific Language* en anglais ou DSL, nous pouvons les définir comme "des langages de programmation dont les spécifications sont conçues pour répondre aux contraintes d'un domaine d'application précis" [12]. Leur utilisation offre certains avantages mais implique également des désavantages. De manière générale, ces derniers vont permettre un gain considérable en termes de productivité et vont pouvoir être réutilisés par la suite à d'autres fins. Cependant, les principaux désavantages sont d'un côté le coût, qui est généralement élevé et d'un autre, la maintenance, qui est compliquée [12]. Il existe différents types de langages dédiés, les DSL internes aussi appelés *Embedded DSL* (EDSL) et les DSL externes. Un langage dédié externe est construit à l'aide d'un *compilateur*, reconnaissant la syntaxe du nouveau langage alors qu'un langage dédié interne est construit à partir d'un langage hôte, réutilisant son infrastructure [13]. Ce dernier est donc plus limité au niveau de la syntaxe et certains sacrifices sont nécessaires à faire en fonction du langage hôte mais en contrepartie cela permet de ne pas tout ré-implémenter et de partir sur de solides fondations.

12. https ://en.wikipedia.org/wiki/Domain-specific_language

2.3 Langages déclaratifs et React

Les langages déclaratifs sont, quant à eux, des langages dans lesquels nous définissons le problème et non pas la solution du problème [13]. Selon une étude comparant les paradigmes de programmation impératif et déclaratif [14], un des points faisant la force des langages déclaratifs est que les calculs ne dépendent pas de l'état du système, ni le modifient. Il en découle que les principaux avantages de ces langages sont tout d'abord qu'un programme peut être considéré par partie impliquant donc une facilité dans la réutilisation du code ainsi que dans la gestion et le débogage des erreurs. Ils donnent aussi la possibilité de programmer à plus haut niveau comparé à un langage de programmation impératif. Finalement, un langage déclaratif permet généralement de réduire la quantité de code produit dans l'optique de simplifier la conception d'une application. Un exemple simple de langage déclaratif est HTML, dans lequel on décrit les éléments que contient une page mais pas la façon de les afficher. Sur le même principe, React, étant une bibliothèque Javascript pour construire des interfaces utilisateurs, utilise le principe de vues déclaratives et la notion de composants afin d'obtenir un rendu complexe de manière simple, intuitive et facilement débogable.

Créer un EDSL basé sur le langage Swift, orienté vers le paradigme de programmation déclaratif et la décomposition du code par blocs sur la même idée que React et se baser sur Rendery semble être une façon élégante et optimale afin de créer des applications graphiques de manière simple et intuitive sans forcément posséder de grandes connaissances dans ce domaine. Le nouveau langage PATL a ainsi été développé dans ce but.

Nous listons en Table 1, sept des fonctionnalités les plus intéressantes pour un langage de programmation graphique visant à faciliter la conception des programmes. Ces fonctionnalités sont les suivantes :

13. https ://en.wikipedia.org/wiki/Declarative_programming

- personnaliser le *render pipeline* (liberté dans la programmation)
- langage de programmation haut niveau
- permet des abstractions sur les types
- langage facile à déboguer (proposant une syntaxe simple et claire lors de la conception des programmes)
- langage facile d'utilisation (documentation fournie, tutoriel, etc.)
- développement graphique intuitif (impliquant des éléments spécialement dédiés pour les applications graphiques)
- langage complètement déclaratif
- et décomposition du code par blocs.

Ces différentes fonctionnalités sont ainsi comparées entre tous les outils/APIs étudiés dans cette partie, c'est-à-dire entre OpenGL, DirectX, SceneKit, Rendery et PATL.

	OpenGL	DirectX	SceneKit	Rendery	PATL
Personnaliser le *render pipeline*	Oui	Oui	Non	Non	Non
Langage de programmation haut niveau	Non	Non	Oui	Oui	Oui
Permet des abstractions sur les types	Non	Non	Oui	Oui	Oui
Langage facile à déboguer	Non	Non	Oui	Oui	Oui
Langage facile d'utilisation	Non	Non	Oui	Oui	Oui
Développement graphique intuitif	Non	Non	Non	Oui	Oui
Langage complètement déclaratif	Non	Non	Non	Non	Oui
Décomposition par blocs	Non	Non	Non	Non	Oui

Table 1 – Comparaison de différents outils pour la programmation d'applications graphiques

A l'aide du tableau, nous pouvons observer qu'OpenGL ainsi que DirectX ne remplissent que la première des fonctionnalités proposées impliquant une certaine liberté de programmation avec ses APIs mais démontrant aussi les manques évoqués de ces dernières. Contrairement, pour Swift et Rendery, la majeure partie de ces fonctionnalités sont remplies à l'exception de la première et des trois dernières pour Swift et des deux dernières pour Rendery. Nous pouvons finalement voir que PATL possède l'intégralité de ces fonctionnalités (à l'exception de la première) et propose une implémentation déclarative et par blocs par rapport à Rendery.

3 Modèle

Nous allons présenter dans cette section le langage PATL et les fondements sur lesquels celui-ci se base.

3.1 Rendery et abstractions

La création d'une application graphique en utilisant la spécification d'OpenGL va impliquer la mise en place d'un certain nombre d'éléments essentiels tels que la notion de contexte OpenGL, de fenêtre ainsi que de pipeline graphique.

Le contexte et la fenêtre de l'application vont définir l'environnement dans lequel une scène tridimensionnelle pourra être mise en place dans le but d'être par la suite observable sur l'écran. Le pipeline graphique, aussi appelé pipeline 3D, est quant à lui, le processus de transformation de coordonnées 3D en pixels 2D. Ce processus est au centre même de toutes les APIs pour le rendu graphique. Il peut être décomposé en deux étapes majeures : transformer dans un premier temps les coordonnées 3D en coordonnées 2D puis, dans un second temps, transformer ces coordonnées 2D en pixels affichables sur un écran. Pour ce faire, ce processus va suivre une suite d'étapes où la sortie de l'une composera l'entrée de la suivante. Certaines de ces étapes font intervenir des *shaders*, de petits programmes configurables par le développeur et exécutables par le GPU. Ces étapes permettent ainsi de gérer une partie du processus du rendu, ce qui n'est pas le cas pour les autres. La Figure 2 représente le schéma du pipeline graphique.

Le pipeline est composé ainsi : une étape d'initialisation avec l'entrée des données, vient ensuite l'étape du *vertex shader* dont le rôle est de calculer la projection des sommets de l'espace 3D initial vers un autre espace 3D nommé NDC (Normalized Device Coordinates). Pour que les sommets d'un objet soient visibles, ceux-ci doivent avoir une représentation dans cet espace [14]. Vient ensuite l'étape de la *primitive assembly* où l'idée est de créer des formes géométriques en assemblant les sommets de l'étape précédente. L'étape suivante est la *tesselation* permettant d'augmenter le niveau de détail des formes géométriques dont les surfaces sont construites à partir de triangles. Après cette étape, on

14. https ://learnopengl.com/Getting-started/Coordinate-Systems

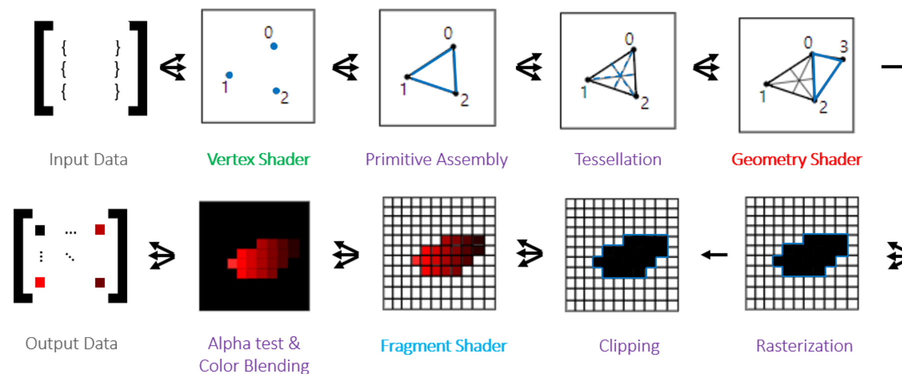

FIGURE 2 – Schéma du pipeline graphique

trouve une étape optionnelle, le *geometry shader* permettant entre autre de transformer les sommets formant une primitive (par exemple un triangle) en une autre primitive. Il y a ensuite la *rasterization* puis le *clipping*. Ces étapes servent respectivement à convertir une image vectorielle (c'est-à-dire composée d'objets géométriques) en une image Raster ou Bitmap (composée de pixels) et à supprimer tous les fragments qui ne seront pas visibles dans le but d'augmenter les performances. L'étape suivante est le *fragment shader* qui va calculer la couleur des pixels. Finalement, les deux dernières étapes sont l'*alpha test* et le *color blending* pour déterminer l'ordre d'affichage des différents fragments et lisser les rendus des couleurs. On obtient ainsi à la fin du processus la valeur RGB (Red, Green, Blue) de chaque pixel de l'écran [15].

