programmez!

Le magazine des dévs - CTO & Tech Lead

AVRIL - MAI - JUIN 2026



SPÉCIAL 50 ans d'Apple

1976 Apple I

1977 Apple II

1984 Macintosh

1998 iMac

2001 iPod

2007 iPhone

M 01642 - 22 - F: 7,00 € - RD

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € P928345 - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

PORTAGE

Fiparco Portage

Portage salarial

Votre carrière est une priorité !

Votre contact

Ikram 07 86 45 01 75

fiparco-portage.fr



FIPARCO Sponsor de PROGRAMMEZ !

SOMMAIRE#HS22

4 **Edito**
Les sorciers du Macintosh

5 **Nos formules d'abonnement à Programmez!**

Agenda
6 Les événements à venir

7 DevCon sécurité édition 2026

8 Des origines à l'iPhone (1970-2007)

Aux origines : une histoire de rencontres

Homebrew Computer Club et les origines de l'Apple I

Apple I

Apple II

Basic : comment Microsoft a imposé son Basic…

Lisa : une incroyable machine quasiment oubliée

Macintosh 128k de janvier 1984

Susan Kare invente l'âme de l'ordinateur

Quels modèles de développements pour les logiciels Macintosh ?

Macintosh Plus : le modèle qui impose le Mac

La programmation facile avec… HyperCard

1993-1997 : psychodrame chez Apple, le retour de Steve Jobs

8 échecs (1980-2017)

Les transitions matérielles : Apple ou l'art de la maîtrise

2007 : l'iPhone révolutionne le téléphone mobile

Dossier spécial de François Tonic

45 
AppleScript : les applications s'ouvrirent au monde extérieur
Christine Buttin

47 Entre passé et présent


Du copier/coller aux Writing Tools d'Apple Intelligence

Switcher & Swift 6

Etienne Vautherin

De Quickdraw à SwiftUI

56 Le présent


50 ans d'Apple : quelle place pour les développeurs ?
Baptiste Davon


Apple & Open source
François Tonic


Une base, plusieurs plateformes : partager le code, adapter l'expérience / **Etienne Vautherin**


Oui, j'adore Xcode !
Florent Morin


A Swift ouvert
Clément Sauvage


Swift : 12 ans d'évolution, du mobile au cloud
Sébastien Stormacq


Xcode : un héritage de 30 ans
Clément Sauvage

79 Présent et demain


Apple, trop en avance sur son temps ?
Franck Perthuis


L'intelligence visuelle, le futur d'Apple ?
Patrick-Le-Geek

42 Boutique Programmez!

43 Boutique Technosaures

Programmez! est une publication bimestrielle de Nefer-IT. Adresse : 150, rue Lamarck, 75018 Paris – France. Pour nous contacter : redaction@programmez.com

Les sorciers du Macintosh : Apple a (déjà) 50 ans !

À peine 10 mois après Microsoft, Apple Computer naît officiellement le 1er avril 1976 avec un accord de partenariat entre Wozniak, Jobs et Wayne. Woz reçoit 45 %, Jobs 45 % et Wayne 10 %. Dès le 12 avril, le partenariat est modifié : Wayne se retire.

Jobs et Woz se connaissent grâce à un ami commun, Bill Fernandez qui deviendra le 4e employé d'Apple. Nous y reviendrons.

Apple, qu'on l'aime ou non, est une des marques technologiques les plus iconiques, et ce, depuis la sortie de l'Apple I.

Comment expliquer cette fascination, cette aura si particulière d'Apple que peu de marques ont réussi à bâtir ? Il y a les produits et les personnalités. Steve Jobs a très vite été le visage d'Apple. Woz, introverti, n'aimait pas être sur scène, et préférait créer.

L'Apple II a permis à la micro-informatique de sortir des Homebrew Computer Club et d'installer la micro-informatique à la maison et en entreprises. Le succès de l'Apple II en entreprise, avec la sortie de VisiCalc, décide IBM à investir sur ce nouveau marché.

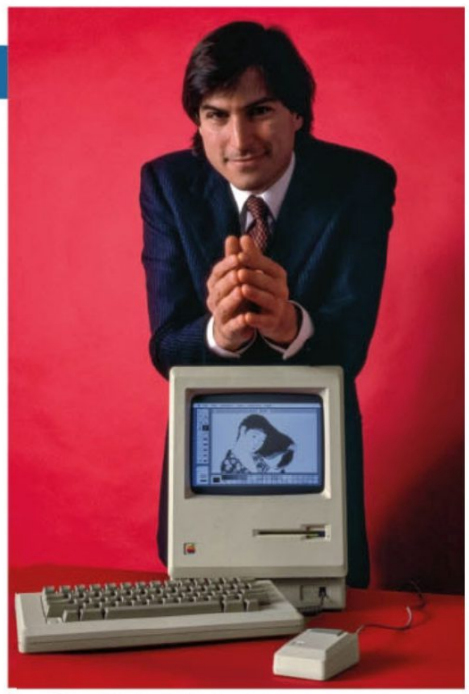
Macintosh est un autre tournant fondamental dans la technologie : la généralisation de l'interface graphique et de la souris. Ce modèle n'est ni le 1er ni le plus performant, mais il réussit une fusion logiciel – hardware et utilise au mieux les ressources limitées du Mac.

Hypercard influence Tim Berners-Lee quand il conçoit le WWW et le HTML.

Newton, une technologie beaucoup trop ambitieuse, montre ce que pourrait être l'informatique ultra mobile. Le Palm Pilot réussit, avec un produit plus « archaïque », à le prouver.

L'iPod révolutionne la manière d'écouter la musique et provoque un séisme dans l'industrie musicale et impose le streaming avec iTunes.

L'iMac de 1998 redéfinit ce que doit être un micro-ordinateur. À lui seul, l'iMac enterre la disquette, impose une connexion à Internet facile et l'USB.



Steve Jobs et Macintosh. Janvier 1984.
Photo de Bernard Gotfryd, Librairie du Congrès (Etats-Unis)

Que dire de l'iPhone. Nokia, BlackBerry et Microsoft ne l'ont pas pris au sérieux.

Bien sûr, Apple a aussi subi de nombreux échecs, des produits catastrophiques et des technologies rapidement abandonnées. La disparition de Jobs en octobre 2011a laissé un vide dans le monde technologique et chez Apple. Succéder au co-fondateur fut sans aucun doute un défi colossal pour Tim Cook. Il n'a jamais eu une approche aussi fusionnelle des produits et de la technologie que Jobs avait. Mais Cook a eu le réflexe de ne pas vouloir faire du Jobs. La question se pose sur la succession : un retour vers une personnalité plus produit et technique ?

Nous avons eu la chance de voir Steve Jobs deux fois à Paris, notamment pour la présentation du Cube. Nous avons rencontré de nombreuses personnalités de la tech, mais peu nous ont autant impressionnés par la présence et le charisme. Vous avez dit champ de distorsion ?

Programmez! ne pouvait pas oublier les 50 ans d'Apple. Au printemps 2025, nous avions publié les 50 ans de Microsoft. Impossible de ne pas revenir sur les débuts du constructeur, l'aventure du Macintosh, les plus grands succès, les plus grands échecs, l'iPhone. Nous voulions aussi revenir sur l'évolution des langages et du modèle de développement des applications sur les systèmes Apple.

Nous parlerons de Swift, un peu d'IA et être développeur macOS / iOS en 2026.

François Tonic

programmez!

Directives de compilation

Programmez! hors-série n°22
AVRIL - MAI - JUIN 2026

Directeur de la publication : **Jean-Christophe Tic**
Rédacteur en chef : **François Tonic**
ftonic@programmez.com

Contacter la rédaction
redaction@programmez.com

Les contributeurs

Christine Buttin
Dominique Bultez
Baptiste Davon
Etienne Vautherin
Florent Morin
Sébastien Stormacq
Clément Sauvage
Franck Perthuis
Patrick-Le-Geek

Maquette
Pierre Sandré

Marketing – promotion des ventes
Agence BOCONSEIL - Analyse Media Etude
Directeur : Otto BORSCHA
oborscha@boconseilame.fr
Responsable titre : Anthony Carrée
Téléphone : 09 67 32 09 34

Publicité
Nefer-IT
Tél. : 09 86 73 61 08
ftonic@programmez.com

Impression
Imprimerie Léonce Deprez
ZAC Artoipole 2
962 Allée de Belgique,
62128 Wancourt

Dépôt légal
A parution

Commission paritaire
1225K78366

ISSN
2279-5001

Abonnement
Abonnement (tarifs France) : 55 € pour 1 an, 90 € pour 2 ans. Etudiants : 45 €.
Europe et Suisse : 65 €
Algérie, Maroc, Tunisie : 64 € - Canada : 80 €
Tom - Dom : voir www.programmez.com.
Autres pays : consultez les tarifs sur www.programmez.com.
Accès aux archives : 25 € (1 an)
30 € (2 ans).

Pour toute question sur l'abonnement :
abonnements@programmez.com

Abonnement PDF
monde entier : 39 € pour 1 an.

Nefer-IT
150, rue Lamarck, 75018 Paris
redaction@programmez.com
Tél. : 09 86 73 61 08

Toute reproduction intégrale ou partielle est interdite sans accord des auteurs et du directeur de la publication. © Nefer-IT / Programmez!, mars 2026.

Abonnez-vous à programmez!

Formules	1 an	2 ans	Etudiant	Numérique
Vous recevez le magazine chez vous	1 an 10 numéros (papier) 55 €	2 ans 20 numéros (papier) 90 €	1 an 10 numéros (papier) 45 €	1 an 10 numéros (format PDF) 39 €
Abonnements + accès aux archives	80 € (1 an + archives)	120 € (2 ans + archives)		64 € (1 an + archives)

Tarifs France métropolitaine.
Tarif PDF : valable partout dans le monde

L'abonnement comprend : les numéros normaux et les hors-séries

L'abonnement **numérique** est disponible uniquement sur notre boutique : **www.programmez.com**

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
PROGRAMMEZ, Service Abonnements
57 Rue de Gisors, 95300 Pontoise

☐ Abonnement 1 an :	55 €
☐ Abonnement 2 ans :	90 €
☐ Abonnement 1 an Etudiant :	45 €
Joindre une photocopie de la carte d'étudiant	

Option : accès aux archives

☐ Pour abonnement 1 an	25 €
☐ Pour abonnement 2 ans	30 €

☐ Mme ☐ M. Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

Adresse email indispensable pour la gestion de votre abonnement

E-mail : @

☐ Je joins mon règlement par chèque à l'ordre de Programmez !

☐ Je souhaite régler à réception de facture

* Tarifs France métropolitaine

Offres pouvant s'arrêter à tout moment, sans préavis **PROG HS22**

événement programmez!

DevCon #27 : édition 2026

28 mai au campus 42 à Paris

agenda & inscription : programmez.com

LES PROCHAINS NUMÉROS À PARAITRE

PROGRAMMEZ! N°274

3 avril 2026

PROGRAMMEZ! N°275

29 mai 2026

EXPOSITION EXCEPTIONNELLE

" 50 ans d'Apple "

à Devoxx France 2026

Du 22 au 24 avril

Les conférences à venir

Juillet

- **Azur Tech Summer** - Valbonne : *2 juillet*
- **Sunny Tech** - Montpellier : *2&3 juillet*
- **Agile Lyon** - *3 juillet*
- **Riviera Dev** - Sophia Antipolis : *du 6 au 8 juillet*

Août

- **State of the Map** - Champs sur Marne : *du 28 au 30 août*

Septembre

- **JUG SUmmer Camp** - La Rochelle : *4 septembre*
- **OWASP AppSec** - Paris : *24 septembre*
- **Tech Summit on AI & Robotics** - Paris : *28 & 29 septembre*

AVRIL 2026

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
		1	2	3	4	5
		FlowCon / Paris	Xen Spring / Grenoble			
		InCyber/Lille				
		AWS Summit / Paris	Pragma Cannes			
6	7	8	9	10	11	12
	PyTorch Europe / Paris		Android Makers / Paris			
			DrupalCamp Grenoble			
13	14	15	16	17	18	19
			MixiT / Lyon			
20	21	22	23	24	25	26
		Devoxx France / Paris				
27	28	29	30	31		

MAI 2026

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
	LyonCraft					
18	19	20	21	22	23	24
				AFUP Day Paris, Lille, Bordeaux, Lyon		
25	26	27	28	29	30	31
			DevCon / Paris			
			Cloud Toulouse			

JUIN 2026

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
1	2	3	4	5	6	7
	OW2Con / Châtillon			Fork It / Rouen	PolyCloud / Montpellier	
8	9	10	11	12	13	14
	JFTP / Montrouge		DevQuest Niot			
			DevLille			
				Tech Fest / Nancy		
15	16	17	18	19	20	21
		VivaTech / Paris				
			Tech'Work / Lyon			
22	23	24	25	26	27	28
		BreizhCamp / Rennes				
29	30	31				

Merci à Aurélie Vache pour la liste 2026, consultable sur son GitHub : https://developers.events/#/2026/calendar

DevCon #27 : I.A. édition 2026



Rendez-vous le 28 mai à 42 (Paris)

Source photo : Adobe Stock

AGENDA & INSCRIPTION SUR WWW.PROGRAMMEZ.COM

Aux origines : une histoire de rencontres 1970-1975

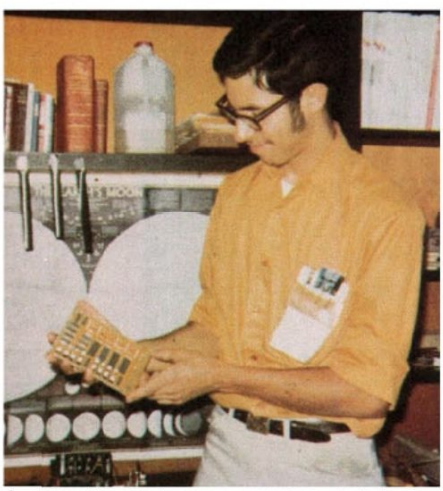
Les premiers protagonistes de notre histoire se connaissent bien avant la création d'Apple. Bill Fernandez et Steve Wozniak (Woz) se connaissent depuis de nombreuses années. Mais Fernandez connaît aussi, de son côté Steve Jobs. Ce sont des nerds. Ils se sont rencontrés durant le collège et n'ont cessé de se voir. Le jeune Steve passait beaucoup de temps au domicile de Fernandez. Déjà adolescent, Jobs aimait marcher pour discuter. Et c'est durant l'une de ces marches que Fernandez présente Woz à Jobs. Nous sommes en 1969/70 (les dates varient).

