Geneva University

Master Thesis

Computer Science department

Working With Graphics Programming

Patrick Sardinha

Year 2020 - 2021

Table des matières

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. Concevoir ce type d'applications est assez compliqué et demande de bonnes connaissances de programmation. L'utilisation d'un langage de programmation haut niveau incluant des abstractions sur les types est un moyen de faciliter la conception de ces applications. A l'heure actuelle, la spécification OpenGL est l'une des plus utilisées pour la création d'applications 3D 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. Le but de ce travail est de créer un nouveau langage nommé PATL. Il s'agit d'un *Domain-Specific Language* (DSL) orienté Swift inspiré d'un langage complètement déclaratif : React. PATL est un langage de programmation haut niveau, permettant la création et l'utilisation de types abstraits. Il agit ainsi comme une sur-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 permet donc de 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 ; OpenGL ; Swift ; Rendery ; 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 API, 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. Les bibliothèques proposées sont basées sur différents langages de programmation tels que le C pour OpenGL et le C++ pour DirectX. Ces API utilisent chacune leur propre langage de programmation de *Shader* [7] qui sont respectivement GLSL (OpenGL Shading Language) [8], HLSL (DirectX High-Level Shader Language) [9] 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 API 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 [10] 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 données tels que vecteurs ou matrices. Dans la littérature, il n'existe pas, à notre connaissance, de proposition de langage haut niveau avec la possibilité d'introduire des abstractions de types pour la programmation graphique.

Ce travail propose ainsi de créer un nouveau langage, nommé PATL. PATL [11] est 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 biblio-

thèque JavaScript pour créer des interfaces utilisateurs. L'idée ici est d'utiliser le principe déclaratif de React ainsi que la notion de composant à état. Les différents composants de l'application peuvent ensuite être combinés 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. 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. Le langage PATL permet ainsi de développer des applications graphiques en Swift et de manière simple, sans se soucier de la spécification d'OpenGL. Le système de structure de Swift permet de créer des abstractions sur les types et de faciliter le développement. PATL va agir comme une sur-couche déclarative à Rendery et utiliser les principes fondamentaux de React.

Nous évaluons expérimentalement notre langage PATL avec différents cas pratiques d'implémentation. Un tutoriel est développé et accessible fournissant une première documentation du langage. PATL facilite la conception d'applications graphiques surtout pour des programmeurs ayant une faible expérience dans ce domaine.

La structure du papier va suivre le plan suivant : dans un premier temps, dans la section 2, nous clarifierons les points techniques importants. Dans un second temps, nous aborderons dans la section 3, les travaux en lien avec notre sujet. Par la suite, nous verrons dans la section 4, la méthodologie proposée puis l'implémentation de celle-ci dans la section 5. Ensuite, nous étudierons un cas d'implémentation, et dans la section 6, nous évaluerons la méthode proposée. Dans un dernier temps, dans la section 7, nous évoquerons les conclusions de ce travail et de possibles travaux futurs.

2 Contexte

Dans cette partie, nous allons présenter plus en détails les éléments importants évoqués dans la partie précédente.

2.1 OpenGL

Pour créer une application graphique avec OpenGL, il faut tout d'abord créer un contexte OpenGL ainsi que la fenêtre dans laquelle la scène sera affichée. Pour se faire, il existe différentes bibliothèques proposant les outils nécessaires telles que par exemple GLUT ou GLFW. Dès que la version utilisée d'OpenGL a été spécifiée et que le contexte a été correctement mis en place, il faut créer un objet fenêtre et préciser ses dimensions ainsi que son nom. Le contexte de l'application sera ainsi le contexte de la fenêtre. Après avoir spécifié à OpenGL que le rendu doit se faire dans cette fenêtre, il faut créer la boucle de rendu. Cette boucle permet ainsi d'afficher de manière continue des images sur la 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. Lorsque la boucle de rendu se finit, l'application se ferme. Ensuite, afin d'afficher des objets, il est nécessaire de fournir des données d'entrées à la première étape du pipeline graphique puis de configurer la façon dont ces données doivent être interprétées.