Comme évoqué auparavant, la manipulation et la configuration des données relatives au pipeline graphique est complexe et nécessite un certain niveau dans la programmation graphique. Il en est de même pour la définition ainsi que la création du contexte et de la fenêtre de l'application. Avec Rendery, il est ainsi possible, grâce aux abstractions mises en place, de créer une application de manière simplifiée mais aussi de définir et de manoeuvrer aisément des objets. En effet, dans une scène, un objet va être perçu comme un noeud pouvant être affecté par des propriétés telles que typiquement un nom, une position ou un modèle. A l'aide de cette dernière, il est possible de définir le noeud comme un objet 3D en spécifiant un *mesh* en particulier tel que par exemple une sphère.

15. https ://learnopengl.com/

La Figure 3 ci-dessous présente la création d'un objet 3D représentant une sphère. L'objet est tout d'abord défini comme un noeud (ligne 1) et ses propriétés sont ensuite définies telles que son nom (ligne 2), son modèle donnant des informations sur sa forme (ligne 4) et sa couleur (ligne 5) ainsi que sa position (ligne 6 à 8).

```
1        let sphereNode = root.createChild()
2        sphereNode.name = "Sphere"
3        sphereNode.model = Model(
4            meshes: [.sphere(segments: 50, rings: 50, radius: 10.0)
                 ],
5            materials: [Material()])
6        sphereNode.translation.x = 5.0
7        sphereNode.translation.y = 10.0
8        sphereNode.translation.z = 2.0
```

FIGURE 3 – Création d'une sphère avec Rendery

3.2 Programmation déclarative et par blocs

La programmation déclarative et la programmation impérative sont deux paradigmes de programmation opposés pouvant être respectivement définis comme suit :

Définition : "La programmation déclarative consiste à créer des applications sur la base de composants logiciels indépendants du contexte et ne comportant aucun état interne. En programmation déclarative, on décrit le quoi, c'est-à-dire le problème" [15].

Définition : "La programmation impérative décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. On décrit le comment, c'est-à-dire la structure de contrôle correspondant à la solution." [16].

Nous pouvons illustrer cette différence à l'aide d'un petit programme Swift dans lequel le but est d'incrémenter de un tous les éléments d'une liste. De manière déclarative (Figure 4), nous ne spécifions aucun détail sur la façon d'obtenir le résultat mais seulement le résultat voulu. La fonction `inc()` définit ainsi ce que l'on fait et non pas comment on le fait.

Alors qu'au contraire, de manière impérative (Figure 5), on définit chaque étape pour obtenir le résultat.

```
1  var x = [1,2,3,4]
2
3  func inc(x: Int) -> Int {
4     return  x + 1
5  }
6
7  x.map(inc)
8
9  >>> [2,3,4,5]
```

FIGURE 4 – Code Swift déclaratif

```
1  var x = [1,2,3,4]
2  var y:[Int] = []
3
4  for el in x {
5    y.append(el+1)
6  }
7
8  y
9
10 >>> [2,3,4,5]
```

FIGURE 5 – Code Swift impératif

Dans cet exemple, nous pouvons donc observer que la différence majeure provient du contrôle de flow. Dans le premier cas, on ne donne pas la manière dont on parcourt la liste contrairement au second.

Pour la programmation d'applications graphiques, passer par le paradigme de programmation déclaratif apporte un certain nombre d'avantages non négligeables par rapport aux inconvénients engendrés.

En effet, l'un des avantages les plus importants concerne la réduction de la taille des codes lors de la conception d'une application et implique de ce fait la possibilité de développer des programmes complexes et longs, ce qui est souvent le cas pour les applications graphiques, de manière synthétique.

Ceci permet d'améliorer la clarté du code et de faciliter sa maintenance mais implique en contre partie aussi beaucoup plus de difficulté pour des tiers à lire et à comprendre le code [16].

D'une manière générale, comme la programmation déclarative se concentre sur l'objectif à atteindre, ce paradigme est ainsi plus adapté que celui de la programmation impérative pour la création de scène 3D puisqu'il permet de transcrire facilement et surtout intuitivement en code, les représentations graphiques voulues.

En plus de l'aspect déclaratif à la base de notre nouveau langage, nous avons décidé de mettre en place un système de programmation par blocs dans l'optique de faciliter le développement. La division du code d'une application graphique en blocs, que nous nommons composants, nous permet dans un premier temps de considérer un élément graphique tel que par exemple un cube ou une caméra, de manière indépendante et donc de le manipuler individuellement sans impacter les autres éléments. Et dans un deuxième temps, de pouvoir imbriquer et combiner les différents composants afin de pouvoir obtenir des constructions plus complexes. Cette construction par blocs renforce donc le but de faciliter la conception d'une application et de pouvoir la maintenir ainsi que de la déboguer aisément en isolant les possibles erreurs de conception.

Sur ce principe de la décomposition du code, il semble intéressant de mettre en place un système de hiérarchie entre les différents blocs dans le but d'obtenir des codes plus structurés et ainsi plus efficaces. L'idée est de définir dans une application, un composant haut niveau, le composant principal de l'application dont le but est de gérer l'état du système et le rendu global ainsi que de multiples composants bas niveau, récupérant les informations fournis par le composant haut niveau et visant à refléter seulement une partie de la scène.

Le schéma en Figure 6 représente l'idée générale de la programmation par blocs du nouveau langage proposé dans ce papier. Nous pouvons ainsi observer le principe d'encapsulation des différents composants, les liens qui les unissent et le système de hiérarchie entre ces derniers.

16. https ://www.ionos.fr/digitalguide/

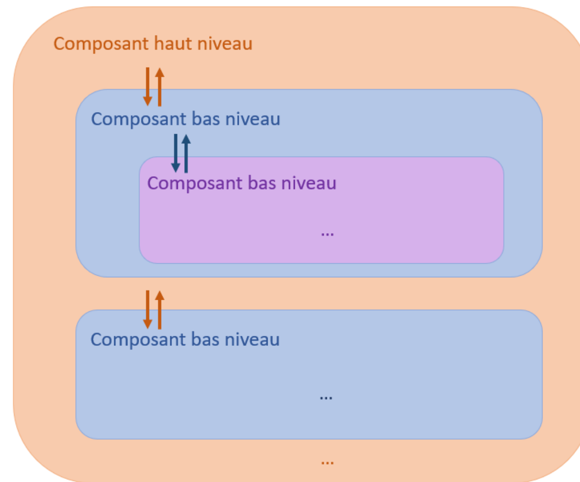

FIGURE 6 – Décomposition du code dans PATL

Pour la création de notre nouveau langage, nous nous sommes ainsi focalisé sur React, une bibliothèque Javascript permettant de créer facilement des interfaces utilisateurs. Cette librairie se base sur la notion de composants autonomes pouvant être combinés pour obtenir des rendus plus complexes, dont chacun maintient son propre état et où l'idée est de mettre à jour de façon optimale seuls ceux dont les données changent. Les composants React sont codés de manière déclarative donnant ainsi aux programmeurs davantage de contrôle sur le code [17]. Un composant React est défini sous forme de classe implémentant au minimum une méthode de rendu dont le but est de retourner les éléments devant être affichés et est ainsi appelée à chaque modification de l'état du composant. Il est possible pour un composant de recevoir des données d'un autre composant et une fonction pour mettre à jour l'état d'un composant de manière régulière est habituellement implémentée. Finalement, une fonction principale permet d'appeler le premier composant de l'application.