Woz, qui adore l'électronique et les ordinateurs, travaille avec Fernandez à son premier ordinateur réel, le Cream Soda. Il avait réussi à trouver une vingtaine de composants TTL dont certains avaient été donnés par l'entreprise où il travaillait, Tenet.
Jobs n'est pas aussi inventif que Woz mais l'électronique et la technologie vont rapidement l'intéresser. Il participe au club d'électronique de son lycée Homestead à la fin des années 1960. Pour un projet, il cherche des composants, Steve appelle directement Bill Hewlett, co-fondateur de HP. Ce dernier lui propose un job d'été dans une des usines du groupe (1968).

Jusqu'à alors, Woz imaginait des concepts mais il n'avait jamais passé l'étape suivante : construire réellement l'ordinateur. Le Cream Soda était très rudimentaire : pas d'écran, pas de clavier, un lecteur de cartes perforées. C'est durant la conception du Cream Soda que Fernandez fait rencontrer les deux Steve. Malheureusement, le prototype est détruit durant une démonstration lorsqu'un journaliste du Peninsula Times marche sur le câble d'alimentation.
Du Cream Soda, il reste les souvenirs de Woz et de Fernandez. Aucun schéma n'est connu et une seule photo de la carte mère subsiste.
La même année, Woz est embauché par HP pour travailler sur les calculatrices électroniques.



Blue Box, exemplaire conservé au Powerhouse Museum.
© Maksym Kozlenko



Sans doute l'unique photo existante du Cream Soda. La carte est tenue par Bill Fernandez. Photo de 1971. Collection Fernandez.

Automne 1971-72 : dans la lignée de Captain Crunch, la Blue Box

Woz lit un article de Ron Rosenbaum sur le principe du phreaking, Secrets of the Little Blue Box. Le phreaking est une technique de piratage des lignes téléphoniques. Il appelle l'autre Steve. Ils sont captivés. L'article évoque les hackers du tèlèphone. L'un d'entre eux est sans doute le hacker le plus connu des années 60 et 70 : Captain Crunch, alias John Thomas Draper. Il découvre que le sifflet des boîtes de céréales Cap'n Crunch émet un ton de 2600 Hz qui est la fréquence du réseau AT&T pour dire qu'une ligne longue distance est disponible.

Woz rencontre le hacker qui lui explique le phreaking et comment construire une Blue Box. Woz veut créer quelque chose de plus stable dans son utilisation. Il crée une Blue Box avec des composants électroniques, des compteurs numériques, un PCB sur mesure.

Jobs y voit une opportunité, alors que Woz y voit surtout un défi et un hobby. Ils ciblent les étudiants. Ce sont plusieurs dizaines de Blue Box qui sont vendues en quelques mois. L'association des deux Steve est un réel succès.

L'aventure Atari (1974-75)

Après la Blue Box, les Steve continuent à se voir. Mais les deux ont des parcours différents. Jobs rencontre Allan Alcorn au siège d'Atari. Il n'a même pas 18 ans et il est dans sa phase hippie. Sans diplôme, Jobs gonfle son CV et veut un poste chez



ATARI

INTER-OFFICE MEMO

TO: Stephen Bristow

FROM: Steven Jobs DATE: 23 August 1975

SUBJECT: Addendum to Final Equations for Astrochart

These are the revisions of the final equations for Planetary Coordinates. The changes are: 1) The 'U' series of equations has been eliminated, 2) The expansions V8 and V9 have been reduced again, and 3) The entire equations are in degrees.

FINAL EQUATIONS

V1= L + d1 * D
V2= P ***For Moon Only: V2= P + dp *D/36525
V3= U
V5= I
V6= A
V7= (V1-V2) MOD 21600
V8= V6 *(1- V5 *COS(V7))
V9= V7 +(2*V5 *SIN(V7) + V5²*SIN(2*V7))
V0= V9 +(V2-V3)
U6= V8 * COS (V0 + V3)
U7= V8 * SIN (V0 + V3)

Enclosed is a printout of the changes made in comparison to the original data. As you can see, the results are most satisfactory, with the greatest deviance being in Pluto's orbit -still less than 3 degrees!!!

Un des mémos de Steve Jobs quand il travaillait chez Atari. Ici sur le projet Astrochart. Le mémo est daté du 23 août 75. Photo : RR Auctions

Atari. Alcorn l'embauche, mais rapidement, la collaboration avec les équipes est difficile. Il ne prend pas de douche et est peu conciliant. Il travaille notamment sur le jeu World Cup (printemps 74).

Après son travail sur World Cup, il part d'Atari pour rejoindre l'Inde puis 3 mois dans une ferme communautaire en Oregon. Il retourne chez Atari début 75.

Le jeu Breakout lui est confié, mais il faut tout créer de 0. Le défi technique est grand et Jobs sait qu'il ne peut y arriver tout seul. Il fait appel à Woz pour concevoir l'électronique et réduire au maximum le nombre de composants. Le prototype est construit entre janvier et mars 1975, difficile d'être plus précis. Alcorn avait promis une importante prime à Jobs. Pour convaincre Woz, Jobs promet de faire 50/50, ce qui ne sera pas le cas, mais Woz est content de recevoir plusieurs centaines de dollars…

Si le prototype fonctionne, impossible de l'utiliser pour une production en masse. Woz adapta l'électronique au délai imposé (par Jobs). David Dean reprend le projet et crée une nouvelle carte électronique.

Le regard de Woz se tourne sur les micro-ordinateurs et la sortie vidéo. Avec le Homebrew Computer Club, qui ouvre en mars 1975, tout va s'accélérer.

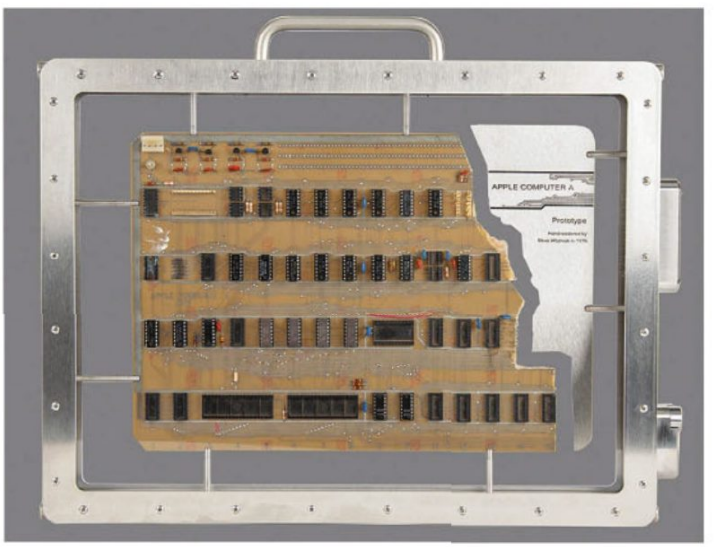
Homebrew Computer Club et les origines de l'Apple I (1975-76)

Nous avons laissé Woz et Jobs sur le prototype Atari. Après le Cream Soda, Woz n'a jamais cessé de suivre les évolutions technologiques et l'apparition des premiers micro-ordinateurs.

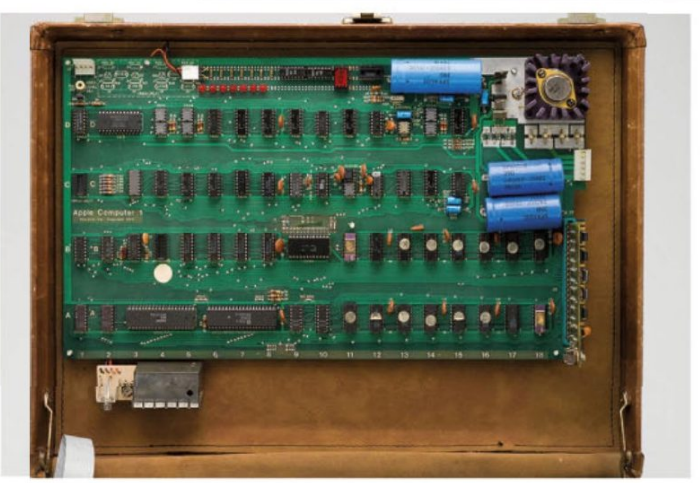
Un des défis de Woz était d'utiliser un téléviseur ou un écran terminal comme moniteur. La lecture d'articles dans Popular Mechanics incite Steve à se lancer dans la conception d'un terminal vidéo. Il bricole en quelques semaines un prototype fonctionnel avec des composants récupérés. Il est capable d'afficher 24 lignes et 60 caractères par ligne. L'idée de concevoir un micro-ordinateur vient durant les séances du Homebrew Computer Club. Ce club est lancé le 5 mars 1975. Woz y assiste.

Il réalise une démonstration de son terminal vidéo. Nous sommes alors en avril 75. Quelques jours lui suffisent pour dessiner les schémas d'un micro-ordinateur. Il est persuadé de pouvoir construire un micro-ordinateur complet avec un clavier et une sortie vidéo au prix de l'Altair 8800. Ce projet est fait en parallèle de son travail chez HP.

Il reste à trouver les composants nécessaires, dont le microprocesseur MOS 6502, un des éléments les plus chers. Mais il n'y connaît rien programmation du MOS. Il apprend rapidement les rudiments de l'assembleur. Nous sommes en juin 75.



Prototype de l'Apple Computer A sans doute du printemps 1976. Carte incomplète. Vendue par RR Auctions pour 677 196 $ en août 2022.



Apple I de 1976. Collection de Powerhouse

Après quelques semaines à monter la carte électronique et à programmer un système moniteur, Woz peut enfin tester réellement sa machine. Le dimanche 29 juin, après un premier essai infructueux, Woz branche l'alimentation. Il tape une combinaison de touches. L'écran affiche le prompt du système moniteur. Woz venait de créer un micro-ordinateur avec une sortie vidéo intégrée…

Le 1er avril 1976 en attendant l'Apple I

Woz améliore son prototype au fur et à mesure. Il emmène Jobs à plusieurs séances du Homebrew Computer Club. Il y va pour aider Woz à porter le matériel et écoute les présentations et suit les échanges.

En mars 76, Woz montre le prototype complet et fonctionnel. Il propose les schémas gratuitement. Lee Felsentien, un des responsables du Computer Club évoque une présentation hésitante, Woz n'était pas bon orateur. Et s'il venait assez souvent, il montait rarement sur scène. Jobs n'était pas d'accord avec l'approche de Woz : les gens ne vont jamais construire leur propre micro-ordinateur. Il faut proposer un modèle complet, monté et prêt à l'emploi. Jobs pousse Woz à construire un micro-ordinateur complet avec un boîtier et un stockage, mais Woz était loin de ces idées ce qui l'intéresse le plus est le défi technique.

Dès l'automne 75, Jobs pousse l'idée d'un kit complet et monté. Mais cela signifie investir de l'argent : faire graver des plaques, acheter les composants. Woz ne veut pas investir autant d'argent et il doute du marché potentiel.

Jobs, au retour d'une réunion, pousse l'idée de créer une société entre amis. Employé par HP, il montre le prototype aux responsables qui ne sont pas intéressés. À partir de là, il est libre d'en faire ce qu'il veut.

Le 1er avril, Apple Computer est officiellement fondé. Les premiers prototypes de l'Apple Computer A (nom initial de l'Apple I) sont construits. Jobs négocie un contrat avec Byte Shop pour une cinquantaine d'exemplaires. Les relations entre Jobs et Paul Terrell, le propriétaire du magasin ne seront pas simples. Mais Terrell est convaincu à la suite d'une démonstration d'un prototype de l'Apple Computer A.

Mais encore faut-il concevoir une interface K7 pour faciliter l'usage d'un cassette pour stocker les données et charger les logiciels et faut aussi créer un interpréteur Basic qui n'existait pas encore sur le MOS 6502. Woz les développe avant la sortie de l'Apple I.

Environ 200 exemplaires de l'Apple I sont vendus. Il illustre ce que doit être un micro-ordinateur : prêt à l'emploi, stockage et interface dédiée, langage de programmation, clavier, sortie vidéo, mémoire vive. D'un coup, Apple venait de créer un nouveau marché et de ringardiser l'Altair et les autres machines équivalentes.

Cette vision d'un micro-ordinateur complet et extensible sera concrétisée avec l'Apple II.

Apple I (1976)

L'Apple I est un concentré de fonctionnalités avancées facturées très chères sur les Altair et autres alternatives :

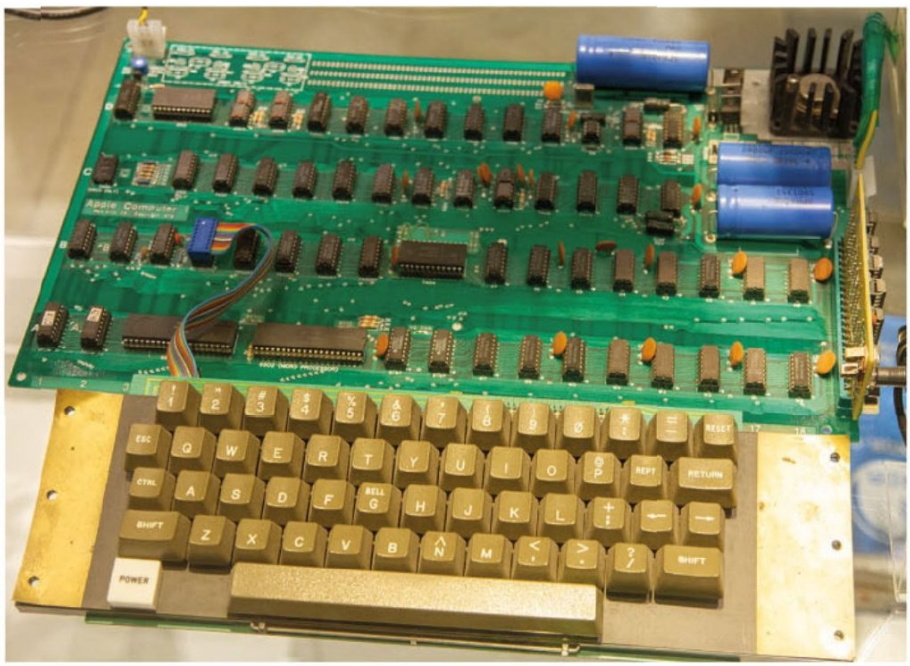
- Carte mère intégrée de 4 à 8 Ko de RAM
- Sortie vidéo par défaut sur un moniteur, une télévision
- Connecteur clavier par défaut
- Interface K7 rapide 1200 bauds
- Simplicité d'extension via un bus
- Possibilité d'utiliser une imprimante (PR-40 de SwTPC)
- WYSIWYG : c'est une révolution pour la micro-informatique