2.2 Pipeline Graphique

Le pipeline graphique, aussi appelé pipeline 3D, est le processus de transformation de coordonnées 3D en pixels 2D. Ce processus est au centre même de toutes les API 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 se 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. 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 [12]. 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 trouve une étape optionnelle, le *geometry shader*. Cette étape ne sera pas traitée dans ce papier. 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 [13].

2.3 Programmation déclarative et React

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" [14].

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." [15].

React est 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 un rendu plus complexe, 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 et facilitant le débogage [16]. Un composant React est défini sous forme de classe implémentant au minimum une méthode `render()` 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 tick() est habituellement implémentée pour mettre à jour l'état d'un composant de manière régulière. Finalement, la fonction principale ReactDOM.render() permet d'appeler le premier composant de l'application. Ci-dessous, en Figure 1, se trouve un exemple d'application React pour un *Timer* affichant toutes les secondes la nouvelle valeur [16].

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }
  tick() {
    this.setState(state => ({seconds: state.seconds+1}));
  }
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }
  componentWillUnmount() {
    clearInterval(this.interval);
  }
  render() {
    return (
      <div> Secondes : {this.state.seconds} </div>
    );
  }
}
ReactDOM.render(
  <Timer />, document.getElementById('timer-example')
);
```

FIGURE 1 – Code exemple React

3 Travaux connexes

3.1 OpenGL et DirectX

OpenGL est l'une des API 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 API.

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 API [17]. 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 API peuvent intégrer des *frameworks* pour faciliter leur utilisation. Il en existe de nombreux pour chacune de ces deux API 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 API bas niveau obligeant ainsi les programmeurs à manipuler des objets, configurer des structures et implémenter des algorithmes de rendu avec précision.

3.2 Gator

Comme évoqué auparavant, ces deux API 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 [18] 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.

3.3 Librairies de *Shaders*

Pour atténuer le problème de la tâche répétitive concernant le développement des *shaders*, différents outils ont été créés tels que par exemple, Shadertoy BETA [19] ou Geeks3D [20] 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.

3.4 Programmation graphique avec Swift

Le langage Swift [21] 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 [22] des langages de programmation les plus utilisés. Apple a créé différents outils pour la programmation graphique dont SpriteKit [23], pour la création de jeux 2D et propose une classe d'objets SKShader [24] afin de créer des *fragments shaders* personnalisés. Similairement, il existe un autre outil nommé SceneKit [25] 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. 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.

3.5 Rendery

Une proposition pour palier au problème décrit précédemment est Rendery [26], 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.

3.6 *Domain-Specific Language* et langages déclaratifs

Les langages dédiés, *Domain-Specific Language* en anglais ou DSL, "sont des langages de programmation dont les spécifications sont conçues pour répondre aux contraintes d'un domaine d'application précis" [27]. L'utilisation d'un DSL offre certains avantages mais implique également des désavantages. De manière générale, un DSL va permettre un gain considérable en termes de productivité et va pouvoir être réutilisé 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 [28]. Il existe différents types de DSL, les DSL internes aussi appelés *Embedded DSL* (EDSL) et les DSL externes. Un DSL externe est construit à l'aide d'un *parser*, reconnaissant la syntaxe du nouveau langage alors qu'un DSL interne est construit à partir d'un langage hôte, réutilisant son infrastructure [29]. Un EDSL sera donc plus limité au niveau de la syntaxe et certains sacrifices seront nécessaires à faire en fonction du langage hôte mais en contrepartie cela permettra de ne pas tout ré-implémenter et de partir sur de solides fondations. Les langages déclaratifs sont, quant à eux, des langages dans lesquels nous définissons le problème et non pas la solution du problème [14]. Selon une étude comparant les paradigmes de programmation impératif et déclaratif [30], 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, 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. 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.