17. https ://fr.reactjs.org/

Ci-dessous, en Figure 7, se trouve un exemple d'application React pour un *Timer* affichant toutes les secondes la nouvelle valeur[17]. La classe *Timer* représente un composant React dans lequel l'état de ce dernier est initialisé avec la valeur du timer. La fonction `componentDidMount()` permet d'appeler à chaque seconde la fonction `tick()` dont son but est d'incrémenter de 1 la valeur du compteur. Au contraire, la fonction `componentWillUnmount()` permet quant à elle de stopper l'affichage de ce composant. Finalement, la fonction de rendu `render()` est appelée dans le but d'afficher la valeur actuelle du timer à l'écran. L'application fonctionne ainsi en appelant le composant *Timer* via la fonction principale `ReactDOM.render()`.

```
1   class Timer extends React.Component {
2     constructor(props) {
3       super(props);
4       this.state = { seconds: 0 };
5     }
6
7     tick() {
8       this.setState(state => ({seconds: state.seconds+1}));
9     }
10
11    componentDidMount() {
12      this.interval = setInterval(() => this.tick(), 1000);
13    }
14
15    componentWillUnmount() {
16      clearInterval(this.interval);
17    }
18
19    render() {
20      return (
21        <div> Secondes : {this.state.seconds} </div>
22      );
23    }
24  }
25
26  ReactDOM.render(
27    <Timer />, document.getElementById('timer-example')
28  );
```

FIGURE 7 – Code exemple React

React combine ainsi les principes de programmation déclarative et de programmation par blocs, ce qui explique notre intérêt de réutiliser les bases fondamentales de cette libraire pour notre nouveau langage PATL.

Finalement, en plus des abstractions fournies par Rendery, donner la possibilité aux programmeurs de créer leurs propres types abstraits pouvant être appliqués aux différents objets rend notre langage plus polyvalent et aide le programmeur à concevoir son application de la manière dont il l'entend. Cette possibilité vient du fait que notre nouveau langage se base sur Swift et permet donc l'utilisation des structures de ce dernier.

3.3 PATL

Dans cette partie, nous allons maintenant aborder plus en détails la façon dont un programme PATL est construit. Tout d'abord, nous expliquerons la sémantique statique de typage du langage. Cette dernière a pour but de spécifier le type de chaque élément fondamental pouvant être défini dans l'application. Ensuite, nous verrons la sémantique opérationnelle de PATL, sémantique basée sur celle de React. Elle vise à définir le comportement des différents éléments composant une application PATL et les opérations qui leurs sont applicables.

3.3.1 Sémantique statique de typage de PATL

Un programme PATL est composé d'au moins un composant haut niveau dans lequel l'état global de l'application et deux fonctions de rendu, nommées `render()` et `rerender()`, sont au minimum définies. Le programme peut ensuite être composé de composants bas niveau implémentant au minimum les deux fonctions de rendu.

En se focalisant sur la sémantique statique de typage, nous pouvons ainsi définir le fonctionnement d'une application PATL comme une fonction $h : X \to Y$, avec X l'ensemble des programmes composés d'instructions PATL et Y l'ensemble des programmes composés d'instructions Rendery, puisque chaque instruction PATL sera traduit en une ou plusieurs instructions Rendery.

Les composants de PATL peuvent être considérés comme des tuples. Il s'agit de collections ordonnées d'objets visant à représenter un élément en question.

Nous définissons un composant haut niveau comme un tuple de quatre éléments *(Name, State, RenderTop, RerenderTop)*.

— Le premier, $Name \in$ String représente le nom du composant.
— Le deuxième élément définit l'état global de l'application et nous avons $State \in V$, où V est l'ensemble regroupant des primitives telles que *Int, String, Double*, des dictionnaires ($d : String \to V$) et des types abstraits comme par exemple *Coord* ou *Angle*.
— Le troisième élément est $RenderTop \in RT$, où RT est l'ensemble des fonctions partielles $rt : V \to (String \times P \times RB \times RRB)$ avec :
  — P, l'ensemble des fonctions partielles $p : String \to V$
  — RB, l'ensemble des fonctions partielles $rb : P \to (String \times P \times RB \times RRB) \cup IR$
  — RRB, l'ensemble des fonctions partielles $rrb : P \to IR$
  — et IR l'ensemble des instructions Rendery.
— Le quatrième et dernier élément est $RerenderTop \in RRT$, l'ensemble des fonctions $rrt : V \to (String \times P \times RB \times RRB)$.

(Name, State, RenderBot, RerenderBot) est le tuple qui définit un composant bas niveau avec comme typage $(String \times P \times RB \times RRB)$.

Cette sémantique de typage statique décrit donc le fait qu'un composant haut niveau peut appeler des composants bas niveau à partir de ses fonctions `render()` et `rerender()` et leur passer les informations de l'état global de l'application. Ensuite, un composant bas niveau appelé va pouvoir lui-même appeler un autre composant bas niveau ou alors faire appel à une ou plusieurs instructions Rendery grâce à sa fonction `render()`. Enfin, sa fonction `rerender()` et les propriétés qui lui sont passées permettent d'appeler des instructions Rendery typiquement utilisées pour la mise à jour de ce composant.

3.3.2 Sémantique opérationnelle de PATL

Nous allons maintenant discuter de la sémantique opérationnelle de PATL. Cette sémantique opérationnelle est basée sur celle de React et proposée par Magnus Madsen, Ondrej Lhotak et Frank Tip dans le papier de recherche intitulé "A Semantics for the Essence of React" [17]. Dans ce papier, les auteurs définissent des termes ainsi que des règles d'inférences pour décrire les comportements d'une application React. L'idée pour le langage PATL est donc de se baser sur ces règles d'inférences définies pour React et de les réadapter à nos besoins.

Trois notions importantes sont tout d'abord nécessaires à définir. Un *Component Descriptor* noté $C(props)$ fait relation à un composant avec un nom et des propriétés. Un *Mounted Component* noté $C@a(props)$ est un composant associé à un objet en mémoire stocké à l'adresse a. Enfin, une configuration se note $(\sigma, \delta, \gamma, l, e)$ avec :

- σ représentant le composant en mémoire
- δ contenant l'état suivant du composant
- γ contenant les informations reflétant l'affichage du composant
- l représentant les événements associés au composant
- et e indiquant l'opération appliquée au composant.

A partir de là , nous pouvons à présent décrire la sémantique opérationnelle des étapes les plus importantes d'un programme PATL. Tout d'abord, concernant l'état initial du système, nous avons la configuration suivante :

$(\emptyset, \emptyset, \emptyset, \emptyset, MOUNT(\pi))$, avec $\pi = C(props)$ étant le composant haut niveau du programme. L'opération $MOUNT()$ est définie par la règle en Figure 8 et indique l'initialisation et la création en mémoire de ce composant. A partir de la configuration $(\sigma, \delta, \gamma, l, MOUNT(\pi))$, on obtient :
$(\sigma', \delta', \gamma, l', MOUNTED(C@a(props), MOUNTSEQ(RENDER(\pi))))$, avec :

- $\pi = C(props)$ le composant à "monter"
- $a \notin dom(\sigma)$ représentant une nouvelle adresse mémoire pour le composant
- σ', l'objet stocké en mémoire avec les propriétés du composant

— δ' et l', les mises à jour de l'état suivant du composant et des événements qui lui sont liés.

$$\frac{\pi = \text{C(props)} \quad a \notin dom(\sigma) \quad \sigma' = \sigma[a \rightarrow \{props\}] \quad \delta' = \delta[a] \quad l' = l[a]}{(\sigma,\ \delta,\ \gamma,\ l,\ MOUNT(\pi)) \longrightarrow (\sigma',\ \delta',\ \gamma,\ l',\ MOUNTED(\text{C}\,@a(\text{ props }),\ MOUNTSEQ(RENDER(\pi))))}$$

FIGURE 8 – Règle MOUNT()