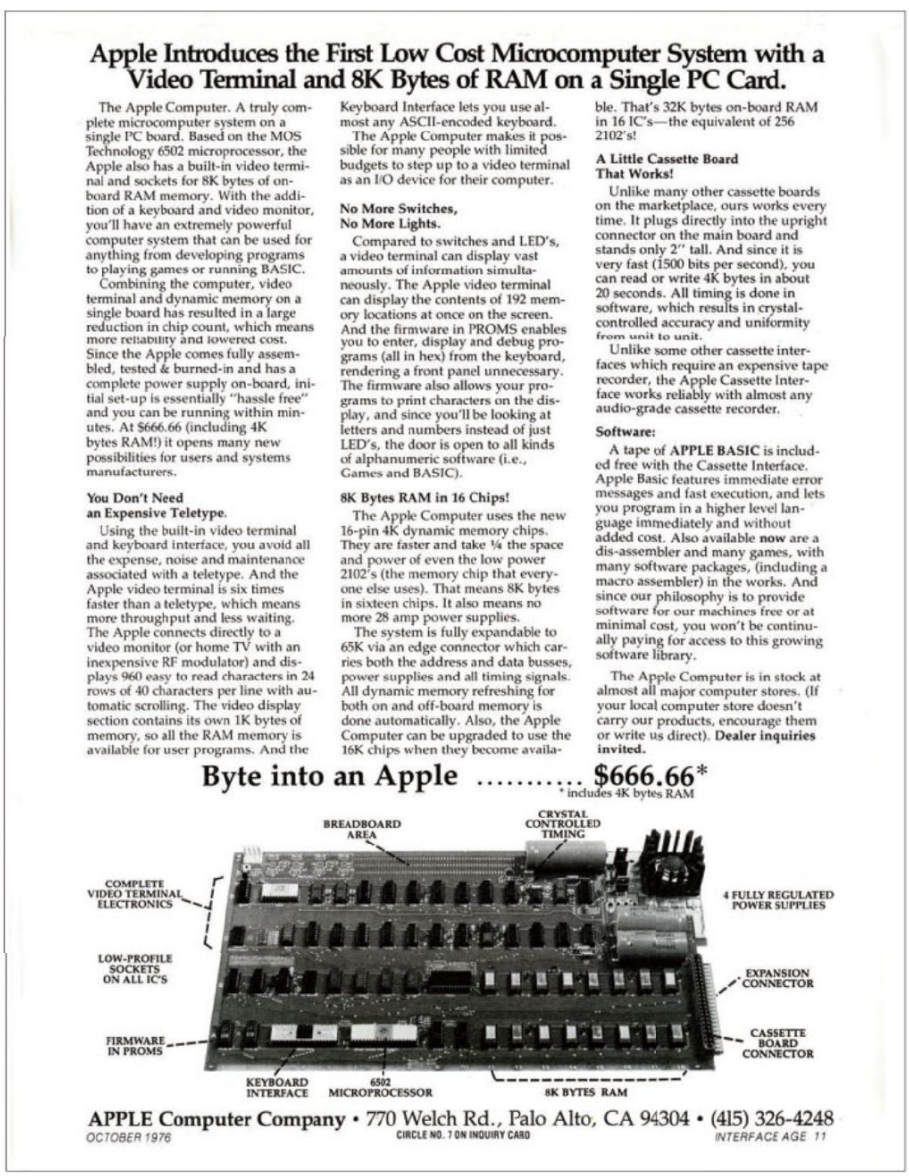
Les caractéristiques

- Processeur : MOS 6502 1 MHz
- Mémoire vive : 4 Ko, extensible à 8 Ko sur la carte mère. En option : carte mémoire de 48 Ko
- Affichage : 40 caractères x 24 lignes
- Vidéo : port vidéo composite
- Interfaces : clavier ASCII, K7, port d'extension
- ROM : 256 bytes
- Logiciels : système moniteur (dans la ROM), Apple Basic pseudo compilé (sur K7)
- Autre : zone planche à pain (pour connecter des circuits et capteurs)

Initialement, aucune interface pour le stockage n'était prévue. Paul Terrell, de Byte Shop, veut une interface K7. Woz finit par accepter. La carte est disponible à la commercialisation de l'Apple I. Elle est vendue 75 $ avec une cassette contenant Apple Basic.

Là où les Altair sont encombrants et lourds, l'Apple I est d'une taille réduite. Au salon The Computer Mart à New York, Stan Veit, propriétaire du 1er magasin d'informatique sur la côte Est, montre l'Apple I et les visiteurs ont parfois du mal à croire qu'un ordinateur peut être aussi petit…





Apple Introduces the First Low Cost Microcomputer System with a Video Terminal and 8K Bytes of RAM on a Single PC Card.

The Apple Computer. A truly complete microcomputer system on a single PC board. Based on the MOS Technology 6502 microprocessor, the Apple also has a built-in video terminal and sockets for 8K bytes of on-board RAM memory. With the addition of a keyboard and video monitor, you'll have an extremely powerful computer system that can be used for anything from developing programs to playing games or running BASIC.

Combining the computer, video terminal and dynamic memory on a single board has resulted in a large reduction in chip count, which means more reliability and lowered cost. Since the Apple comes fully assembled, tested & burned-in and has a complete power supply on-board, initial set-up is essentially "hassle free" and you can be running within minutes. At $666.66 (including 4K bytes RAM!) it opens many new possibilities for users and systems manufacturers.

You Don't Need an Expensive Teletype.

Using the built-in video terminal and keyboard interface, you avoid all the expense, noise and maintenance associated with a teletype. And the Apple video terminal is six times faster than a teletype, which means more throughput and less waiting. The Apple connects directly to a video monitor (or home TV with an inexpensive RF modulator) and displays 960 easy to read characters in 24 rows of 40 characters per line with automatic scrolling. The video display section contains its own 1K bytes of memory, so all the RAM memory is available for user programs. And the Keyboard Interface lets you use almost any ASCII-encoded keyboard.

The Apple Computer makes it possible for many people with limited budgets to step up to a video terminal as an I/O device for their computer.

No More Switches, No More Lights.

Compared to switches and LED's, a video terminal can display vast amounts of information simultaneously. The Apple video terminal can display the contents of 192 memory locations at once on the screen. And the firmware in PROMS enables you to enter, display and debug programs (all in hex) from the keyboard, rendering a front panel unnecessary. The firmware also allows your programs to print characters on the display, and since you'll be looking at letters and numbers instead of just LED's, the door is open to all kinds of alphanumeric software (i.e., Games and BASIC).

8K Bytes RAM in 16 Chips!

The Apple Computer uses the new 16-pin 4K dynamic memory chips. They are faster and take ¼ the space and power of even the low power 2102's (the memory chip that everyone else uses). That means 8K bytes in sixteen chips. It also means no more 28 amp power supplies.

The system is fully expandable to 65K via an edge connector which carries both the address and data busses, power supplies and all timing signals. All dynamic memory refreshing for both on and off-board memory is done automatically. Also, the Apple Computer can be upgraded to use the 16K chips when they become available. That's 32K bytes on-board RAM in 16 IC's—the equivalent of 256 2102's!

A Little Cassette Board That Works!

Unlike many other cassette boards on the marketplace, ours works every time. It plugs directly into the upright connector on the main board and stands only 2" tall. And since it is very fast (1500 bits per second), you can read or write 4K bytes in about 20 seconds. All timing is done in software, which results in crystal-controlled accuracy and uniformity from unit to unit.

Unlike some other cassette interfaces which require an expensive tape recorder, the Apple Cassette Interface works reliably with almost any audio-grade cassette recorder.

Software:

A tape of **APPLE BASIC** is included free with the Cassette Interface. Apple Basic features immediate error messages and fast execution, and lets you program in a higher level language immediately and without added cost. Also available **now** are a dis-assembler and many games, with many software packages, (including a macro assembler) in the works. And since our philosophy is to provide software for our machines free or at minimal cost, you won't be continually paying for access to this growing software library.

The Apple Computer is in stock at almost all major computer stores. (If your local computer store doesn't carry our products, encourage them or write us direct). **Dealer inquiries invited.**

Byte into an Apple $666.66*

* includes 4K bytes RAM

APPLE Computer Company • 770 Welch Rd., Palo Alto, CA 94304 • (415) 326-4248

OCTOBER 1976 CIRCLE NO. 7 ON INQUIRY CARD INTERFACE AGE 11

Apple II (1977) : la révolution de l'informatique personnelle

L'Apple I avait montré tout le potentiel d'un véritable micro-ordinateur intégré et tout-en-un. À peine un an plus tard, en 1977, trois modèles mythiques confirment le rôle de la micro-informatique : l'Apple II, le Commodore Pet et le TRS-80 Model I. Pourtant, les ventes de l'Apple II progressent lentement. Il souffre de plusieurs faiblesses : un tarif élevé, pas de véritables killer app, un stockage limité. Comment et pourquoi l'Apple II va-t-il devenir une référence du marché ?

Wozniak, avant même la finalisation du hardware, dit que l'Apple II sera dix fois plus puissant que l'Apple I. Pour concevoir l'Apple II, il part de zéro. Comme il l'écrit dans sa biographie, il ne veut pas améliorer l'existant. Il se veut ambitieux : gestion par défaut de la couleur, gestion du texte et du graphisme, une nouvelle gestion de la mémoire, intégrer l'alimentation à l'intérieur (une des exigences de Jobs). En parallèle, il faut préparer la commercialisation de l'Apple I, Woz est toujours employé chez HP. L'électronique de l'Apple II est terminée au milieu de l'été 1976. Bill Fernandez est d'une aide précieuse pour transformer les notes et les schémas de Woz en véritables schémas exploitables.

À Atlanta, à la même période, Apple montre l'Apple I, mais officiellement l'Apple II n'existe pas. En réalité, la petite équipe avait finalisé un premier prototype. Il est testé plus ou moins secrètement. Il teste la sortie vidéo, le son et son casse-brique écrit en Basic.



Apple II, système complet. Exemplaire du musée Bolo. Photo : Rama

Apple cherche des investisseurs

Alors que l'Apple I est commercialisé, il faut acheter les composants, monter chaque exemplaire. Les spécificités de l'Apple II sont discutées et tranchées. Woz veut 8 connecteurs d'extension, Jobs n'en veut que 2 pour l'imprimante et le modem. Woz finit par gagner. L'Apple II sera une machine ouverte.

Mais avant de concevoir la prochaine machine, il faut s'occuper de l'Apple I. Byte Shop veut 50 Apple I pour juillet. Apple n'a pas l'argent pour financer la fabrication : il faut 15 000 $. Et le coût de l'Apple I dépasse les prévisions. Jobs contacte des banques, des sociétés d'électronique. Rien. Cramer Electronics accepte de fournir les composants, avec un paiement à 30 jours. Un ami prête 5 000 $.

Mais la production coûte encore trop cher. Jobs impose à Byte Shop la fourniture uniquement de la carte mère. À Byte Shop de proposer le boîtier, le clavier, l'écran. En juillet, Apple respire : les 50 exemplaires sont livrés, les composants payés.

Pour terminer l'Apple II, il faut de l'argent : agrandir l'équipe, créer un boîtier, développer les logiciels, lancer une fabrication de milliers de machires.

Les Steve tentent de proposer l'Apple II à Commodore, qui refuse. Jobs prend contact avec Bushnell d'Atari : 50 000 $ contre 33 % de la société. Il refuse, mais il conseille d'appeler Don Valentine.

La rencontre se fait à la maison des Jobs. Le rendez-vous se passe plutôt bien. Don propose que Mike Markkula soit l'investisseur. Tout s'accélère. Markkula pousse à changer le statut de la société. Le 3 janvier 1977, Apple Computer devient Apple Computer Inc. Et Markkula trouve le premier CEO du nouvel Apple : Michael Scott. Eh oui, Steve Jobs ne fut CEO qu'en 1997, après son retour chez Apple…

Markkula demande à Wozniak d'être à plein temps et de démissionner de HP. Dans un premier temps, il refuse et décide de rester chez HP. Heureusement, il change d'avis quelques jours plus tard. L'Apple II peut continuer sa naissance.

Finaliser l'Apple II

La pression est énorme pour la petite équipe, car il fallait terminer l'Apple II pour espérer le présenter au West Coast Computer Faire d'avril 1977. Quatre mois seulement pour intégrer l'électronique, l'alimentation, le clavier, finir le BASIC et les ROM. Et il fallait créer le design du boîtier, qui n'était pas fixé, créer le moule, etc. Mais la conception prend beaucoup de temps et, quelques jours avant le salon, Apple reçut enfin les premiers exemplaires du boîtier.

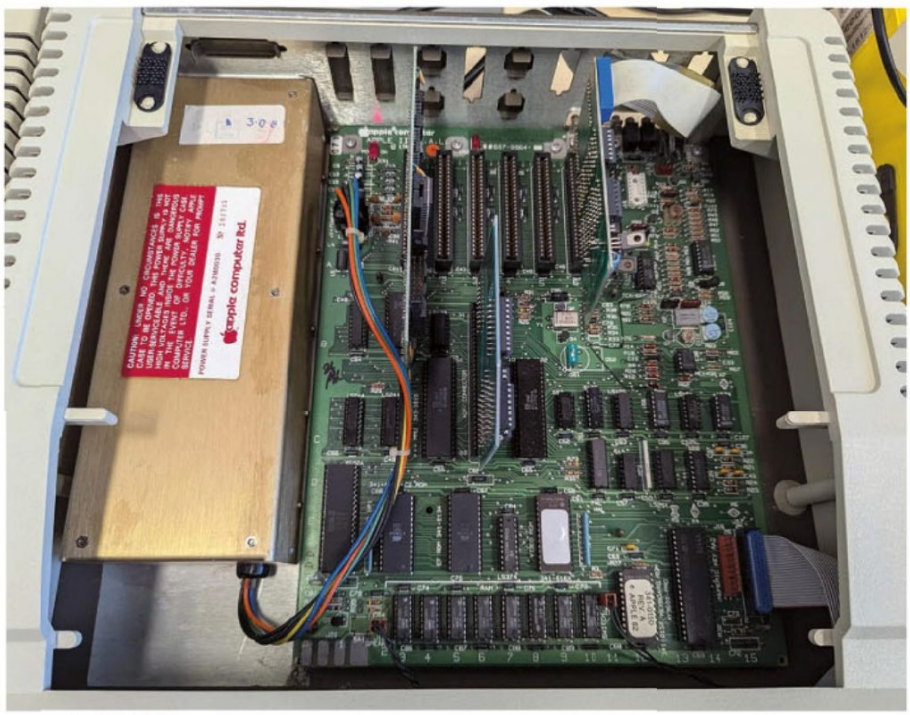
Durant trois mois, Jerry Manock, designer industriel de génie, va définir la caisse et permettre sa fabrication. Mais trouver un designer fut compli-

qué, car déjà à cette époque, Jobs avait une idée précise de ce qu'il ne voulait pas. Manock était indépendant et les délais étaient intenables : à peine neuf semaines pour concevoir la coque et le design général. Jobs comprend rapidement qu'il y a un problème de budget et de temps : six mois de travail et 100 000 $ pour créer les moules ! Manock a dû concevoir le design en fonction de l'électronique à intégrer : la carte mère, l'alimentation, le clavier, prévoir les fentes pour les cartes insérées dans les slots. Dans les premières ébauches, Manock avait imaginé des poignées intégrées, mais Jobs rejeta l'idée. Manock a dû concevoir la structure interne, la dissipation thermique, choisir la couleur. Il n'y a pas de ventilateur. Dès l'époque de l'Apple II, Jobs bannit le ventilateur.