3.7 PATL

Créer un EDSL basé sur le langage Swift, orienté vers le paradigme de programmation déclaratif 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. Ce nouveau langage est le PATL. Nous listons en Table 1, les différentes fonctionnalités des langages de programmation d'applications graphiques étudiés dans cette partie.

	OpenGL	DirectX	Swift	Rendery	PATL
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

TABLE 1 – Comparaison de différents langages de programmation d'applications graphiques

4 Modèle

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

4.1 Un langage multiplateforme

L'essence même de PATL se base sur Rendery [26], 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 la librairie standard de PATL, 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 [13]. Or, le langage PATL vise à être utilisable sur la majeure partie des plate-formes et notamment sur Linux. Afin d'utiliser sur Linux les 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 le but de créer un chargeur de fonctions permettant d'appeler des fonctions OpenGL et fonctionnant sur Linux [31]. 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. De ce fait, avec quelques modifications du code, Rendery peut fonctionner sur Linux.

Afin de développer une application graphique avec OpenGL, il est tout d'abord nécessaire de créer tout d'abord 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.

4.2 Application graphique avec Rendery

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, ou alors modifier le type de lumière s'il s'agit d'une lumière, etc.

Un exemple d'une application Rendery est présenté en Figure 18. 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 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.

4.3 Inspiration de React

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. La fonction de rendu est la fonction `render()`. Celle-ci 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 `tick()` permet de spécifier en quelque sorte l'intervalle des mises à jour (par exemple 60 fois par seconde).

PATL vise à avoir une syntaxe simple et significative pour définir et manipuler des objets 3D. A partir de l'idée générale de React, PATL 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. Dans PATL, 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 (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, dans PATL, 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 ré-afficher un élément déjà affiché. Les fonctions `render()` et `rerender()` du composant haut niveau vont appeler les fonctions du même nom des composants bas niveau 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 la fonction `updateState()` 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 fonction `rerender()` du composant haut niveau pour mettre à jour l'affichage.

4.4 Un EDSL basé Swift

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 PATL 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.

4.5 PATL

Le fonctionnement de PATL est représenté en Figure 2. L'idée est d'écrire le code de l'application PATL dans un `main.swift` et d'y importer la librairie standard de PATL. Cette librairie fournit les méthodes essentielles pour créer une application graphique ainsi qu'un certains nombres de types abstraits prédéfinis tels que par exemple pour les Coordonnées (cartésiennes 3D ou polaires) ou les Angles (degrés ou radians). 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.

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 pour chacun des éléments. Ensuite, nous verrons la sémantique opérationnelle de PATL, sémantique basée sur celle de React.

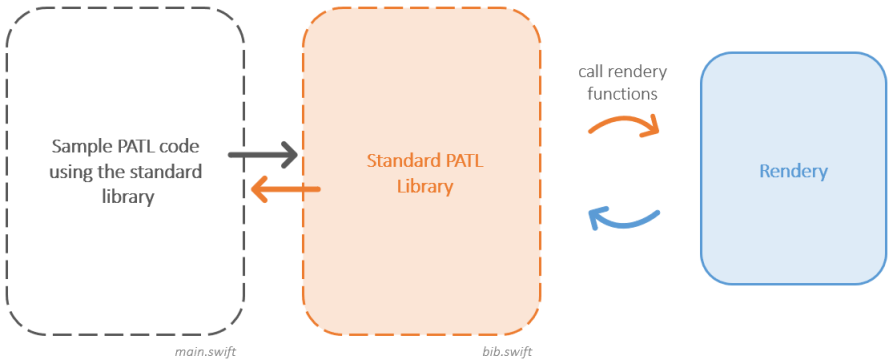

FIGURE 2 – Schéma de l'organisation d'une application PATL

4.5.1 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 les deux fonctions de rendu, `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 haut niveau de PATL peuvent être considérés comme des tuples de 4 éléments *(Name, State, RenderTop, RerenderTop)*. Le premier élément, *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*, etc., des dictionnaires ($d : String \to V$) et des types abstraits comme par exemple *Coord, Angle*, etc. 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, Reren-*

derBot) 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.