La règle $MOUNTED()$ est décrite dans la Figure 9 et précise la mise à jour de l'affichage d'un composant et de ses sous-composants associés avec $\gamma' = \gamma[a \rightarrow (C@a(props), \bar{a})]$, où $\bar{a}$ spécifie les adresses mémoires des sous-composants de a. Finalement, $MOUNTSEQ(RENDER(\pi))$ agit comme $MOUNT()$, mais pour la séquence des sous-composants de π en appelant leur fonction `render()`.

$$\frac{\gamma' = \gamma[a \rightarrow (\text{C}\,@a(\text{ props }), \bar{a})]}{(\sigma,\ \delta,\ \gamma,\ l,\ MOUNTED(\text{C}\,@a(\text{ props }), \bar{a})) \longrightarrow (\sigma,\ \delta,\ \gamma',\ l,\ a)}$$

FIGURE 9 – Règle MOUNTED()

L'opération inverse de $MOUNT()$ est $UNMOUNT()$ et permet de "démonter" un composant, c'est-à-dire de ne plus l'impliquer dans le rendu (et non de le supprimer en mémoire). La règle est représentée en Figure 10 et présente le fait que pour "démonter" un composant a, il faut d'abord "démonter" la séquence de ses sous-composants, où les sous-composants peuvent être donnés par $\gamma(a) = (C@a(props), \bar{a})$.

$$\frac{\gamma(a) = (\text{C}\,@a(\text{ props }),\ \bar{a})}{(\sigma,\ \delta,\ \gamma,\ l,\ UNMOUNT(a)) \longrightarrow (\sigma,\ \delta,\ \gamma,\ l,\ UNMOUNTSEQ(\bar{a})\,;UNMOUNTED(a))}$$

FIGURE 10 – Règle UNMOUNT()

Finalement, $UNMOUNTED()$ a pour but de retirer tous les événements liés au composant "démonté" et est représentée en Figure 11.

$$\frac{l' = l - a}{(\sigma,\ \delta,\ \gamma,\ l,\ UNMOUNTED(\ a\ )) \longrightarrow (\sigma,\ \delta,\ \gamma,\ l',\ NIL)}$$

Figure 11 – Règle UNMOUNTED()

Concernant la mise à jour de l'état global de l'application, deux règles sont importantes à définir : *object equality* et *state merge*. La première définit l'égalité entre deux objets, typiquement utilisée entre les deux dictionnaires décrivant l'état global actuel et l'état global suivant de l'application dans un programme PATL. On compare donc les clés partagées, les valeurs et les références des types primitifs. Cette règle est illustrée en Figure 12.

$$\frac{\begin{array}{c} keys(o_1) = keys(o_2) \\ \forall k \in keys(o_1)\ \ o_1(k)\ is\ primitve \Rightarrow o_1(k) == o_2(k) \quad \forall k \in keys(o_1)\ \ o_1(k)\ is\ reference \Rightarrow o_1(k) === o_2(k) \end{array}}{o_1 \equiv o_2}$$

Figure 12 – Règle OBJECT EQUALITY

Le *state merge*, en Figure 13 va quant à lui appliquer à l'état global actuel les modifications voulues et décrites dans l'état global suivant en fonction des clés utilisées dans les deux dictionnaires.

$$\frac{\begin{array}{c} \forall k_i \in keys(o_1), o_1(k_i) = v_i \qquad \forall k_i' \in \left(keys(o_2) - keys(o_1)\right), o_2(k_i') = v_i' \\ o_3 = \{k_1 : v_1, \dots, k_n : v_n, k_1' : v_1', \dots, k_n' : v_n'\} \end{array}}{\text{MERGE}(o_1, o_2) = o_3}$$

Figure 13 – Règle STATE MERGE

Pour mettre à jour un composant "monté", il existe deux cas. Dans le premier cas, le composant actuel est remplacé par un autre composant, alors que dans le second cas, le composant "monté" reste en place mais voit ses propriétés changer avec l'état global du système. Ces deux cas sont représentés respectivement en Figure 14 et en Figure 15 et cette opération se nomme *reconciliation*.

$$\frac{\pi = C_1(\,nextprops\,) \quad \gamma(a) = (C_2\,@a(\,prevprops\,),_) \quad C_1 \neq C_2}{(\,\sigma,\ \delta,\ \gamma,\ l,\ RECONCILE(\,\pi, a\,)\,) \longrightarrow (\,\sigma,\ \delta,\ \gamma,\ l,\ UNMOUNT(a)\,;\ MOUNT(\pi)\,)}$$

Figure 14 – Règle RECONCILIATION cas n°1

$$\frac{\begin{array}{lll}\pi = C(\,nextprops\,) & \gamma(a) = (C\,@a(\,prevprops\,), \bar{a}) & nextstate = \delta(a) \\ o = \sigma(a) & o' = o[\text{props} \rightarrow nextprops]\,[\text{state} \rightarrow nextstate] & \sigma' = \sigma[a \rightarrow o']\end{array}}{(\,\sigma,\ \delta,\ \gamma,\ l,\ RECONCILE(\,\pi, a\,)\,) \longrightarrow (\,\sigma',\ \delta,\ \gamma,\ l,\ RECONCILED(C\,@a(\,nextprops\,), RECONCILESEQ(RERENDER(a), \bar{a})))}$$

Figure 15 – Règle RECONCILIATION cas n°2

Dans le premier cas, π définit le nouveau composant à "monter", $\gamma(a)$ indique l'adresse mémoire du composant à remplacer et ainsi l'ancien composant est d'abord "démonté" puis le composant π est monté à sa place. Le second cas est plus complexe : on définit π et $\gamma(a)$ de la même manière, puis on récupère les propriétés suivantes du composant et l'état global suivant pour remplacer les valeurs actuelles à l'adresse mémoire de π et ainsi mettre à jour cette adresse (σ'). Lorsqu'un composant est mis à jour, il est nécessaire de faire de même avec tous ses sous-composants, ce qui est défini par la règle $RECONCILESEQ(RERENDER(a), \bar{a})$ où le principe est d'appeler la fonction `rerender()` du composant mis à jour pour ensuite appliquer $RECONCILE()$ sur chacun des sous-composants. La règle $RECONCILED()$, Figure 16, décrit la façon de mettre à jour l'affichage du composant après sa mise à jour ainsi que celles de ses sous-composants.

$$\frac{\gamma' = \gamma[a \rightarrow (C\,@a(\,props\,), \bar{a})]}{(\,\sigma,\ \delta,\ \gamma,\ l,\ RECONCILED(C\,@a(\,props\,), \bar{a})\,) \longrightarrow (\,\sigma,\ \delta,\ \gamma',\ l,\ a\,)}$$

Figure 16 – Règle RECONCILED()

Les deux fonctions de rendu `render()` et `rerender()` sont définies en Figure 17 et 18. La différence significative entre ces deux fonctions est le fait qu'on applique la fonction `render()` lorsque c'est la première fois que l'on veut afficher le composant, alors que si ce n'est pas le cas, on passe par `rerender()` en utilisant l'adresse mémoire du composant.

$$\frac{\pi = C(\text{ props }) \quad (\sigma,\ \delta,\ \gamma,\ l,\ render(\,)\,) \longrightarrow (\sigma,\ \delta,\ \gamma,\ l,\ \bar{\bar{\pi}})}{(\sigma,\ \delta,\ \gamma,\ l,\ RENDER(\pi)\,) \longrightarrow (\sigma,\ \delta,\ \gamma,\ l,\ \bar{\bar{\pi}})}$$

FIGURE 17 – Règle RENDER()

$$\frac{\gamma(a)\ = (\text{ C } @a\ (\text{ props }),_) \quad (\sigma,\ \delta,\ \gamma,\ l,\ render(\,)\,) \longrightarrow (\sigma,\ \delta,\ \gamma,\ l,\ \bar{\bar{\pi}})}{(\sigma,\ \delta,\ \gamma,\ l,\ RERENDER(a)\,) \longrightarrow (\sigma,\ \delta,\ \gamma,\ l,\ \bar{\bar{\pi}})}$$

FIGURE 18 – Règle RERENDER()

Finalement, $UPDATESTATE()$ permet de mettre à jour l'état global de l'application en passant par la fonction $MERGE()$ décrite auparavant. Lorsque l'état global change, il est nécessaire de faire une reconciliation sur les composants affectés puisque les retours de leur fonction `render()` ainsi que ceux de leurs sous-composants peuvent changer. La règle $UPDATESTATE()$ est représentée en Figure 19.