L'accueil de l'Apple II fut tellement positif que les estimations de Jobs furent très rapidement dépassées. Le cofondateur espérait 5 000 ventes la première année. Cette forte demande obligea Manock à trouver des fournisseurs capables de produire en quantité et rapidement la caisse, sans rupture de matières premières.

En juin 1979, l'Apple II+ sort. Il reprend le même design. Il inclut l'AppleSoft Basic en ROM. Le matériel évolue peu. La machine sait booter directement sur une unité de stockage. La version Europlus est la version adaptée du II+ pour l'Europe et l'Australie. Le J-Plus est destiné au Japon.

Les premiers mois de l'Apple II sont difficiles et les ventes progressent lentement. La plateforme est récente et la machine est chère : presque 1 300 $. Mais progressivement, les éditeurs s'intéressent à cet ordinateur. Les logiciels se multiplient, les périphériques aussi grâce aux slots d'extension. Les évolutions apportées par le II+ et le IIe vont faire exploser les ventes. Une autre faiblesse est le Basic de Wozniak : il gère uniquement les entiers.



Intérieur de l'Apple IIe

Une évolution majeure : l'Apple IIe

Le IIe aura un énorme succès. Il embarque 64 Ko de RAM, gère plusieurs modes texte (40 et 80 colonnes) et graphiques (basse, haute, double basse et double haute résolution). Il gère enfin l'affichage des textes en minuscules. On peut rajouter facilement de la RAM via une simple carte fille. Son électronique a été simplifiée grâce à de nouveaux composants. Le IIe sera commercialisé jusqu'en 1993 ! Il y a quelques évolutions de la ROM et de la carte mère, mais elles resteront globalement inchangées. En 1984, la caisse sera légèrement modifiée. Deux évolutions seront lancées : Enhanced IIe (1985) et Platinum IIe.

La gamme Apple II reste la plus vendue même après l'introduction du Macintosh. Après les échecs de l'Apple III et du Lisa, et le demi-succès du Macintosh, l'Apple II maintient les finances du constructeur. Il est tellement important que Microsoft va s'en servir lorsque vient le moment de renégocier le contrat de licence du Microsoft Basic, qui arrive à expiration en 1985…

Apple IIgs (1986) : une si belle machine qu'Apple n'hésite pas à brider

Le GS est l'ultime évolution de la gamme Apple II. C'est aussi le plus clivant. On aime ou on n'aime pas. Au départ, la succession du IIe était très similaire au modèle existant. Ce successeur était le projet IIx. Wozniak n'est plus chez Apple depuis 1981, à la suite d'un accident d'avion et à son éloignement du constructeur. Il y retourne seulement en juin 1983.

DISK II : LE LECTEUR QUI CHANGE TOUT

Le premier lecteur de disquettes Apple est le Disk II. En quelques mois, Wozniak réussit à créer le contrôleur et à programmer les couches logicielles pour gérer les disquettes 5¼ pouces. Jobs avait compris l'intérêt de proposer rapidement un stockage de masse plus performant que la K7 et bien moins cher que le disque dur. Le Disk II arrive en juin 1978. Pour l'utiliser, il fallait installer une carte contrôleur, livrée avec le lecteur. Son succès fut énorme.



Disk II. Photo : Benjamin Healley / Museums Victoria

Pour contrôler le lecteur, Wozniak développa un contrôleur dédié : le fameux Integrated Woz Machine (IWM), nom de code Liron. Il sera utilisé dans les premiers Macintosh. Pour créer le Disk II, la décision est prise d'utiliser le mécanisme du Shugart SA-400 puis du 390 (version dégradée du 400).

Grâce à l'IWM, Woz réussit à réduire le nombre de composants. Il sera fabriqué par Synertek, avec l'aide de Wendell Sander, un ingénieur d'Apple. Une évolution du IWM sera créée pour le Macintosh II : le SWIM (Sander-Wozniak Integrated Machine). Par la suite, Apple utilisera le mécanisme du constructeur Alps, pour des raisons de coût.



Apple IIgs complet. Photo : Ruben de Rijcke

Le projet IIx était déjà en gestation. Il devait utiliser le nouveau processeur 65816 (plus exactement le 65C816 de Western Design Center), compatible avec le vénérable MOS 6502. Le potentiel est là : compatibilité avec le II et processeur plus puissant. Il reprend l'architecture des slots du IIe, avec huit slots et deux slots supplémentaires pour des cartes d'extension dédiées. Ces deux slots étaient connus sous l'appellation : slot -1 et slot 0. Le slot -1 pouvait servir à ajouter une carte processeur (Z80, Motorola 68000 ou Intel 8088).

Mais le projet IIx était un cas de conscience pour la direction : l'Apple II ne devait pas proposer une machine équivalente ni gêner le lancement du Mac. En mars 1984, le projet fut arrêté. Quelques semaines plus tard, le bon accueil du IIc incite la direction à relancer l'idée d'un Apple II 16 bits. À l'été 1984, le projet Phoenix naît sur les bases du IIx. Deux éléments étaient importants : le graphisme et le son.

Wozniak et les ingénieurs matériels s'occupèrent de dessiner et de stabiliser la carte mère. Pour assurer la compatibilité avec les programmes Apple II, il fallut créer une mémoire lente dédiée à la partie 8 bits et une mémoire rapide pour la partie 16 bits. Côté design, le IIgs reprend le langage « Snow White ». Ce design concerne l'unité centrale, le clavier, la souris, l'écran, etc. Les concepteurs optimisent aussi l'intérieur de la caisse en supprimant les vides non essentiels. Visuellement, le IIgs est rectangulaire. Indirectement, on retrouve les lignes du Macintosh II. Pour la première fois, l'Apple II possède un clavier séparé.

L'Apple IIgs souffre de plusieurs limitations :

- Un processeur bridé sur les performances
- Pas de disque dur interne : c'est la grosse lacune du IIgs
- Pas de lecteur de disquette interne

L'absence de stockage interne bride l'usage de la machine et pénalise fortement les performances.

L'une des innovations majeures du IIgs est l'apparition de l'Apple Desktop Bus (ADB). Cette nouvelle connectique permet de chaîner jusqu'à six périphériques (clavier, souris, etc.). Elle deviendra le standard sur Macintosh. ADB ne devait pas être présent initialement : il fut ajouté tardivement.

L'arrivée de GS/OS, un OS graphique couleur reprenant les bases du système Macintosh, fut remarquée. Mais malheureusement, les lenteurs de la machine se ressentent fortement sur l'OS graphique. À noter que pour la partie graphique, les équipes utilisent les routines QuickDraw II développées pour les nouveaux Macintosh. Ces routines sont stockées dans la ROM. Leur développement fut entamé en juin 1985. Cette version fut spécifique au IIgs.

La machine est très bonne. Mais la direction l'a volontairement bridée pour ne pas nuire au Macintosh. Apple manque une occasion unique de s'imposer face aux Atari ST et Amiga, même si Apple n'aurait jamais vendu cette machine au prix d'un ST.

Le manque de logiciels adaptés explique aussi le succès relatif de la machine. Elle sort au mauvais moment pour Apple. Le Macintosh devient, en 1986, la gamme de référence et commence à dépasser les ventes de l'Apple II en valeur.

Basic : comment Microsoft impose son Basic et force la main à Sculley en 1985 ?

Quand Woz travaille sur l'Apple I, le langage Basic n'a pas été porté sur le processeur MOS 6502. Woz développe un interpréteur Basic en quelques semaines alors qu'il n'a aucune base de programmation. Avec la sortie de l'Apple II, la question du langage est plus que jamais discutée. Microsoft voudrait bien signer un deal avec Apple…

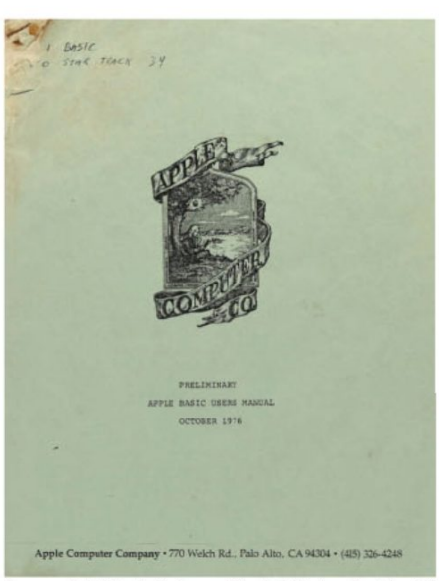
Woz crée l'Integer BASIC. Il s'agit d'un interpréteur disponible sur K7 pour l'Apple I. Par la suite, il sera intégré en ROM. Le gros défaut de cette version est l'absence des nombres flottants. Or le floating-point n'intéresse pas réellement Woz. Gérer les entiers est plus performant. L'avantage de l'Apple Basic est d'être gratuit.

Dans Dr. Dobb's Journal, Woz publie un article sur les routines à virgule flottante pour le 6502. L'article paraît en août 76. L'Integer BASIC est très bien pour les jeux et l'éducation, estime Woz. Les routines sont intégrées dans les ROM de l'Apple. À la sortie de l'Apple II, l'Integer BASIC reste le langage du constructeur. Les utilisateurs veulent un Basic avec la virgule flottante.

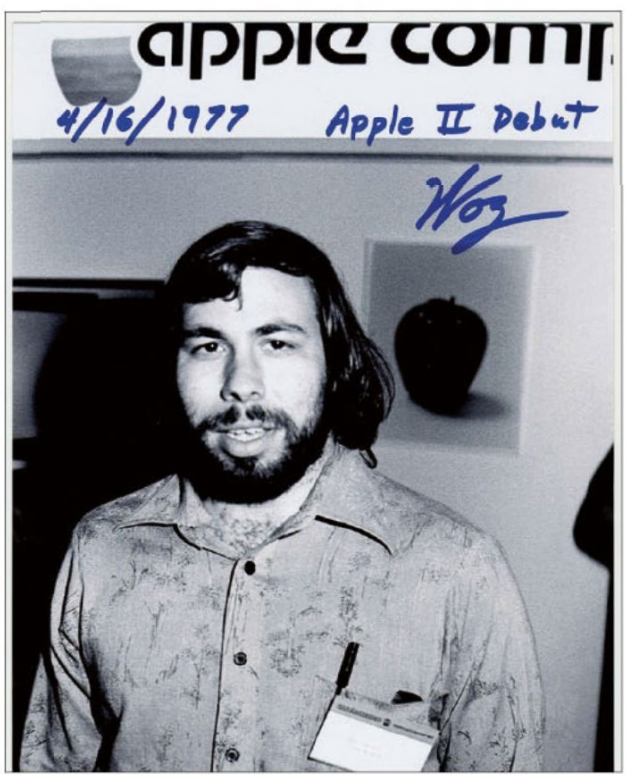
Sur l'Apple I, il faut un lecteur K7 et l'interface K7 :

1. taper C100R
2. taper E000.EFFFR
3. appuyer sur la touche lecture du lecteur K7

Il faut environ 30 secondes pour charger le langage. Pour accéder au Basic, on tape E000R. Pour en sortir, le bouton RESET suffit pour retourner au système moniteur.



Le manuel de l'Apple Basic pour l'Apple I. Version d'octobre 76.



Woz – avril 1977

L'incontournable Microsoft Basic

Depuis 1975, Microsoft fournit le Basic à de nombreux constructeurs. Il s'occupe de porter son langage sur le MOS 6502. Microsoft cherche de nouveaux clients en attendant que l'accord avec Commodore, pour le PET, génère le chiffre d'affaires attendu. Apple est une cible potentielle. Microsoft avait entendu des rumeurs selon lesquelles Apple concevait un micro-ordinateur utilisant le MOS 6502. Des contacts sont pris, mais Apple refuse en arguant qu'un Basic existe déjà.

Mais tout change avec la sortie de l'Apple II et la nécessité d'avoir rapidement un Basic complet. Un accord est signé en août 77 : le contrat a une durée de 8 ans pour licencier le Microsoft Basic sur l'Apple II. Apple paie 21 000 $ (ou 31 000 selon les sources). En attendant de pouvoir l'intégrer à la ROM, cette version sera disponible sur cassette : Applesoft Basic.

Ce Basic se révèle plus lent que l'Integer BASIC, ce qui n'était pas une surprise, et supporte mal les graphiques haute résolution.

Le Microsoft Basic 6502 prend le nom d'Applesoft Basic. Des développeurs d'Apple se chargent d'adapter le logiciel de Microsoft. L'interpréteur et le langage sont portés, et l'équipe ajoute des fonctionnalités propres à l'Apple II. La première version sort sur cassette à l'automne 77. La version disquette sera disponible quand le Disk II sera commercialisé en juin 78.

Avec la sortie de l'Apple II+, en juin 1979, le Basic est désormais directement intégré dans la ROM. C'est pour cela que l'on trouve la mention © Microsoft 77 sur les cartes de l'Apple II intégrant le langage.

Les relations Apple – Microsoft sont plutôt bonnes. Microsoft commercialise des cartes pour l'Apple II, dont une carte pour faire tourner CP/M.

1985 : renouveler le contrat pour l'Apple II et chantage pour les logiciels Macintosh

Le destin du Macintosh Basic, ou MacBasic, est étroitement lié à la relation avec Microsoft et à l'Apple II. Lorsque le Macintosh est développé, la question du langage devient rapidement un sujet central. L'idée de concevoir un Basic optimisé pour le Macintosh fut rapidement actée par l'équipe Macintosh. Donn Denman commence à travailler sur le futur MacBasic dès l'hiver 1981-82.

MacBasic devait disposer du langage et d'un envi-

ronnement complet de développement, comme c'était le cas sur le Lisa. Une petite équipe est mise en place. Le travail avance en parallèle du projet Macintosh, mais en 84, l'environnement est toujours en développement. La lenteur du projet s'explique aussi par la nécessité pour Donn Denman de travailler sur des éléments du futur système et, jusqu'à la sortie officielle du Macintosh, MacBasic est en pause. Le travail reprend au printemps 84.

Entre-temps, Microsoft sort officiellement son Basic sur Macintosh en mars 84, sans qu'Apple n'en soit averti. Il s'agit uniquement de l'interpréteur ; le compilateur arrivera seulement en 1986. Pour l'éditeur, le Macintosh avait un potentiel énorme et il voulait occuper le terrain. Mais cette version est limitée : elle est lente, avec un accès incomplet à la Toolbox Macintosh (donc une gestion limitée de l'interface). L'utilisation est limitée par la mémoire vive disponible et le logiciel nécessite plus de 40 Ko, laissant à peine 40 Ko pour interpréter le code ! La version 2 gérera mieux la mémoire avec le Macintosh 512 Ko.