4.5.2 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" [32]. Dans ce papier, les auteurs définissent des termes ainsi que des règles d'inférence 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 3 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 puis δ' et l', les mises à jour de l'état suivant du composant et des événements qui lui sont liés.

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

FIGURE 3 – Règle MOUNT()

La règle $MOUNTED()$ est décrite dans la Figure 4 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()`.

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

FIGURE 4 – 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 5 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 $\gamma(a) = (C@a(props), \bar{a})$. Finalement, $UNMOUNTED()$ a pour but de retirer tous les événements liés au composant "démonté" et est représentée en Figure 6.

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

FIGURE 5 – Règle UNMOUNT()

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

FIGURE 6 – 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 7.

$$\frac{keys(o_1) = keys(o_2)}{\forall k \in keys(o_1)\ \ o_1(k)\ is\ primitve \Rightarrow o_1(k) == o_2(k) \qquad \forall k \in keys(o_1)\ \ o_1(k)\ is\ reference \Rightarrow o_1(k) === o_2(k)}$$
$$o_1 \equiv o_2$$

FIGURE 7 – Règle OBJECT EQUALITY

Le *state merge*, en Figure 8 va quant à lui appliquer à l'état global actuel les modifications voulu et décrites dans l'état global suivant en fonction des clés utilisées dans les deux dictionnaires.

$$\frac{\forall k_i \in keys(o_1), o_1(k_i) = v_i \qquad \forall k_i' \in \big(keys(o_2) - keys(o_1)\big), o_2(k_i') = v_i' \qquad o_3 = \{k_1 : v_1, \dots, k_n : v_n, k_1' : v_1', \dots, k_n' : v_n'\}}{\mathsf{MERGE}(o_1, o_2) = o_3}$$

FIGURE 8 – 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 9 et Figure 10 et cette opération se nomme *reconciliation*.

$$\frac{\pi = \mathtt{C}_1(\,nextprops\,) \quad \gamma(a) = (\mathtt{C}_2\,@a(\,prevprops\,), \_) \quad C_1 \neq C_2}{(\,\sigma,\ \delta,\ \gamma,\ l,\ RECONCILE(\,\pi, a\,)\,) \longrightarrow (\,\sigma,\ \delta,\ \gamma,\ l,\ UNMOUNT(a)\,;\ MOUNT(\pi)\,)}$$

FIGURE 9 – Règle RECONCILIATION cas n°1

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

FIGURE 10 – 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 11, décrit la façon de mettre à jour l'affichage du composant après sa mise à jour ainsi que celles de ses sous-composants.

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

FIGURE 11 – Règle RECONCILED()

Les deux fonctions de rendu `render()` et `rerender()` sont définies en Figure 12 et 13. 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.

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

FIGURE 12 – Règle RENDER()

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

FIGURE 13 – 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 14.

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

FIGURE 14 – Règle UPDATESTATE()

5 Implémentation

Pour le développement du langage PATL, 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.

Cette 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.

5.1 Application PATL

Tout d'abord, lors de la création d'une application PATL, la première fonction à devoir être appelée est `createScene()` 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, concernant la mise à jour de l'état global de l'application, nous avons la fonction `updateState()` qui, à partir de l'état actuel, nous donne l'état suivant en fonction des modifications voulues. La fonction `registerTick()` permet, quant à elle, d'exécuter une partie du code de rendu et de manière continue. Dans cette fonction, le but en général est de mettre à jour l'état global du système ainsi que d'appeler les fonctions `rerender()` 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 avec la fonction `importCustomShader()` dans le but 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. La fonction `createCamera()` 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, la fonction `createLight()` permet de placer une source de lumière dans la scène. La fonction indispensable pour créer des objets 3D dans une scène est `createChildNode()`. 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 de la même manière des noeuds enfants de la scène. `createChildNode()` permet donc 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) Les fonctions `createSphere()`, `createBox()` et `createRectangle()` donnent la possibilité de faire d'un noeud, un objet géométrique 3D. Sur ces noeuds, une des propriétés spécifie le modèle de l'objet, qui est lui-même définit 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 s'agit des fonctions suivantes : `applyTextureFromImg()`, `applyTextureFromColor()` et la dernière est nommée `applyTextureFromShaders()`. Comme leur nom l'indique, il est possible d'appliquer des textures aux objets via différents moyens.