$$\frac{\gamma(a)\ = (\text{ C } @a\ (\text{ props }),\bar{a}) \quad nextstate = \delta(a) \quad \delta' = \delta[a \rightarrow \text{MERGE}(newstate, nextstate)]}{(\sigma,\ \delta,\ \gamma,\ l,\ UPDATESTATE(a\ ,\ newstate)\,) \longrightarrow (\sigma,\ \delta',\ \gamma,\ l,\ RECONCILED(a\ ,\ RECONCILESEQ(RENDER(a),\bar{a})))}$$

FIGURE 19 – Règle UPDATESTATE()

Le papier sur lequel cette sémantique est basée, propose de montrer que le montage, le démontage et la réconciliation d'un composant se terminent. La notion de "composant qui se comporte bien", en anglais *well-behaved component*, est ainsi mise en place et définit le fait que la fonction de rendu d'un composant doit renvoyer une liste de *Component Descriptor* dont chacun est plus petit que le composant en question. Ceci permet d'éviter qu'un composant soit appelé par lui-même et donc possiblement de manière infinie impliquant des processus ne terminant pas. Les preuves liées aux propriétés relatives à la préservation du "bien-être" d'un composant sont détaillées dans un rapport technique [18] et spécifient que tant que les fonctions de rendu se terminent et qu'il existe une hiérarchie entre les composants, alors les processus de montage, démontage et réconciliation des composants terminent tous.

4 Implémentation

4.1 Rendery et React

L'essence même de PATL se base sur Rendery [11], un moteur de rendu 2D/3D écrit en Swift, développé pour la plate-forme iOS et fondé sur OpenGL. Rendery est une librairie autonome, ce qui permet de l'importer facilement dans un nouveau projet. Une fois Rendery importé dans le nouveau projet, il est possible de développer une application PATL en se basant sur les fonctionnalités justement promulguées par Rendery. Un élément important à prendre en considération est le fait que les bibliothèques OpenGL sont développées par les différents constructeurs de cartes graphiques et il existe donc différentes "versions" d'OpenGL. Sur un système iOS, la bibliothèque OpenGL est maintenue par Apple lui-même ce qui est différent sous Linux, où il existe un mélange de versions [19]. Or, le nouveau langage proposé vise à être utilisable sur la majeure partie des plate-formes et afin de pouvoir accéder sur Linux aux fonctions OpenGL utilisées par Rendery, il est nécessaire d'appeler un chargeur de fonctions OpenGL développé justement pour Linux. Un projet a été développé en 2017 dans ce but [20]. Ce chargeur de fonctions a été développé en Swift et permet de charger n'importe quelle fonction OpenGL jusqu'à la version 4.5. A partir de là, il est possible d'intégrer cette dépendance à Rendery (via le Swift Package Manager) puisque Rendery se base sur la version 3.30 d'OpenGL et donc de faire fonctionner Rendery sur Linux.

Afin de développer une application graphique avec OpenGL, il est tout d'abord nécessaire de créer un contexte OpenGL ainsi que la fenêtre de l'application pour afficher les éléments sur l'écran. Or, ces opérations se font de manière différente en fonction de l'OS (Operating System) utilisé. Pour ce faire, Rendery intègre la librairie GLFW, écrite en C. Cette librairie permet également de gérer les entrées utilisateurs ainsi que les événements.

La façon de créer une application graphique avec Rendery suit un schéma assez simple et intuitif. En effet, une fois le contexte OpenGL et la fenêtre sur laquelle celui-ci a été attaché ont été créés, il faut définir une scène dans laquelle les objets 3D seront placés. Chaque objet est ainsi défini dans la scène comme un noeud enfant possédant des propriétés telles qu'un nom par exemple.

Il faut dans un premier temps placer l'objet crucial de la scène, il s'agit de la caméra qui est, elle aussi, un noeud enfant. Celle-ci permet d'obtenir un point de vue sur la scène. Une fois les objets définis, il est possible d'effectuer des actions sur ces derniers. On peut typiquement modifier leur position en jouant sur leurs coordonnées (x,y,z), modifier leur couleur ou leur texture, modifier le type de lumière s'il s'agit d'une lumière, etc.

La particularité de PATL est le fait qu'il s'agit d'un langage complètement déclaratif. Ce dernier est largement inspiré de React, une bibliothèque Javascript donnant la possibilité de facilement créer des interfaces utilisateurs interactives. React se base sur un certains nombres de notions tels qu'un système de composants, de propriétés, d'états locaux et globaux ainsi que de fonctions de rendu et de mise à jour. Le système de composants permet de dissocier le code et de considérer chaque morceau de manière indépendante. L'intérêt est ensuite de combiner les différents composants pour obtenir un rendu plus ou moins complexe. Chaque composant possède de potentielles propriétés ainsi qu'un état local qui représente ces données spécifiques, données pouvant changer au cours du temps. Une fonction de rendu est définie dans chaque composant et son but est d'afficher les éléments relatifs à ce composant en question. Elle est appelée à chaque fois que l'affichage d'un composant doit être mis à jour. Finalement, une fonction peut être implémenter dans le but de spécifier en quelque sorte l'intervalle des mises à jour, par exemple 60 fois par seconde.

4.2 PATL

PATL vise à avoir une syntaxe simple et significative pour concevoir une application graphique dans laquelle des objets 3D doivent être définis et manipulés. A partir de l'idée générale de React, notre langage réutilise donc les principes : de composants pour déclarer des objets, de propriétés héritées entre composants, d'un état global pour l'application et de fonctions de mise à jour et de rendu. De plus, les composants possèdent une abstraction supplémentaire spécifiant s'il s'agit de composants haut niveau ou bas niveau. Le composant haut niveau de l'application est le premier à être appelé et va définir l'état global de l'application. L'idée principale ici est de séparer l'état de l'application

du rendu, c'est-à-dire de la représentation graphique. C'est dans ce composant haut niveau que toutes les modifications de l'état de l'application se font. On y déclare donc toutes les fonctions nécessaires à cela. Les composants bas niveau quant à eux ne définissent pas d'état mais récupèrent les propriétés qui leur sont passées depuis le composant haut niveau. De plus, un composant, quelque soit son niveau, possède deux fonctions de rendu : `render()` et `rerender()` dont le but est respectivement d'afficher pour la première fois un élément et de réafficher un élément déjà affiché. Dans le composant haut niveau, ces fonctions vont appeler les fonctions du même nom des composants bas niveau voulus et ces dernières vont effectuer les actions nécessaires pour l'affichage d'un objet. Finalement, la mise à jour de l'état global de l'application est effectuée par une fonction dédiée à cela et est définie dans le composant haut niveau. Celle-ci prend l'état actuel de l'application ainsi que l'état suivant voulu puis appelle la deuxième fonction de rendu du composant haut niveau pour mettre à jour l'affichage.

PATL est un EDSL basé sur le langage hôte Swift. Le choix de l'EDSL permet de ne pas avoir à recréer un compilateur mais, au contraire, de passer par celui du langage hôte et en l'occurrence celui de Swift. Le langage Swift est un langage relativement jeune et ayant une documentation très fournie. Il s'agit du langage parfait pour mettre en place une syntaxe simple et intuitive. De plus, le désavantage principal d'un EDSL qui concerne les sacrifices à faire au niveau des mots-clés du langage est ici très atténué du fait que la syntaxe Swift est de base très épurée. De ce fait, les composants introduits dans ce nouveau langage peuvent de manière générale être déclarés sous forme de classe et les composants bas niveau comme des sous-classes du composant haut niveau. L'état global de l'application prend la forme d'un dictionnaire dans lequel chaque objet de l'application est représenté avec ses propriétés. De plus, il est facilement possible de créer des abstractions sur les types avec les structures Swift.

Pour le développement du langage, une librairie standard a été créée ayant pour but de regrouper toutes les fonctions, les structures et les éléments principaux permettant à un utilisateur de développer son application graphique.

Le fonctionnement d'une application PATL est représenté en Figure 20. L'idée est d'écrire le code de l'application dans un `main.swift` et d'y importer la librairie standard du langage. Lorsqu'une méthode de la librairie est appelée, celle-ci va elle-même appeler une ou plusieurs méthodes Rendery dans le but de retourner des valeurs ou d'obtenir un rendu graphique.