Chez Apple, le développement de MacBasic avance rapidement. Une version bêta est prête vers le printemps 85. Elle sera même distribuée à des développeurs. La documentation est rédigée, un livre complet est prêt. La bêta porte le numéro 0.335. La version 1.0 est prête à être commercialisée à l'automne 85.

Mais le projet est brutalement stoppé en novembre 1985. La décision est radicale : tout doit être détruit, le code source effacé. Cette décision est la conséquence du renouvellement de l'accord avec Microsoft pour continuer à fournir le Basic pour l'Apple II. Microsoft profite des négociations pour tuer un redoutable concurrent. L'accord expirait au milieu de l'été 85. Les négociations s'engagent entre Gates et Sculley. Gates demande la disparition pure et simple de MacBasic. Sculley cède et l'accord de renouvellement est signé quelques semaines plus tard, provoquant colère et incompréhension au sein de l'équipe qui avait fourni un important travail.

Pour la direction d'Apple, Microsoft Basic reste essentiel à l'Apple II, qui demeure la cash machine du constructeur. Gates profite de la situation pour faire pression sur Sculley. L'initiative vient de Gates, qui appelle Sculley dès qu'il est mis au courant du développement de MacBasic. Il menace même d'une poursuite judiciaire pour rupture de contrat.

Gates met en balance le contrat et le développement d'Excel sur Macintosh. Durant les négociations, Sculley cède sur un autre point sensible qui avait provoqué la colère de Jobs en 84 : Microsoft peut développer des interfaces graphiques sur PC, même si Apple conserve certains aspects spécifiques de l'interface.

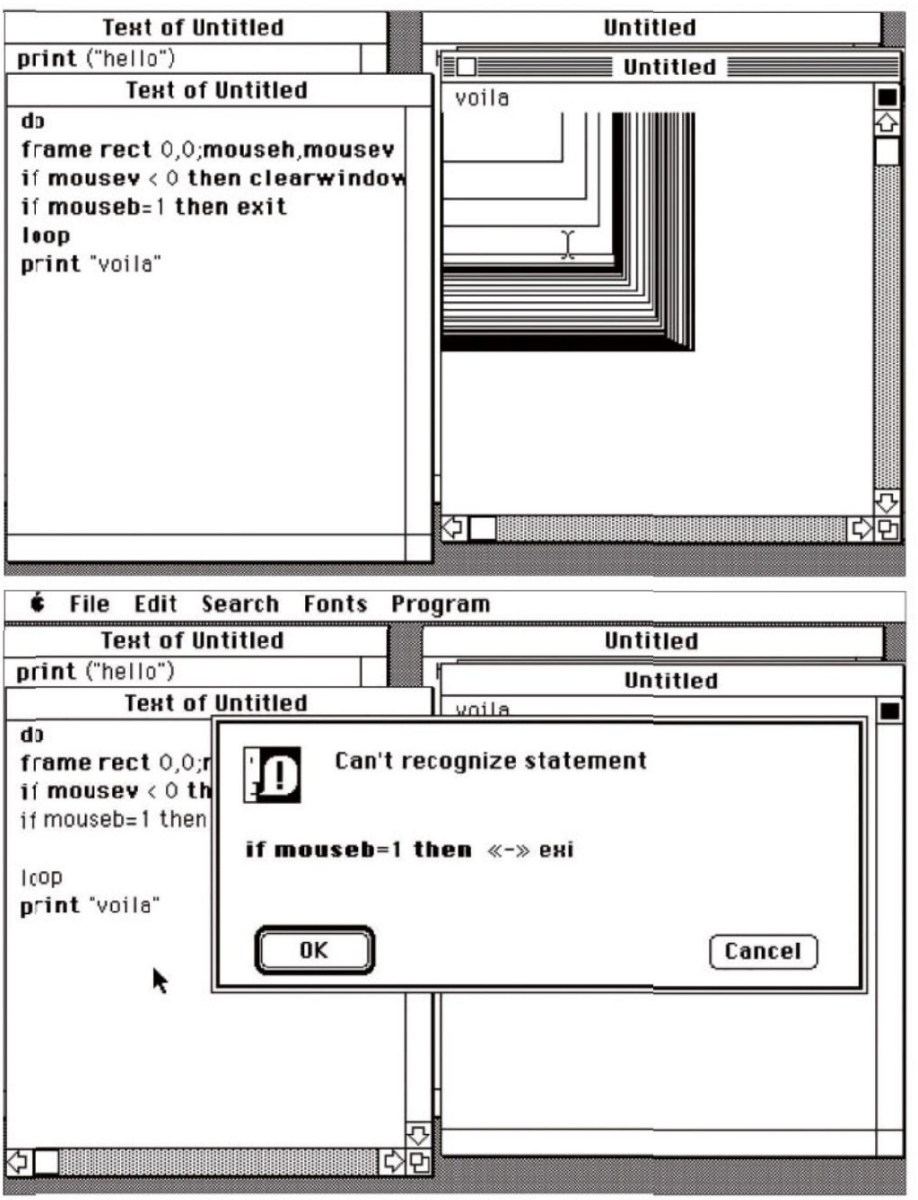
D'autre part, Microsoft veut le droit d'utiliser le nom de Mac Basic, et l'éditeur s'engage à ne pas sortir Excel sur PC dans les 2 ans.

Atkinson résuma la négociation dans le Wall Street Journal ainsi : « Bill Gates insista pour qu'Apple se débarrasse d'un formidable logiciel en nous mettant un pistolet sur la tête ».

Et Steve Jobs dans cette affaire ? Depuis la sortie du Macintosh, il perd peu à peu son rôle opérationnel. En 85, il ne participe à aucune décision stratégique. Sculley le marginalise jusqu'à le pousser à la démission en septembre 85.



Cassette de l'Applesoft Basic, version 1977



Éditeur du MacBasic. Quelques versions bêtas ont survécu à la destruction voulue par Microsoft

Lisa : une incroyable machine quasiment oubliée

Nous avons beaucoup hésité sur les machines à intégrer dans ce spécial 50 ans. Nous aurions pu parler de l'iPod, de l'iMac, du premier PowerBook ou encore de la Pippin, plus spécifiquement d'une technologie. Il y a cependant une machine exceptionnelle oubliée de la plupart des fans d'Apple : le Lisa.

Techniquement, le Lisa est parmi les meilleures machines jamais sorties au début des années 80. *Elle a été pensée dans le moindre détail*. L'OS est une pure merveille. Avec le Xerox Star, le Lisa n'a pas d'équivalent. Mais la direction d'Apple, et l'équipe du projet, ont multiplié les erreurs : obligation d'un disque dur, beaucoup trop cher, pas de logiciels, une dispersion des gammes, une concurrence avec le projet Macintosh.

« Le Lisa a échoué parce qu'il était sous-alimenté (en puissance) et même s'il faisait de belles choses, il les faisait (trop) lentement ». (Alan Kay, peu après la sortie du Macintosh). Oui, le Motorola 68000 est puissant, mais la faiblesse de sa fréquence d'horloge (5 MHz) bride les performances de l'OS et de la couche graphique... Le Macintosh a un hardware limité mais avec un Motorola 68000 à 8 MHz, une RAM réduite au minimum, mais il exécute bien mieux les opérations de l'OS et la couche graphique ne pose aucun problème.



Un long, très long, développement et la révolution de l'interface graphique (octobre 78 – décembre 79)

L'Apple II est sorti en juin 77. Immédiatement, les équipes pensent déjà à l'après. Les projets se multiplient : Apple III, le projet Annie (Macintosh) et le Lisa. Trois gammes différentes et totalement incompatibles... Les premières spécifications apparaissent à l'automne 78. L'idée est de créer un micro-ordinateur nouveau, plus modulaire. La petite équipe pioche les idées dans l'Apple II et l'Apple III.

Tout change avec les visites de Steve Jobs, avec une petite équipe d'ingénieurs, au Xerox Parc en novembre et décembre 79. Apple négocie depuis plusieurs mois un investissement de Xerox dans la société. Jobs veut visiter le Parc et accéder à toutes les technologies en développement.

Pour l'équipe d'Apple, c'est un véritable choc technologique : souris, interface graphique, logiciels, écran haute résolution, système, langage de programmation. Jobs comprend que l'avenir de l'informatique est là. Il veut aider le projet Lisa. Dès mars 80, le Lisa est entièrement rebooté : souris, interface graphique, système, micro-processeur. Et Apple accueille plusieurs ingénieurs clés du Parc dont Alan Kay et Steve Capps.

Un impressionnant hardware

Créer une interface graphique, gérer en temps réel la souris, concevoir un OS graphique complet, le projet prend beaucoup de retard. Pis, l'OS exige un hardware surpuissant :

- Un disque dur : eh oui, le Lisa est livré avec un disque dur de 5 Mo pour installer l'OS qui à lui seul occupe 2,5 Mo ! Le disque dur fait exploser le prix du Lisa
- 1 Mo de RAM : une quantité « gigantesque » pour l'époque. Le Lisa Desktop Manager occupe 512 Ko et 32 Ko pour l'affichage. En comparaison, l'OS du Macintosh occupe 16 Ko de RAM et 22 Ko pour l'affichage graphique.

L'électronique du Lisa est complexe. Les principaux composants sont sur des cartes dédiées (processeur, I/O, mémoire) que l'on enfiche dans un panier.

La fameuse souris à un bouton ne vient pas du Macintosh mais du Lisa. La souris du Xerox Parc possède 3 boutons : trop complexe. L'équipe opte pour un unique bouton : simplifier l'usage et les interactions.

Un OS incroyable mais trop lent et trop complexe

Là où le Macintosh est fluide et homogène avec des interactions claires, le Lisa souffre de la multiplication des fonctionnalités. Au lieu de rationaliser l'interface, les options se multiplient et les interactions manquent parfois de cohérence. La comparaison avec le Macintosh fait mal sur ce point.

Le système de fichiers évolue à chaque version de l'OS :

- Pas de sous-dossier dans les premières versions
- Lisa OS 3 provoque une rupture : structure hiérarchique des dossiers basée sur une approche B-Tree, notion de privilèges d'accès, gestion des étiquettes. Le catalogue d'un disque pouvait contenir 1 200 fichiers.

La gestion des fichiers est fascinante et complexe. Par exemple, quand on duplique un fichier, la copie garde exactement le même nom. Le Desktop Manager (le cœur de l'OS) assigne à chaque fichier un nom unique de bas niveau : le {TxDy}. T spécifie le tool (= le logiciel qui a créé le document) et D est le numéro du document. Par exemple : T1D3, soit Tool 1 Document 3. Cette référence strictement interne n'est jamais visible à l'utilisateur. Cet identifiant unique différencie chaque fichier même s'il a exactement le même nom.

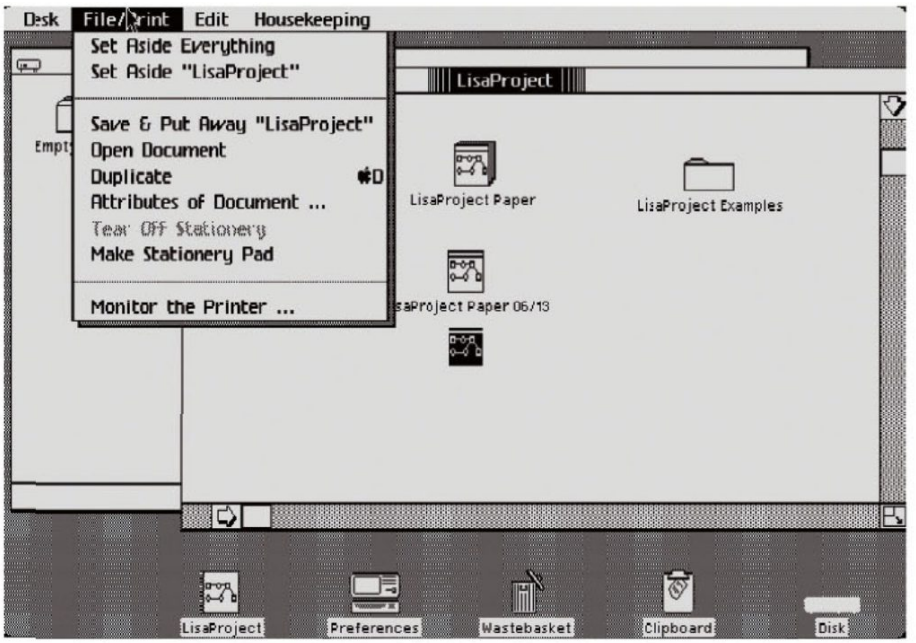
Le LisaOS est un système multitâche coopératif. Les applications en arrière-plan sont suspendues tant qu'elles ne passent pas au premier plan. Ce qui, en 1981/82, complique considérablement le développement car il faut garder l'intégrité des processus et des tâches de chaque application ouverte. Pour éviter qu'un crash d'une application ne corrompe l'ensemble du système, LisaOS intègre une mémoire protégée pour assurer l'isolation de chaque processus.

Autre raffinement de l'OS : la mémoire virtuelle. Le LisaOS peut créer une mémoire virtuelle en utilisant le disque dur (mécanisme de swap) pour créer un espace mémoire. Pratique pour les logiciels exigeant beaucoup de mémoire, mais qui provoque un effondrement des performances : le swap entre la RAM réelle et la mémoire virtuelle pèse sur la charge du processeur.

Selon la localisation du clavier, LisaOS détecte la langue au boot et change automatiquement la langue du système !

Le design physique du Lisa impacte la forme du pixel ! Le Lisa est horizontal, contrairement au Macintosh. Cette horizontalité est influencée par les machines à écrire TypeWriter d'IBM. Le Lisa est avant tout un environnement bureautique.

L'horizontalité du Lisa se voit jusque dans l'affichage : 80 colonnes, une résolution de 720x360 pixels. Et les pixels sont plus hauts que larges, là encore, l'influence de la bureautique. Sur le Macintosh, le pixel est appelé square dot. Il est carré. Cette forme neutre convient à la fois aux usages bureautiques et graphiques.





Un système anticopie

Le Lisa propose plusieurs niveaux de sécurité. La PROM Video MMI 6309-J installée sur la carte processeur contenait 2 numéros de série uniques : 1 pour AppleNet (AppleNet ID, le réseau) et 1 pour le Lisa proprement dit.

Ce ruméro est unique à chaque Lisa et non interchangeable ! Si vous devez changer de carte processeur, il faut transférer la puce sur la nouvelle carte.

Le Lisa intègre aussi un système anticopie logiciel avec le mécanisme de sérialisation. Quand vous copiez une disquette sur le disque dur du Lisa ou quand vous exécutez pour la première fois un logiciel, le système écrit un numéro de série lié au logiciel ! Ainsi, le logiciel ne peut s'exécuter que sur le Lisa identifié.