(3) Finalement, la fonction `setNodePosition()` va permettre de modifier la position d'un noeud en lui attribuant de nouvelles coordonnées. Il est ainsi possible d'obtenir un rendu graphique lisse avec des objets en mouvement en appelant cette fonction à chaque *frame* et en passant par les fonctions `rerender()`.

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 standard de PATL, un certains nombres de types abstraits sont fournis, soit hérités de Rendery tels que `Color` ou encore `Angle`, soit créés 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 nécessaire d'importer la librairie PATL 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 dans la section 4 et concernent notamment les chargeurs de fonctions OpenGL et les différentes librairies à importer.

5.2 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`.

Différentes fonctions telles que `sizePlanet()` ou `distPlanet()` permettent d'obtenir une mise à l'échelle réaliste des tailles des planètes et des distances qui les séparent. La fonction `render()` 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 la fonction `render()` du composant bas niveau `Sphere` et en leur passant tous les paramètres nécessaires contenus dans l'état global (nom, scène, position, propriétés). Dans la fonction `render()` du composant `Sphere`, comme évoqué auparavant, la construction d'une sphère 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 `update()` appelle la fonction `updateState()` 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` ce qui justifie l'utilisation de la fonction `updatePlanetPos()` dans la fonction de mise à jour, permettant de recalculer le nouvel angle de chaque planète. Un appel à la fonction `rerender()` est effectué dans la fonction `update()` dans le but de mettre à jour l'affichage du rendu après chaque modification de l'état. Cette fonction `rerender()` dans le composant haut niveau va appeler la fonction `rerender()` de chaque sphère dans laquelle seule leur position sera modifiée.

La fonction `update()` permettant de mettre à jour l'état global du système avec `updateState()` ainsi que l'affichage avec `rerender()` est appelée en permanence (en fonction du nombre de *frame* par seconde défini pour l'application) dû à la fonction `registerTick()`.

Finalement, le point d'entrée du programme est donné par l'appel à la fonction `createScene()` avec comme argument le composant haut niveau de l'application, c'est-à-dire `systemSolar`.

Une image du rendu de cette application est donnée en Figure 15 dans laquelle nous pouvons bien observer au centre le soleil et gravitant autour les différentes planètes.

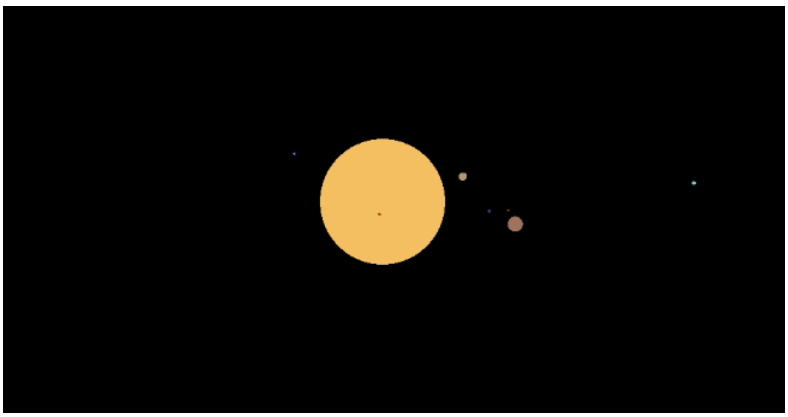

FIGURE 15 – Rendu du Système solaire avec une application PATL

5.3 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 PATL 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 PATL 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 [11]. La Figure 16 présente l'accueil et les différentes rubriques de ce dernier tandis que la Figure 17 montre la partie du didacticiel sur l'explication concernant la création d'une caméra pour visualiser la scène ainsi que le rendu obtenu.

FIGURE 16 – Didacticiel PATL

FIGURE 17 – Création d'une caméra dans un programme PATL

6 Evaluation

6.1 Observations sur le langage

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

L'utilisation du langage Swift avec sa documentation extrêmement bien détaillée va dans ce sens et permet d'obtenir des programmes aisément maintenables. En effet, le système de composants, 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 de dictionnaire permet aussi de regrouper de manière simple les informations, ce qui est très utile pour la mise à jour des différents composants. L'utilisation des structures proposées par le langage Swift facilite elle aussi la création et l'usage de types abstraits.

Créer une application avec PATL 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 construire des objets 3D dans une scène tridimensionnelle. Ces fonctions utilisent celles implémentées dans Rendery, le moteur de rendu graphique à la base de PATL.

La particularité de PATL est le fait qu'il s'agit 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 facilite la résolution d'erreurs dans le code. De plus, le langage PATL est un langage haut niveau et accentue de ce fait la volonté de vouloir faciliter l'écriture des programmes.

Le didacticiel mis en place permet de facilement prendre en main le langage PATL grâce aux explications données ainsi qu'à la représentation de l'affichage du rendu pour des codes exemples. Cette documentation permet ainsi de savoir comment manipuler les éléments les plus importants en ce qui concerne la création d'une application graphique en PATL.