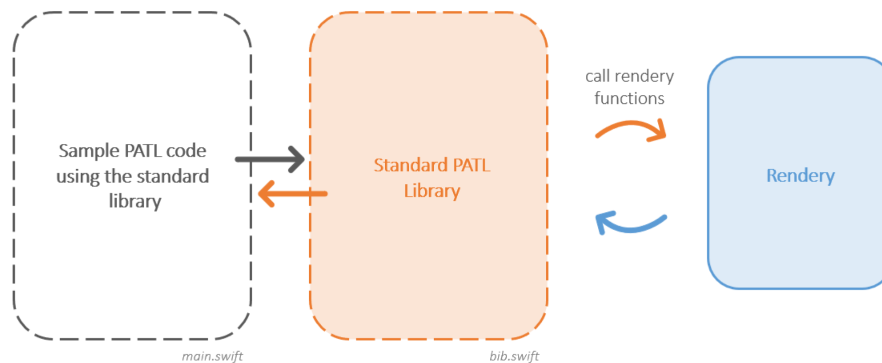

FIGURE 20 – Schéma de l'organisation d'une application PATL

La librairie standard peut être divisée en deux parties dont la première regroupe des fonctions utiles à la création de l'application en soi tandis que la deuxième définit des fonctions pour la création, la modification et l'affichage des objets 3D.

Tout d'abord, lors de la création d'une application, la première fonction à devoir être appelée est celle permettant de mettre en place tous les éléments essentiels et définit ainsi le point d'entrée du programme. Cette fonction permet d'initialiser la fenêtre et le contexte de l'application, de définir la scène de cette dernière (c'est-à-dire le composant haut niveau de l'application) ainsi que la boucle de rendu en spécifiant le nombre de *frame* par seconde. Ensuite, la mise à jour de l'état global de l'application peut être effectuée à l'aide d'une fonction qui, à partir de l'état actuel, nous donne l'état suivant selon les modifications voulues. Il est possible d'exécuter une partie du code de rendu de manière continue dont l'idée en général est de mettre à jour l'état global du système ainsi que d'appeler les fonctions de rendu des composants souhaités dans le but d'appliquer et d'afficher toutes les modifications voulues.

D'autres fonctions permettent entre autre de gérer les événements liés aux entrées utilisateurs tels que l'appui d'une touche du clavier ou de la souris, d'importer des modèles d'objets 3D (fichiers .gltf) avec l'*importer* GLTF de Rendery ou d'importer des *shaders* personnalisés afin d'appliquer des effets visuels à partir de programmes GLSL.

Les fonctions pour la création et la manipulation des objets sont définies dans la classe `ComponentTopLevel` et donc utilisables par le composant haut niveau de l'application ainsi que par les composants bas niveau dans lesquels la scène où les objets doivent être placés/modifiés est précisée. Une fonction permet de créer une caméra donnant un point de vue sur la scène pour l'utilisateur. Différents paramètres peuvent être modifiés tels que sa position, la distance pouvant être perçue ou alors des contraintes permettant par exemple d'éviter de nouveaux calculs d'angle si la position de la caméra est modifiée. Selon le même principe, il est possible de placer une source de lumière dans la scène. L'idée principale pour créer un objet dans une scène est tout d'abord de voir cette dernière comme la racine (*root*) et que chaque objet prenant place dans la scène sera un noeud possédant des caractéristiques diverses. Une caméra ou une source de lumière sont donc de la même manière des noeuds enfants de la scène. La libraire implémente donc une fonction permettant de créer des noeuds et de spécifier leur nom afin de pouvoir les manipuler dans d'autres fonctions. Ces autres fonctions peuvent typiquement : (1) définir un noeud comme un objet géométrique, (2) appliquer une texture sur un noeud ou alors (3) modifier la position d'un noeud.

(1) Différentes fonctions donnent la possibilité de faire d'un noeud, un objet géométrique 3D tel que par exemple, une sphère, un cube ou un rectangle. Sur ces noeuds, une des propriétés spécifie le modèle de l'objet, qui est lui-même défini par un *mesh* ainsi que par un *material.* Typiquement, pour le cas d'une sphère, le *mesh* est prédéfini dans Rendery et possède certaines propriétés telles que le nombre de points (pour obtenir un rendu plus ou moins lisse et plus ou moins coûteux) ou alors le rayon. Le *material* précise, quant à lui, la couleur de l'objet ou la texture qui lui est appliquée.

(2) Concernant les textures des objets, des fonctions sont définies dans le but d'en appliquer ou de les modifier. Il est possible d'appliquer ces textures aux objets via différents moyens tels qu'à l'aide d'une image, d'une couleur ou d'un programme GLSL (shader).

(3) Finalement, il est possible de modifier la position d'un noeud en lui attribuant de nouvelles coordonnées. De ce fait nous pouvons obtenir un rendu graphique lisse avec des objets en mouvement en appelant cette fonction à chaque *frame* et en passant par les fonctions de rendu.

Pour le moment, seule une partie des fonctions de Rendery est implémentée à travers la librairie standard de PATL. Cette librairie aura pour but d'être, par la suite la plus complète possible. La Table 2 dans la section Annexes liste les fonctions les plus importantes de la librairie et donne une description succincte pour chacune d'entre elles.

Avec la librairie, un certains nombres de types abstraits sont fournis, soit hérités de Rendery tels que `Color` ou encore `Angle`, soit créés directement tel que par exemple `Coord`. Ces abstractions sur les types ont pour but de simplifier la création et la manipulation des objets. Les types prédéfinis visent ainsi à être le plus complet possible pour représenter les caractéristiques essentielles d'un objet 3D : sa position, sa couleur, etc. A titre d'exemple, le type `Coord` permet de spécifier une position selon deux différents systèmes de coordonnées : en coordonnées cartésiennes 3D (x,y,z) ou en coordonnées polaires/sphériques (ρ, θ, ϕ) et il est possible de passer d'un système à l'autre avec une simple fonction de conversion. Les utilisateurs ont aussi la possibilité de créer leur propres types abstraits en passant par le système de structure de Swift.

Pour créer une application PATL, il est donc nécessaire d'importer sa librairie standard afin d'accéder aux méthodes et aux abstractions de type fournis. Certaines dépendances nécessaires sont à installer et dépendent de l'OS (Operating System) utilisé. Ces dépendances ont déjà été évoquées auparavant et concernent notamment les chargeurs de fonctions OpenGL et les différentes librairies à importer.

4.3 Cas pratique : Système Solaire

Un exemple de cas pratique a été développé lors de ce travail et porte sur le système solaire. Dans cette application, l'objectif principal est de représenter le soleil ainsi que les différentes planètes du système solaire (Mercure, Vénus, Terre, Mars, Jupiter, Saturne, Uranus et Neptune) et d'observer leur déplacement au fil du temps. Chacune d'entre elles possède un rayon, une position, une période de révolution et une texture unique. Ces informations sont représentées dans l'état global du système, lui-même déclaré dans le composant haut niveau `SystemSolar`.

Dans cet exemple d'implémentation, différentes fonctions permettent d'obtenir une mise à l'échelle réaliste des tailles des planètes et des distances qui les séparent. La première fonction de rendu du composant est ensuite appelée, dans laquelle la caméra de l'application est créée ainsi que les différentes planètes représentées par des sphères. Ces dernières sont créées en appelant de même la première fonction de rendu du composant bas niveau `Sphere` et en leur passant tous les paramètres nécessaires contenus dans l'état global (nom, scène, position, propriétés). Dans cette fonction de rendu du composant `Sphere`, comme évoqué auparavant, la construction de l'objet passe d'abord par la création d'un noeud enfant puis, en lui appliquant le *mesh* prédéfini "sphere", et enfin, en ajoutant à l'objet une texture et en le plaçant à une certaine position.