Macintosh 128k de janvier 1984

Impossible de résumer l'aventure que fut la création du Macintosh en quelques pages. Il nous a fallu 150 pages pour le numéro spécial « 40 ans du Macintosh » dans le magazine Technosaures. Nous avons pris 5 points clés, réflexions et éléments qui font du Macintosh une machine incroyable où la symbiose hardware–software a été poussée à son extrême comme jamais.

Il faut se rappeler qu'il y a eu 2 projets : le projet Annie / Macintosh de Raskin et le projet Macintosh de Jobs lorsqu'il rejoint, le 20 janvier 1981, le projet avant d'en reprendre la direction à la place de Raskin et de rebooter totalement le projet. Ce « nouveau » projet reprend des concepts établis par Raskin, par le projet Lisa et la vision d'ensemble de Jobs. Quand il arrive dans les locaux Macintosh, peu de choses fonctionnent. Le design n'est pas défini, le hardware est à peine balbutiant et le logiciel n'est qu'un ensemble de code avec quelques fonctionnalités.

1 / le design : verticalité vs horizontalité

Pourquoi le Macintosh est-il vertical et non horizontal comme le Lisa ou encore l'IBM PC et la majorité des ordinateurs ? Pour travailler sur le form factor, définir les matériaux et le design global, Jobs recrute des designers connus d'Apple : Jerry Manock, Terry Oyama et Zeidek. Malgré quasiment 3 ans de travail, le projet de Raskin n'avait qu'une idée plus ou moins arrêtée : un design horizontal à la Lisa.

Pour Jobs, le Macintosh est dédié à la maison, à l'école, aux travailleurs indépendants et aux petites entreprises. Très rapidement, le concept de base émerge : il faut une forme compacte minimisant l'emprise en cm^2 sur un bureau. Visuellement, il ne doit pas agresser le regard. Il doit s'intégrer naturellement à son environnement. L'autre critère est la conception industrielle de la Ford T : une forme unique pour une production de masse avec une seule configuration.

Rapidement, des pistes commencent à émerger :

- Un boîtier fermé, sans avoir la portabilité comme objectif
- Un boîtier qui prend peu de place sur un bureau

Le Macintosh, selon Jobs, n'a pas besoin d'être ouvert. Et pour l'ouvrir, il faut un outil spécial. En ce sens, Jobs reprend le concept de Raskin. L'équipe regarde vers les appareils du marché tels que les machines à café, le grille-pain, etc. Ils prennent peu de place : il faut que le Macintosh fasse de même. Le Macintosh de Raskin utilise un écran de 9'' monochrome. Le nouveau Mac utilisera le même. Il intégrera une carte mère carrée et un lecteur de disquettes.

Les premières ébauches datent de mars 81 et déjà, le concept final est proche.

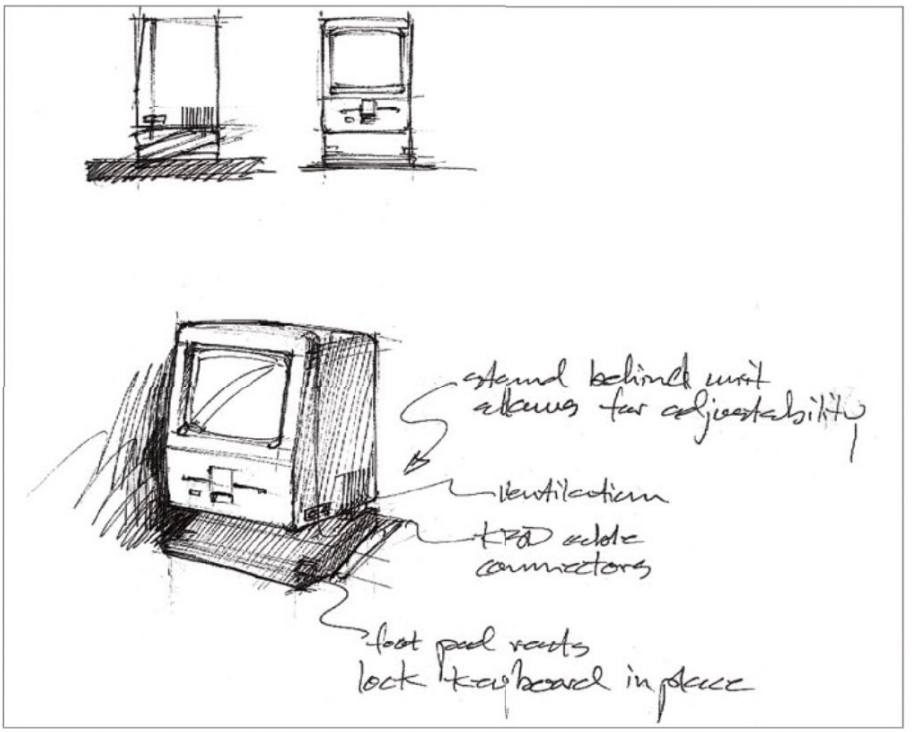


Prototype datant de 1981 avec un assemblage de hardware provenant de l'Apple II

La simplicité élégante s'impose visuellement (donc le boîtier), mais aussi à l'intérieur. Les cartes électroniques doivent être simples et élégantes. La carte mère doit être ordonnée et structurée avec une sérigraphie lisible et visuellement belle.

Dès le printemps, les prototypes peuvent intégrer les premières itérations du hardware. Fin de l'année, Jobs valide le design. Janvier et février 82 : les premiers prototypes complets sont construits : les Macintosh Twiggy. À cette époque, le projet intègre encore le lecteur 5 ¼ du Lisa. À l'été 83, après le fiasco du lecteur Twiggy, le lecteur 3 ½ de Sony est intégré : modification de la baie, du système et de la façade.

Une autre idée forte de Jobs mène les équipes : le 70/20. Le Macintosh doit offrir 70 % des fonctionnalités du Lisa pour 20 % du prix. Ce diktat complique le travail des designers et des ingénieurs. Par exemple, le Mac n'aura qu'un seul lecteur de disquettes ; l'interface utilisateur impose l'utilisation d'un processeur puissant, etc. Le 70/20 conforte l'idée d'une machine fermée et non extensible. Autre décision forte : tout doit se faire à la souris.



Jobs met régulièrement au défi les équipes. Avec le design Cuisinart, Jobs demande de nouvelles idées pour le Macintosh. Dessin d'avril 82. Photo : d'après Apple Design

Jobs ne veut pas de disque dur : trop cher, pas indispensable et nécessitant un ventilateur. Jobs n'aime pas le ventilateur : rien ne doit perturber l'utilisateur. Cette absence de ventilateur avait déjà été un cauchemar dans le design du Lisa. Et l'autre concept de base du boîtier : aucune vis apparente et le boîtier se divise en deux parties qui s'emboîtent parfaitement.

En parallèle, les designers doivent travailler sur le clavier et la souris. La souris du Lisa ne plaît pas ; il a fallu reprendre la conception : la forme générale, la surface du bouton, calculer l'inertie de la boule...

Tous les détails furent passés en revue, discutés, modifiés. Par exemple, le bouton d'alimentation est déplacé de l'avant à l'arrière. Une des raisons de cette modification est simple : réduire le temps de fabrication et limiter les fils à l'intérieur. Et cela avait un autre avantage : éviter d'éteindre l'ordinateur par mégarde.

Un important travail fut réalisé sur la connectique développée et adaptée par les ingénieurs. Pour faciliter l'utilisation de chaque connecteur, Susan Kare crée les icônes servant de signalétique.

Pour remercier les équipes des efforts fournis, Jobs fit graver les signatures des 47 membres de l'équipe Macintosh à l'intérieur du boîtier. Autre forme de reconnaissance que Jobs impose : les noms des concepteurs des logiciels sont indiqués dans le logiciel. Bill Atkinson avait été particulièrement déçu du manque de reconnaissance de l'équipe Lisa envers son travail. Jobs a voulu rattraper cette erreur.

2 / le design hardware

Le hardware du Macintosh se résume à une carte logique contenant les composants, le microprocesseur, etc., et une carte analogique pour la vidéo et l'alimentation. Et tout s'intègre dans le boîtier.

Le design hardware du Macintosh subit 5 modifications profondes. Plusieurs raisons expliquent ces changements : processeurs, composants, compacité de l'électronique, contrainte du boîtier. Une sixième évolution est faite à l'été 83. Mais l'électronique est grosso modo celle de l'été 82, 18 mois avant la sortie officielle. Dès cette date, la mémoire vive est fixée à 128 Ko. Non pour une question de place, mais pour une question de coût. L'électronique du Macintosh est un compromis sur le coût, les délais de finalisation et de production et les possibilités matérielles et logicielles de la machine.

À partir de la 2e version du hardware, la construction des prototypes s'accélère. Plusieurs dizaines sont construits à chaque étape, pour éprouver les concepts et les composants.

« À l'intérieur du Macintosh, matériel et logiciel travaillent ensemble pour offrir un système capable de supporter de hautes performances graphiques, de supporter les périphériques intégrés et les canaux de communication vers le monde extérieur » Burrell Smith (Byte, février 1984).

Cette intégration était indispensable pour concevoir une machine cohérente et créer un logiciel exploitant au mieux les possibilités (limitées) du Macintosh. Sans cela, il n'aurait jamais été possible de sortir un OS tenant sur une disquette de 400 Ko ni de définir le contenu des ROM.

La première itération hardware est finalisée fin 79. Les ingénieurs s'appuient sur les spécifications définies par Raskin, à savoir :

- Processeur 6809E
- 64 Ko de RAM
- Affichage de 256x256
- Affichage bitmap

Alors que le projet Lisa est déjà passé au Motorola 68000, un des processeurs les plus puissants de l'époque, le projet Macintosh reste au design de 79. Or, impossible de faire fonctionner les routines graphiques du Lisa sur le Macintosh. C'est en voyant le travail d'Atkinson sur le 68000 que le développeur Mac, Bud Tribble, commence à travailler de son côté sur une version 68000 de la carte mère.

Burrell, en charge du hardware, va réfléchir à un nouveau design incluant le 68000, tout en minimisant les changements et le coût des composants.

Ce design est prêt en décembre 80 :

- Processeur 68000 à 8 MHz (contre 5 MHz sur le Lisa)
- 64 Ko de RAM
- Affichage de 384x256
- Affichage bitmap

La 3e itération arrive au moment où l'équipe Mac doit booter directement sur le Macintosh et qu'un système Monitor est développé. Sur la partie purement hardware, un prototype avec un boîtier transparent est construit, incluant l'écran et un lecteur 5 ¼ provenant de l'Apple II. Ce design comprend l'utilisation d'un nouveau composant : le SCC. Il supporte le réseau local sans ajout de matériel supplémentaire. Mais il était gourmand en RAM.

Durant la seconde moitié de 1981, les équipes travaillent sur un nouveau composant personnalisé : le LSI. Cet élément sera surnommé Integrated Burrell Machine. C'est en 82 qu'un autre composant essentiel arrive sur le Mac : le contrôleur disque issu du fameux IWM, conçu par Steve Wozniak. Wendell Sander redésigne le composant. L'ambition de Burrell était de simplifier la carte mère et de pouvoir utiliser un seul composant quand cela est possible au lieu de plusieurs.

C'est au milieu de l'été 82 que le design matériel est fixé définitivement. Ce 5e design deviendra le Macintosh de janvier 84. Durant plusieurs mois, Burrell s'inquiète des capacités réelles du Macintosh et des besoins demandés par les logiciels. C'est à ce moment que l'ingénieur conçoit une carte mère avec 128 Ko de RAM. La fixation de la carte analogique contenant l'alimentation libère de la place sur la carte mère, facilitant l'ajout de nouvelles puces mémoire. Et grâce à ces modifications, l'affichage passe de 384x256 à la résolution finale du Mac : 512x342 !

Le dernier changement matériel majeur fut l'implé-

mentation du lecteur 3 ½, que Sony venait de sortir sur le marché. Jusqu'à présent, Macintosh utilisait le lecteur maison, le Twiggy. Son développement et sa production s'avèrent une catastrophe. En quelques semaines, les équipes modifient le design, les ROM et le système pour pouvoir utiliser les disquettes 3 ½. Cet ajout n'est pas anodin, car il prend moins de place dans le boîtier, obligeant à revoir une partie de l'organisation interne de la baie, même si, au départ, ce travail fut caché à Jobs.

Dans la version System du 15 août, le changement est terminé : implémentation du lecteur dans les ROM, changement visuel sur le Finder et gestion transparente de la disquette 3 ½. Quelques semaines plus tôt, la version du 5 juillet, Mac 16D0, affiche toujours la disquette 5 ¼, même si le lecteur Sony est en cours d'intégration : nous sommes sans doute encore en plein développement.

La carte verticale est conçue par George Crow et David Egner. Il rejoint l'équipe en janvier 1981. Elle contient l'alimentation et le circuit vidéo. Crow va aussi participer à l'intégration du lecteur Sony 3 ½. Crow est recruté spécialement pour concevoir cette carte. Dès son arrivée, Crow commence à designer la carte et les différents circuits. Sans elle, le Macintosh ne peut pas démarrer ni afficher la moindre image.

La dernière grande modification se fera avec l'intégration du composant IBM (Integrated Burrell Machine). Cette puce permettra d'afficher la résolution finale du Macintosh. À l'automne, la partie audio est terminée avec les dernières modifications.

Dès 83, une version améliorée est créée par Burrell Smith. Elle embarque 512 Ko de RAM. Techniquement, le changement nécessite relativement peu de modifications. Les ingénieurs savent les faire en quelques semaines. Le principal obstacle est le surcoût que ces nouvelles puces mémoire impliquent. Or, intégrer 512 Ko dans le Macintosh obligerait à augmenter (encore) le prix de vente. Jobs a conscience qu'un tarif élevé est un problème pour le succès du Mac.

Les premières cartes mères 512 Ko, MAC512, apparaissent courant 1982. En août, Burrell pousse pour intégrer 256 Ko de RAM avec la possibilité d'étendre à 512 Ko. Par design, le Macintosh peut supporter 512 Ko de RAM avec seulement 16 puces. Cette possibilité permettra de sortir le modèle 512 Ko en septembre 84.

3 / Le système

Le système du Macintosh repose sur une partie du développement réalisé sur le Lisa et sur les réflexions de l'équipe. Les premières démonstrations aux éditeurs se déroulent en 1981, alors que le système est à peine en développement, et que le Macintosh ne sait pas encore démarrer seul, sans l'aide d'un Lisa. Ce n'est que durant le 1er semestre 82, que les prototypes peuvent enfin booter seuls, via un environnement conçu par Rich Page.