6.2 Améliorations

Cependant, le langage PATL a été testé pour le moment uniquement au travers d'implémentation de cas pratiques. Il semblerait donc intéressant de pouvoir faire tester ce langage à différents types de programmeur, aussi bien experts que novices de la programmation graphique, afin d'observer son efficacité. Ce langage vise à faciliter le développement mais peut, en contrepartie, impliquer trop de "perte de liberté" pour le programmeur.

7 Conclusion et travaux futurs

A travers ce projet, nous avons étudié une comparaison entre OpenGL et DirectX, deux API 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. Nous nous sommes penchés sur Rendery, un moteur de rendu écrit en Swift ayant ainsi été développé afin de combler les manques liés aux outils proposés par Swift. Dans l'idée de créer un nouveau langage facilitant le développement de programmes graphiques, nous nous sommes focalisés sur les langages déclaratifs et plus précisément sur React. Le langage PATL a ainsi été créé. Ce dernier est complètement déclaratif, se base sur Rendery ainsi que sur le langage Swift et reprend des notions importantes de React. PATL permet globalement de 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 ensuite été implémentés tel que par exemple le système solaire et nous donnent de bons rendus graphiques. La librairie standard du langage reste en cours de développement et a pour but d'être la plus complète possible afin d'implémenter toutes les fonctionnalités de Rendery. 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] Opengl. https ://www.opengl.org/.

[2] Silicon graphics. https ://en.wikipedia.org/wiki/Silicon_Graphics.

[3] Khronos groups. https ://www.khronos.org/.

[4] Directx. https ://docs.microsoft.com/en-us/windows/win32/directx.

[5] Microsoft. https ://www.microsoft.com/en-us/.

[6] Vulkan. https ://www.khronos.org/vulkan/.

[7] Shaders. https ://fr.wikipedia.org/wiki/Shader.

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

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

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

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

[12] Definition système de coordonnées. https ://learnopengl.com/Getting-started/Coordinate-Systems.

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

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

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

[16] React. https ://fr.reactjs.org/.

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

[18] 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.

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

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

[21] Swift documentation. https ://swift.org/documentation.

[22] Classement tiobe. https ://www.tiobe.com/tiobe-index/.

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

[24] Documentation skshader. https ://developer.apple.com/documentation/spritekit/skshader.

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

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

[27] Definition domain-specific language. https ://en.wikipedia.org/wiki/Domain-specific_language.

[28] 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.

[29] 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.

[30] Guillaume Huard. Paradigmes de programmation.

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

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

Annexes

```
class SphereExample: Scene {
  override init() {
    super.init()

    backgroundColor = "#000000"

    let sphereNode = root.createChild()
    sphereNode.name = "Sphere"
    sphereNode.model = Model(
      meshes: [.sphere(segments: 100, rings: 100, radius: 50.0)],
      materials: [Material()])
    sphereNode.model?.materials[0].multiply =
     .color(Color(red: 0.5, green: 0.5, blue: 0.5, alpha: 0.5))
    sphereNode.translation.x = 0.0
    sphereNode.translation.y = 0.0
    sphereNode.translation.z = 0.0

    let cameraNode = root.createChild()
    cameraNode.name = "Camera"
    cameraNode.camera = Camera()
    cameraNode.camera?.farDistance = 500.0
    cameraNode.translation = Vector3(x: 100.0, y: 10, z: 0.0)
    cameraNode.constraints.append(LookAtConstraint(target: root))
  }

func sampleScene() {
  guard let window = AppContext.shared.initialize(width: 1500,
      height: 800, title: "Example")
    else { fatalError() }
  let scene = sphereExample()
  window.viewports.first?.present(scene: scene)
  AppContext.shared.targetFrameRate = 60
  AppContext.shared.render()
}
```

FIGURE 18 – Rendery code exemple

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