Ensuite, dans le composant `SystemSolar`, une fonction est définie dans le but de mettre à jour l'état global du système en modifiant uniquement la position des planètes dans l'état suivant puisque le but est d'observer la trajectoire de ces dernières. Il a été choisi de représenter la position des planètes avec des coordonnées polaires en passant par l'abstraction de type `Coord`. L'idée est ainsi de recalculer le nouvel angle de chaque planète lors de chaque mise à jour. Un appel à la deuxième fonction de rendu du composant haut niveau et donc également à celles des composants bas niveau concernés est effectué lors de la mise à jour dans le but de refléter l'affichage correspondant à la modification de l'état.

Les fonctions permettant de mettre à jour l'état global du système ainsi que l'affichage sont appelées en permanence en fonction du nombre de *frame* par seconde défini pour l'application.

Finalement, le point d'entrée du programme est donné en spécifiant le composant haut niveau de l'application, c'est-à-dire `systemSolar`.

Une image du rendu de cette application est donnée en Figure 21 dans laquelle nous pouvons bien observer au centre le soleil et gravitant autour, les différentes planètes.

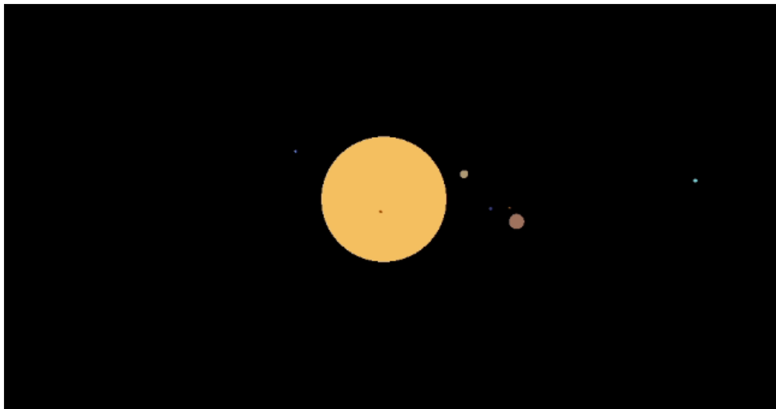

FIGURE 21 – Rendu du Système solaire avec une application PATL

4.4 Didacticiel

Pour la création d'applications avec le langage PATL, une documentation sous forme de didacticiel a été créée. Ce tutoriel explique comment faire pour mettre en place une application et quels sont les éléments à intégrer pour cela. Les différentes parties expliquent ensuite de manière intuitive comment manipuler les éléments importants d'un programme tels que par exemple, la caméra, les sources de lumières, les objets 3D, etc. Une présentation du code source avec une explication de celui-ci permet de comprendre facilement son fonctionnement. De plus, un rendu graphique est associé à chaque élément expliqué dans le but de pouvoir visualiser le résultat. Ce didacticiel est disponible sur le Github du projet [4]. La Figure 22 présente l'accueil et les différentes rubriques de ce dernier.

Figure 22 – Didacticiel PATL

La Figure 23 montre quant à elle, la partie du didacticiel sur l'explication concernant la création d'une caméra pour visualiser la scène ainsi que le rendu obtenu.

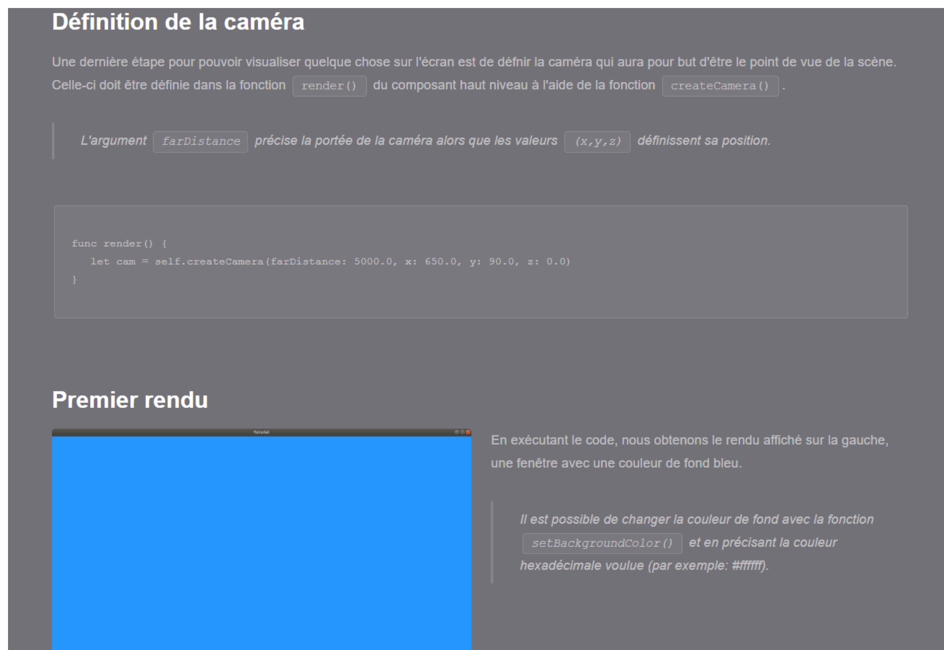

Figure 23 – Création d'une caméra dans un programme PATL

5 Evaluation

Le langage PATL a pour objectif de permettre la création d'applications graphiques, de manière simple et efficace.

Le fait de nous baser sur le langage hôte Swift est un choix qui va dans ce sens puisque ce dernier possède une documentation extrêmement bien détaillée et implique donc une certaine facilité pour la programmation. De plus, le didacticiel mis en place pour notre langage PATL a pour but de promouvoir une meilleure prise en main de celui-ci grâce aux différentes explications données ainsi qu'aux représentations graphiques pour des codes exemples. Cette documentation permet de comprendre comment manipuler les éléments les plus importants du langage et vise à faire apprendre la façon de créer une application graphique en PATL.

Le système de composants et la façon de programmer par blocs inspiré de React et développé à l'aide de classes Swift, permet de structurer le code de manière efficace. La représentation de l'état global de l'application sous la forme d'un dictionnaire permet de regrouper les différentes informations utiles à l'application à un seul et même endroit du code et implique donc par la suite la possibilité d'effectuer des mises à jour optimisées seulement pour les composants nécessaires.

Créer une application avec ce nouveau langage proposé évite de manipuler les fonctions OpenGL en elles-mêmes ainsi que de préciser la façon dont les données doivent être interprétées. De simples fonctions permettent de mettre en place l'application et de construire des objets 3D dans une scène tridimensionnelle. Ces fonctions réutilisent les abstractions promulguées par Rendery, le moteur de rendu graphique à la base de PATL.

La particularité de ce dernier est le fait qu'il s'agisse d'un langage complètement déclaratif. Ceci implique de ce fait un changement fondamental sur la manière de développer une application graphique. Cette notion déclarative met en avant la division du code en plusieurs parties et améliore de ce fait sa clarté.

En prenant l'exemple proposé du système solaire, le fait de séparer le composant haut niveau des composants bas niveau de l'application permet de bien visualiser l'idée que l'information est gérée et manipulée à un seul endroit (composant haut niveau) et qu'elle est envoyée aux composants bas niveau dont leur but est de représenter les éléments graphiques.

Cependant, le langage a été testé pour le moment uniquement au travers d'implémentation de cas pratiques. Il serait donc intéressant de pouvoir faire tester ce langage à différents types de programmeur, aussi bien à des experts qu'à des novices de la programmation graphique, afin d'observer son efficacité. De plus, bien que ce langage vise à faciliter le développement d'applications graphiques, il est possible en contrepartie, qu'il implique une "perte de liberté" pour le programmeur par rapport aux langages de programmation graphique actuels. Finalement, il pourrait être intéressant d'essayer de développer un jeu plus ou moins complexe plutôt que de petites applications afin de pouvoir nous rendre compte si notre langage apporte toutes les facilités voulus.

6 Conclusion et travaux futurs

A travers ce projet, nous avons étudié une comparaison entre OpenGL et DirectX, deux APIs bas niveau pour la programmation graphique. Nous avons présenté Gator, un langage visant à introduire un nouveau type pour éviter les bogues de géométrie dans les *shaders*. Après cela, nous avons regardé les outils pour la programmation graphique proposés par le langage Swift et avons pu en déduire qu'ils ne permettent pas de développer de manière efficace des applications graphiques.