Andy Hertzfeld développe les briques fondamentales du système logiciel. En 81, il conçoit les premiers pilotes matériels et les premières couches d'Entrées/Sorties du Mac. Ce travail est indispensable pour pouvoir communiquer avec les couches matérielles et commencer à utiliser le logiciel. En 82, tout s'accélère avec la User Interface Toolbox. À partir de là, les développeurs peuvent s'attaquer au système et aux logiciels. La toolbox inclut toutes les bases de l'interface et des interactions : fenêtres, boutons, menus, tous les éléments graphiques. En 83, il développe les premiers accessoires du System (album, tableaux de bord). Sans ce travail, aucun OS n'aurait pu être développé.

Andy développe la première version du bureau (Macintosh Desktop) en mars/avril 82. Cette couche permet de tester les routines graphiques et de l'optimiser au fur et à mesure. Comme le dit si bien Andy : le gestionnaire de fenêtres est un des éléments les plus importants de la User Interface Toolbox, et permet de montrer toute la puissance des régions définies par QuickDraw. Andy implémente, sur Macintosh, le code d'Atkinson pour le gestionnaire de fenêtres.

Le Macintosh impose l'idée du gestionnaire de fichiers, mieux que sur Lisa. Depuis janvier 84, sur Macintosh, nous utilisons le Finder. Cette application n'était pas présente dans le développement initial du System. Le gestionnaire de fichiers et le desktop sont absents dans les builds de 1981. Nous trouvons le Flounder. Il se présente sous la forme d'un écran avec l'image d'une disquette 5 ¼ et des boutons permettant d'utiliser les logiciels et le Macintosh : éjecter une disquette, imprimer, etc. Flounder commence son développement au printemps 82. Il est codé par Bruce Horn et Andy. L'idée d'un desktop natif, comme sur le Lisa, se pose au printemps et les développements démarrent vers mai – juin.

Une des premières versions intégrées de l'OS est le System Software 1.0 de février 1982, avec la première version de Flounder. Des développements du Lisa concernant le gestionnaire de fichiers furent cachés à Jobs pour éviter toute récupération au



Durant les premiers mois de 1983, une série de Macintosh Twiggy est construite. Ils doivent permettre de valider le boîtier et l'électronique. Ils sont envoyés aux équipes et aux éditeurs. Ces prototypes embarquent le lecteur 5 1/4 du Lisa, le Twiggy. Décision est prise, au printemps 83, d'intégrer le lecteur 3 1/2 de Sony. Ce changement oblige à changer la façade et à adapter la baie intérieure et l'électronique.
Photo : vente Bonhams

profit du Mac (voir Technosaures n°4). Bruce Horn développe le gestionnaire de ressources (Resource Manager), qui devient un des éléments essentiels de la toolbox. Ce travail est terminé à l'été 82. Les ressources sont des blocs de données utilisées par les applications et le système.

Jusqu'à l'été 83, le Flounder est en développement. La version 4.4 était incluse dans la version système Mac16D0 datant du 5 juillet 83. C'est durant cette période que le lecteur Sony 3 ½ est implémenté dans le hardware et que l'implémentation logicielle est réalisée.

À peine 4 semaines plus tard, le Flounder est remplacé par le Finder, qui s'inspire directement du Lisa Filer. Pour la première fois, nous trouvons, avec le Finder, la notion de desktop, les icônes des volumes, les dossiers, la corbeille, la barre de menu, etc. L'équipe logicielle unifie donc l'ensemble des développements, pour construire un OS complet. Tout n'est pas figé, mais les fondations sont là. Le menu Pomme est encore symbolisé par un @ et les boîtes d'alerte sont agrémentées de l'humoristique Steve Sez... Parallèlement, le développement de MacPaint et de MacWrite avance rapidement.

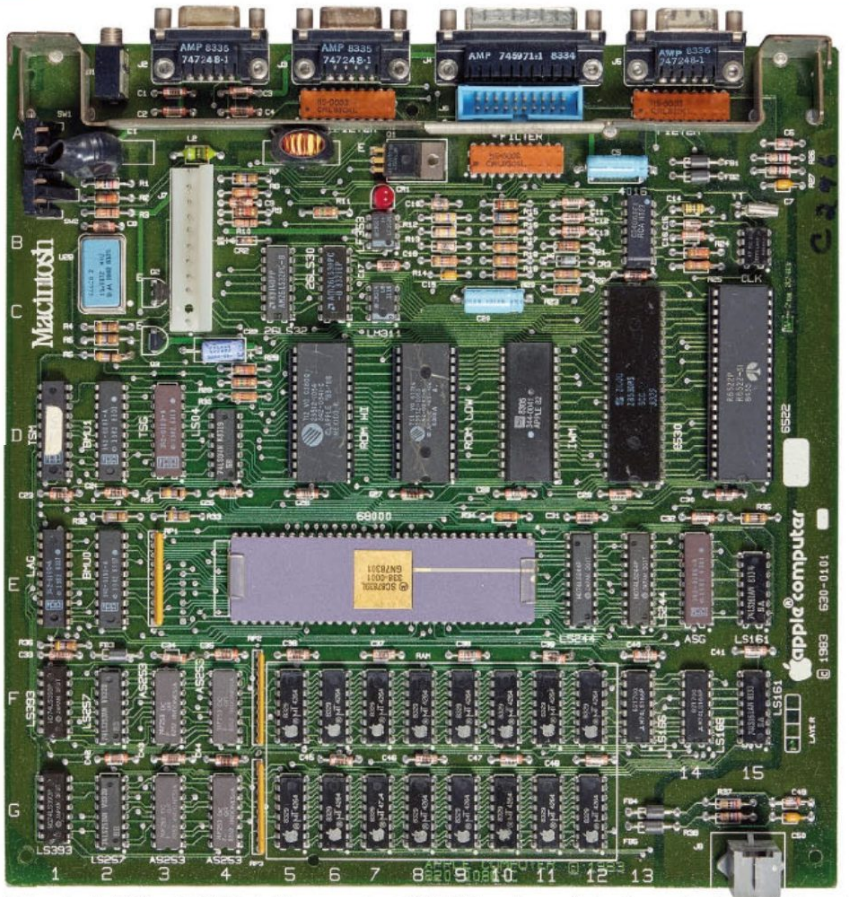
En septembre – octobre, le System rentre dans une phase de stabilité et de finalisation. Le Finder reçoit plusieurs évolutions : nouveaux menus, modifications sur la gestion des noms de fichiers, gestion des dossiers, glisser-déposer amélioré au niveau du Finder. Le menu Apple apparaît, mais les icônes système ne sont pas encore toutes disponibles.

Les versions System 1.0 et Finder 0.85 datent de décembre 83, à peine 4-5 semaines avant la présentation officielle. L'écran de démarrage, le Finder, le Guided Tour sont finalisés. Le menu Spécial apparaît définitivement, ainsi que l'ensemble des accessoires et les icônes validés. À noter que les versions de l'OS et du Finder évoluent différemment, d'où l'écart des versions.

Le 24 janvier 84, le Macintosh embarque le System 0.97 et le Finder 1.0. Ce Finder a été compilé le 18 janvier, pour avoir le temps de préparer les disquettes. Jusqu'aux dernières heures, les équipes ont corrigé les bugs les plus importants.

La version 1.1 apparaît fin avril, elle améliore énormément la vitesse de copie, le temps de démarrage ; et le redémarrage du Finder gagne aussi en vitesse.

L'un des plus grands défis fut de faire tenir tout le System et le Finder sur une seule disquette de 400 Ko ! La prouesse technique est remarquable.



Carte mère de 1983 : pré-série finale. L'agencement est définitif. Jobs exige une électronique ordonnée et belle à regarder. Cette directive n'a pas facilité le travail des ingénieurs. La carte mère, par rapport au Lisa, apparaît «simple» avec le strict minimum de composants.

Le Finder pesait seulement 46 Ko, contre plus de 350 Ko pour le Lisa Filer. On comprend qu'en janvier 84, l'équipe Macintosh était épuisée après plusieurs mois intenses, pour finaliser les logiciels nécessaires.

Les premières builds

- de mars 81 au printemps 83 : les versions du Mac System se succèdent même si les développements attendent les prototypes matériels du Macintosh. Tout d'accélère à partir de mars 83 avec des versions compilées chaque mois.
- juillet 83 (Mac16D0) : version avec Flounder, support du lecteur Sony
- 8 août : apparition du Finder
- 4 octobre : stabilisation de l'OS et du Finder
- 4 décembre (System 0.85) : version quasi finale du System

System 1.1 (5 mai 84)

La version 1.1 est une évolution importante de l'OS sorti en janvier. De nombreuses améliorations ont pu être intégrées :

- Amélioration de la gestion mémoire du Finder
- Vitesse de copie en hausse
- Optimisation système dont le démarrage
- Réaffichage du Finder plus rapide
- Modifications de certains menus et fenêtres
- Possibilité de lancer une application directement au démarrage

4 / Le problème du système de fichiers

Les concessions des ingénieurs et développeurs sont nombreuses : pas de disque dur, seulement 128 Ko de mémoire vive, un système qui doit tenir sur une seule disquette de 400 Ko ! Pour tenir les délais, les développeurs de l'OS doivent aller au plus simple : pas de gestion des noms longs, pas de système de fichiers hiérarchique — du moins, pas dans la version 1.0.

Le Lisa OS est un système multitâche, gérant les noms longs : chaque fichier possède son propre ID, des dossiers hiérarchiques, un support natif du disque dur. Le Lisa possède un disque dur et surtout 1 Mo de RAM !

Le système de fichiers initial est le Macintosh File System (MFS). Le développement débute à l'automne 82. Il est entièrement réécrit en décembre pour supporter les resource forks et le gestionnaire de ressources. Il fallait aussi intégrer la gestion des labels. Pour simplifier la gestion dans le System, l'annulation de suppression fut abandonnée. D'autres changements apparaissent début 83.

L'implémentation du lecteur de disquettes Sony oblige à une nouvelle réécriture du MFS pour sup-

porter le stockage de 400 Ko. Pour l'anecdote, la gestion des disquettes 800 Ko (double face) était déjà implémentée à l'été 83, alors que le lecteur 800 Ko ne sera intégré qu'avec le Mac Plus (1986).

Le MFS repose sur un adressage plat non hiérarchique. Il gère un seul niveau de dossiers et supporte en interne 255 caractères pour la longueur des noms de fichiers, mais le Finder n'en supportait que 63.

MFS est, avant tout, conçu pour gérer les fichiers sur les disquettes et non un disque dur.

Quelques spécificités de MFS :

- Une taille de volume de 20 Mo
- Une taille de fichiers de 20 Mo maximum
- 4094 fichiers par volume

MFS fut arrêté en septembre 85 quand HFS sort. Cependant, les volumes sont toujours utilisables jusqu'au System 7.6.1. À partir de Mac OS 8, MFS est retiré.

Structure de MFS :

- Volume (avec un nom)
- Blocs logiques de 512 octets
- Chaque volume contient une carte
- d'allocation des blocs du volume

Chaque fichier stocké sur le volume est identifié par son nom et son numéro de version. Le nom n'est pas sensible à la casse (minuscules/majuscules). Le File Manager est stocké directement en ROM. Un fichier se compose de deux éléments : le data fork (les données du fichier) et le resource fork.

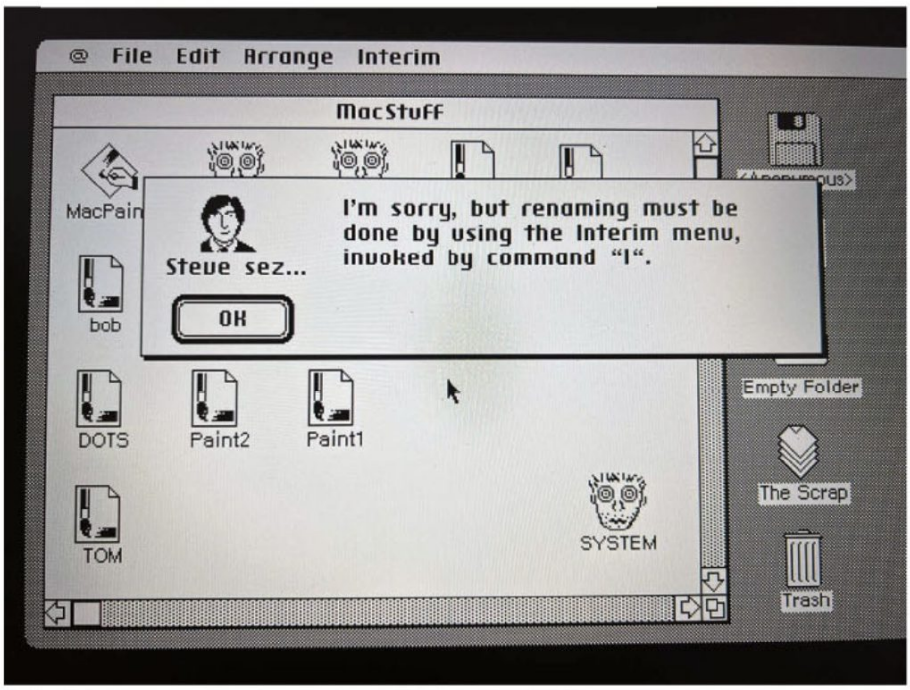
Sur les premières versions du System, la commande Nouveau dossier n'existe pas. Seul un dossier sans titre est affiché. Si on change le nom ou qu'il est dupliqué, un nouveau dossier vide est créé. Mais cela reste fondamentalement un système de fichiers non hiérarchique.

Il faut attendre le System 2.0 (avril 85) pour que la commande Nouveau dossier apparaisse.

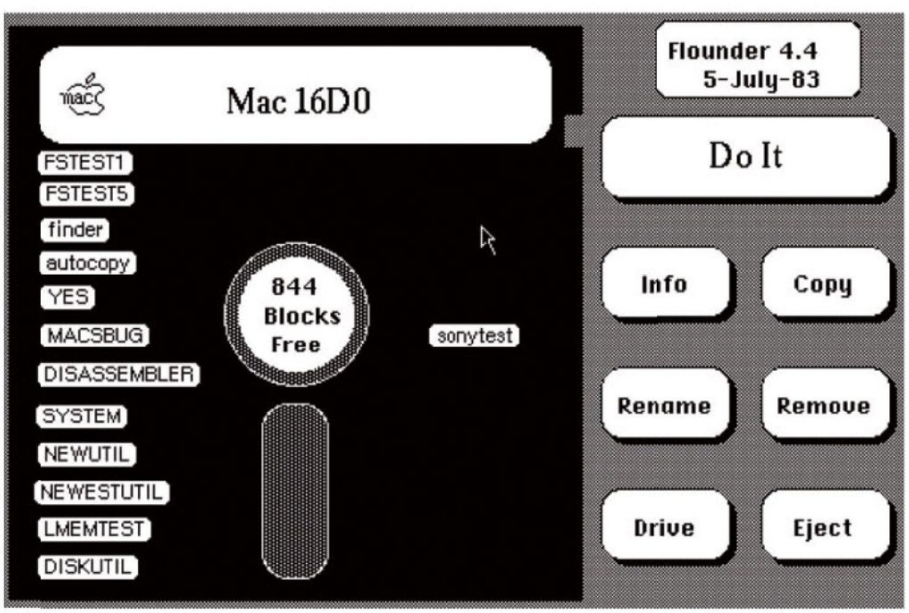
5 / Au cœur du système : le copier-coller et le presse-papier

La difficulté du copier-coller, et du couper, est de garder les informations (textes, graphiques, etc.) dans un espace mémoire temporaire, le presse-papiers. Cette fonction sert au sein du même logiciel ou entre deux logiciels. Le presse-papiers est donc un élément essentiel du copier-coller. Tout se gère en mémoire. Or, plus les données à copier sont volumineuses, plus elles nécessitent de la mémoire. Ce tampon mémoire ne doit pas disparaître quand le logiciel destinataire est chargé et le document créé. Car n'oublions pas que le système ne permet pas d'exécuter plusieurs logiciels simultanément.