Dans l'idée de créer un nouveau langage facilitant le développement de programmes graphiques, nous nous sommes focalisés sur le paradigme de programmation déclaratif et sur le principe de la décomposition du code par blocs. Ceci nous permet ainsi d'introduire une certaine clarté dans le code des applications, de produire des programmes synthétiques et de pouvoir manipuler les différents éléments de manière individuelle ainsi que de les combiner afin d'obtenir des résultats complexes. De plus, nous avons voulu par le biais de ce langage augmenter le niveau d'abstraction concernant la création d'une application graphique ainsi que l'initialisation et la manipulation des objets 3D.

Nous nous sommes ainsi penchés sur React et sur ses notions fondamentales qui sont la programmation déclarative et le système de composants. Nous avons aussi basé notre approche sur Rendery, un moteur de rendu écrit en Swift, dans le but de réutiliser ses abstractions pour la programmation graphique.

Le langage résultant se nomme PATL. Ce dernier est complètement déclaratif et propose les fonctionnalités évoquées auparavant. PATL vise globalement à faciliter la conception d'applications graphiques et l'idée principale du langage repose sur une librairie standard promulguant des fonctionnalités basées sur celles de Rendery.

Des cas pratiques ont été implémentés tel que par exemple sur la représentation du système solaire et nous donnent de bons rendus graphiques. La librairie standard du langage reste en cours de développement et aura pour but d'être à terme la plus complète possible. Il en est de même pour le didacticiel visant à expliquer toutes les fonctionnalités du langage.

Nous nous sommes concentrés dans ce travail sur la façon d'implémenter des applications graphiques et il semble intéressant pour un travail futur de se focaliser sur l'implémentation des *shaders* dans le but d'une part, de simplifier leur conception et d'autre part, d'éviter le processus répétitif de cette tâche.

Remerciements

Je souhaite remercier Didier Buchs, Dimitri Racordon, Damien Morard et Aurélien Coet pour leur aide précieuse, leurs conseils ainsi que leur patience.

Références

[1] Randi Rost John Kessenich, Dave Baldwin. *The OpenGL ES Shading Language*. John Kessenich, Google, 2017.

[2] Hlsl documentation. https ://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl.

[3] Pipeline graphique. https ://en.wikipedia.org/wiki/Graphics_pipeline.

[4] Patl. https ://github.com/sardinhapatrick/Msc_Project.

[5] Karl Hillesland. Opengl and directx. In *SIGGRAPH Asia 2013 Courses*, SA '13, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Dietrich Geisler, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson. Geometry types for graphics programming. *Proceedings of the ACM on Programming Languages*, 4 :1 – 25, 2020.

[7] Shadertoy beta. https ://www.shadertoy.com/.

[8] Site geeks3d. https ://www.geeks3d.com/shader-library/.

[9] Documentation spritekit. https ://developer.apple.com/spritekit/.

[10] Documention scenekit. https ://developer.apple.com/documentation/scenekit/.

[11] Rendery. https ://github.com/RenderyEngine/Rendery.

[12] Tomaž Kosar, Pablo Martínez López, Pablo Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information & Software Technology*, 50 :390–405, 04 2008.

[13] Jesús Sánchez Cuadrado, Javier Cánovas, and Jesus Garcia Molina. Comparison between internal and external DSLs via RubyTL and Gra2MoL. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages : Recent Developments*. IGI Global, September 2012.

[14] Guillaume Huard. Paradigmes de programmation.

[15] Définition langages déclaratifs. https ://en.wikipedia.org/wiki/Declarative_programming.

[16] Définition langages impératifs. https ://en.wikipedia.org/wiki/Imperative_programming.

[17] Magnus Madsen, Ondrej Lhotak, and Frank Tip. A semantics for the essence of react. *European Conference on Object-Oriented Programming.*

[18] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A Semantics for the Essence of React. Technical Report CS-2020-03, University of Waterloo, 2020. URL : https ://cs.uwaterloo.ca/sites/ ca.computer-science/files/uploads/files/cs-2020-03.pdf.

[19] Site web learn opengl. https ://learnopengl.com/.

[20] Swift opengl loader. https ://github.com/kelvin13/swift-opengl.

Annexes

La Figure 24 ci-dessous, représente un exemple d'application Rendery. Tout d'abord, la fonction `sampleScene()` est appelée dans le but de créer la scène dans laquelle les objets seront placés ainsi que de définir la boucle de rendu avec le nombre d'images par seconde. La scène est ensuite définie comme une classe où l'on peut modifier certaines propriétés, comme par exemple la couleur du fond de la scène. Un noeud enfant est ensuite créé dans le but de représenter une sphère. Il possède ainsi un modèle, construit comme un *mesh* et un *material* : le *mesh* est défini comme une sphère (un type de *mesh* prédéfini dans Rendery) et le *material* est quant à lui défini comme celui par défaut. La couleur de l'objet est ensuite modifiée vers une teinte rouge et sa position dans l'espace est initialisée. Finalement, la caméra est définie en spécifiant sa position, la zone pouvant être observée (ligne 20) et les contraintes qui lui sont associées (ligne 22).

```
1   class SphereExample: Scene {
2       override init()
3           super.init()
4
5           backgroundColor = "#000000"
6
7           let sphereNode = root.createChild()
8           sphereNode.name = "Sphere"
9           sphereNode.model = Model(
10              meshes: [.sphere(segments: 100, rings: 100, radius:
                     50.0)],
11              materials: [Material()])
12          sphereNode.model?.materials[0].multiply = .color(Color(
                red: 0.5, green: 0.5, blue: 0.5, alpha: 0.5))
13          sphereNode.translation.x = 0.0
14          sphereNode.translation.y = 0.0
15          sphereNode.translation.z = 0.0
16
17          let cameraNode = root.createChild()
18          cameraNode.name = "Camera"
19          cameraNode.camera = Camera()
20          cameraNode.camera?.farDistance = 500.0
21          cameraNode.translation = Vector3(x: 100.0, y: 10, z:
                0.0)
22          cameraNode.constraints.append(LookAtConstraint(target:
                root))
23  }
24
25  func sampleScene() {
26      guard let window = AppContext.shared.initialize(width:
            1500, height: 800, title: "Example")
27      else { fatalError() }
28      let scene = sphereExemple()
29      window.viewports.first?.present(scene: scene)
30      AppContext.shared.targetFrameRate = 60
31      AppContext.shared.render()
32  }
```

FIGURE 24 – Code exemple Rendery

Nom de la fonction	Description
`createChildNode()`	Créée un noeud enfant dans la scène spécifiée, pouvant par la suite être utilisé pour la création d'un objet 3D, d'une source de lumière, etc.
`createCamera()`	Créée un noeud enfant dans la scène spécifiée et l'interprète comme une caméra ayant une position, une portée et des contraintes.
`createLight()`	Créée un noeud enfant dans la scène spécifiée et l'interprète comme une source de lumière avec une position, une direction et des contraintes.
`applyTextureFromImg()`	Applique une texture sur un noeud de la scène (une sphère par exemple). Les fonctions `applyTextureFromShaders()` et `applyTextureFromColor()` agissent de la même façon avec respectivement un programme GLSL et une couleur hexadécimale.
`createSphere()`	Définit un noeud comme un objet Sphère avec des propriétés tels que le rayon, etc. Les fonctions `createBox()` et `createRectangle()` ont des comportements similiaires pour d'autres types d'objets.
`setNodePosition()`	Permet de modifier la position d'un noeud d'une scène. La position est donnée en coordonnées cartésiennes 3D (x,y,z).
`loadGLTF()`	Permet de charger le modèle 3D d'un objet pouvant ensuite être intégré à une scène.
`importCustomShader()`	Permet d'importer des programmes GLSL personnalisés.
`registerTick()`	Exécute le code passé en argument de manière continue.
`getKeyEvent()`	Récupère les entrées utilisateurs liées au clavier. La fonction `getMouseEvent()` fonctionne de la même manière pour la souris.
`updateState()`	Met à jour l'état global de l'application en fonction de l'état courant et de l'état suivant passés en arguments.
`createScene()`	Initialise la fenêtre de l'application, le contexte ainsi que la scène. Cette fonction met aussi en place la boucle de rendu et spécifie le nombre d'images par seconde.

Table 2 – Liste des principales fonctions PATL