Une fois la donnée collée, il faut vider le presse-papiers et la mémoire tampon utilisée. Cette allocation–désallocation a provoqué des instabilités du copier-coller au moment de fixer la ROM. Tout est géré dans la Toolbox via le Scrap Manager.



Version d'août 83 avec les fameux Steve sez et le menu @. Photo : François Tonic



Founder version du 5 juillet 1983. © F.T.

L'autre difficulté du presse-papiers est de contenir des données variées : textes, données (type tableur), graphiques, dessins.

Les limitations mémoire et l'absence de multitâche interdisent d'implémenter le multi-presse-papiers ou encore de gérer un historique. Il arrivait souvent que le System alerte : mémoire insuffisante pour copier l'élément ou encore pour le coller, notamment car l'élément copié n'était pas compatible avec le document destinataire…

« Les programmes du Macintosh, s'ils sont bien conçus, seront capables d'échanger des données entre eux sans qu'un programme ait à connaître la nature des autres. Cela se fait grâce à l'utilisation d'une zone mémoire partagée appelée presse-papiers et à la standardisation des données qui peuvent y être stockées.

Le presse-papiers est un élément mémoire qui n'est pas effacé quand un nouveau programme remplace le programme actif. Il est utilisé comme suit : le 1er programme copie les données dans le presse-papiers, le 2e programme remplace le 1er programme par son propre code et ses données. Les données du presse-papiers sont ajoutées aux données du 2e programme.

Le presse-papiers peut contenir un nombre variable d'éléments de données, bien que chaque élément doive être d'un type différent. Chaque élé-

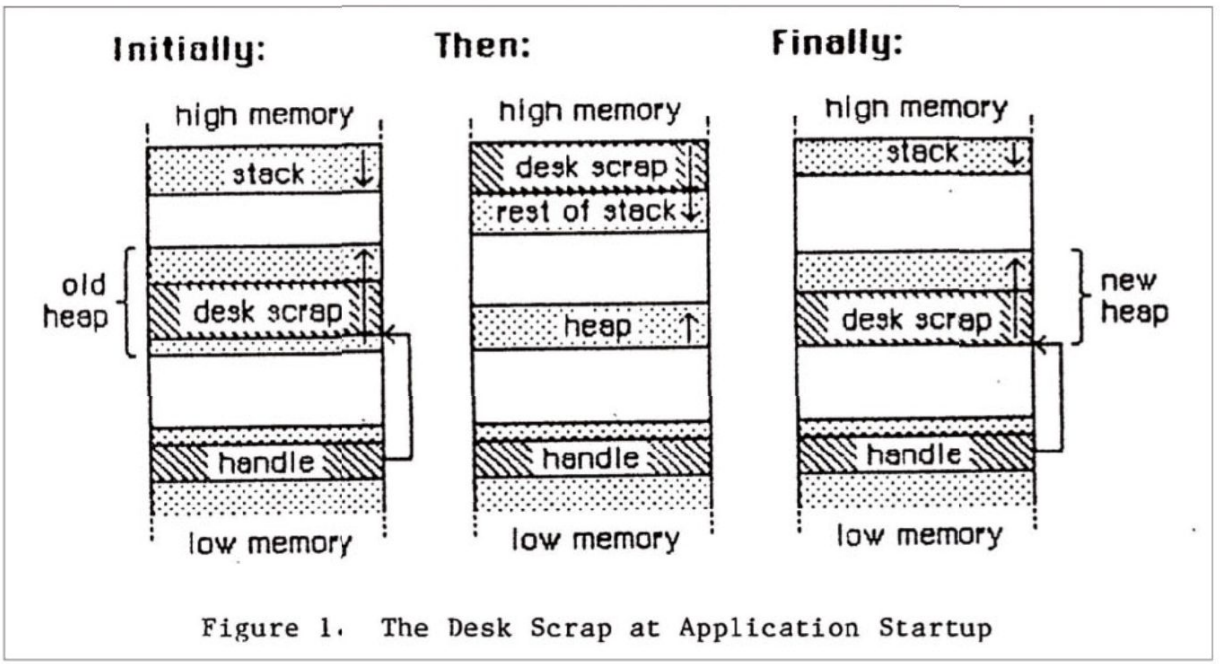


Figure 1. The Desk Scrap at Application Startup

Fonctionnement du copier-coller dans la pile mémoire

ment de données se compose d'un identifiant de type de données à 4 caractères, d'une longueur de 32 bits (en octets) et d'un flux d'octets qui constitue le contenu réel à copier et à coller. Les programmes sont libres d'implémenter leur propre type de données. Apple en a défini deux : le texte et les images QuickDraw. Apple encourage chaque programme à être capable d'écrire et de lire au moins l'un des deux types.

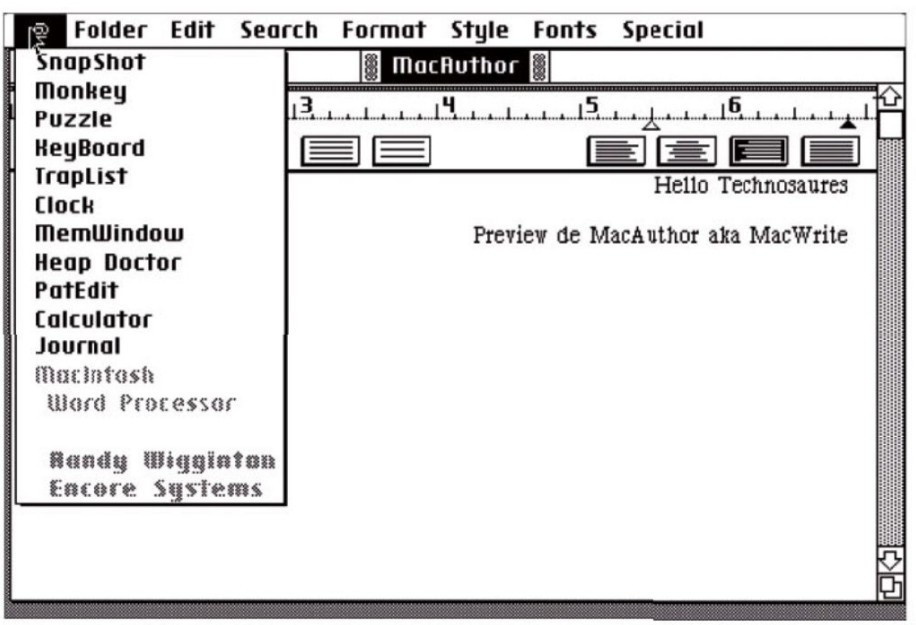
Le texte est une simple chaîne ASCII sans les informations de police, de taille ou de position. Les images QuickDraw sont définies comme une séquence de commandes pouvant être comprises par les routines QuickDraw. Une image QuickDraw peut contenir du texte (incluant la taille, la police, la position), une séquence de commandes graphiques élémentaires qui recréeront l'image ou l'image décrite comme un flux d'octets. Ces deux types de données constituent un moyen de communication garanti entre les programmes Macintosh ». (The Apple Macintosh Computer (Greg Williams, Byte, février 1984))

La fonction « couper » copie la donnée et l'efface du document d'origine. Le copier-coller garde la donnée dans le document. Cela oblige une double programmation dans la Toolbox et une gestion particulière de la mémoire.

6/ MacWrite : une killer-app

MacWrite est l'un des logiciels les plus marquants du Mac et reprend les concepts du traitement de texte définis, dans les années 70, par les développeurs du Xerox PARC pour l'Alto. Randy Wigginton en est le principal développeur.

En 81, il quitte Apple et crée Encore. Il est appelé par l'équipe d'Apple pour les aider à développer le futur traitement de texte. Le projet changea plu-



Le futur MacWrite

sieurs fois de nom : MacAuthor, Macintosh WP. Rardy travaille de manière hybride pour Apple et Encore. Pour s'assurer les services de Randy, un contrat est signé le 12 juillet 1982 : un Computer Software Development Agreement.

MacWrite surmonte les contraintes de la RAM en stockant tout le document en RAM, ce qui ne permetait pas de créer des documents trop longs. En décembre 83, MacWrite souffre toujours de nombreux bugs et d'une instabilité chronique, mais les principales fonctionnalités sont implémentées. Les développeurs ont eu à peine un mois pour corriger les principaux bugs et pour pouvoir créer les disquettes dès la mi-janvier, soit 10 jours avant la présentation du Macintosh.

MacWrite possède un avantage sur LisaWrite : une véritable barre d'outils pour faciliter l'ajustement des règles et le formatage des textes (justification, interlignage).

MacWrite exigeait toute la mémoire disponible possible, soit 90 Ko sur les 128 Ko du Mac. Tout comme MacPaint, MacWrite devient rapidement un logiciel bureautique incontournable sur Mac, et il sera livré durant plusieurs années avec chaque Mac.

Parmi les fonctions, nous trouvons :

- Justification des textes
- La règle pour ajuster les documents
- Fonction recherche et remplacement
- Les menus police et style

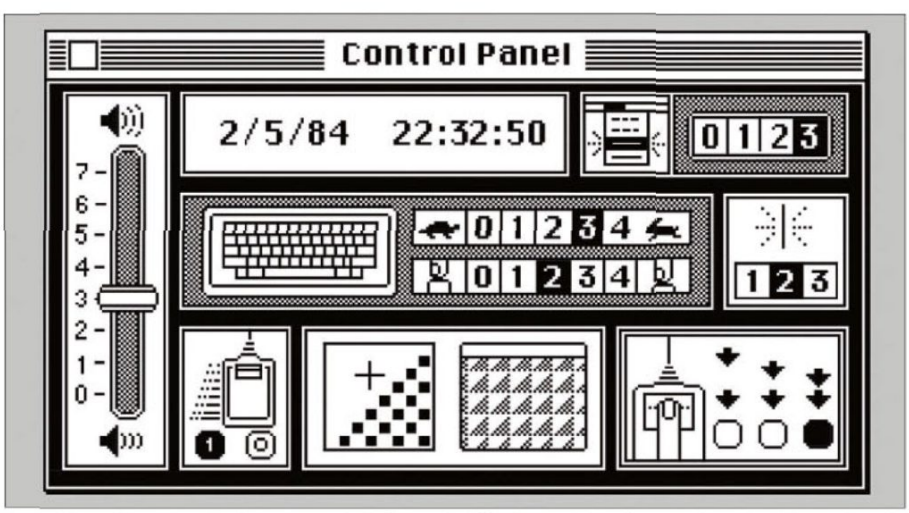
Susan Kare invente l'âme de l'ordinateur



Une des plus célèbres photos de Susan dans son bureau. 1984. © Norman Seeff

L'équipe du Macintosh était sans doute l'une des meilleures d'Apple. Jobs n'hésitait pas à chercher ailleurs les profils qui pouvaient aider au développement du Macintosh. Soit le co-fondateur les trouvait lui-même, soit un membre de l'équipe faisait le lien. C'est le cas avec Susan Kare. Elle va jouer un rôle crucial dans la représentation visuelle de l'interface. L'équipe connaissait les faiblesses de l'interface du Lisa : complexe, lourde et sans chaleur. Mais comment donner une « âme » au Macintosh ?

Début 1982, Andy Herzfeld demande à Susan de créer des icônes et des polices de caractères. Le défi est immense : il faut tout définir, tout créer, tout imaginer. Jobs veut humaniser l'informatique, et donc le Macintosh. Le Lisa apparaît comme trop froid. À cette date, de nombreux détails de l'interface sont déjà créés ou repris du Lisa. Susan va alors créer, au fur et à mesure, les icônes des logiciels, du système, et donner une cohérence à l'ensemble. L'enjeu est de faire comprendre immédiatement la signification de chaque icône.



Susan définit et designe un des éléments essentiels du System de 1984 : le Control Panel pour configurer des éléments clés tels que la date, l'heure, le volume sonore, la vitesse du clic.

Mr Macintosh et les premiers dessins (février 82)

Andy raconte qu'un soir, Jobs parlait de Mr Macintosh. La possibilité d'embaucher l'artiste Folon avait été évoquée. Andy contacte une camarade de classe, Susan Kare, et lui demande si elle pourrait imaginer des animations autour de ce mystérieux Mr Macintosh.

Elle accepte de travailler sur Mr Macintosh, les icônes et les polices de caractères. Son premier travail, elle le fait en échange d'un Apple II ! Elle rejoint officiellement l'équipe en janvier 1983. Travaillant pixel par pixel, son premier outil est une grille de 32 × 32, la taille d'une icône. Là, elle commence à imaginer et à dessiner chaque carré. Elle sait ce qu'elle veut créer : des images claires et expressives, sans pour autant gêner l'utilisateur. Susan dessine le fameux Mac qui sourit au démarrage, à la demande de Jobs. Dans les premières semaines de 1983, les développeurs, dont Herzfeld et Atkinson, adaptent et créent des outils de création : Icon Editor et Font Editor, en plus de MacPaint.

Dès son arrivée chez Apple, l'équipe lui demande

FÉVRIER 1982 : UN AUTRE NOM QUE MACINTOSH ?

Steve a longtemps hésité sur le nom final du Macintosh : Macintosh ou autre chose ? Un soir, il revient voir l'équipe avec une nouvelle idée : Mr Macintosh. Impossible cependant de rajouter une image ou une animation de ce Mr Macintosh, ni d'utiliser MrMacHook, un menu spécialement développé par Herzfeld. L'idée sera rapidement abandonnée. Il était aussi question de changer le nom de code du projet : Bicycle. Le nom Macintosh sera définitivement choisi comme nom pour la machine en janvier 1983.

aussi de créer une police de caractères pour le système. Ce travail aboutit à la fonte Elefont, qui prendra le nom définitif de Chicago. À partir de là, Jobs utilise des noms de grandes villes (Genève, Monaco, etc.). C'est une étape importante dans la cohérence de l'expérience utilisateur.

Comme le dira Susan dans une interview en 2019 : « La première police que j'ai dessinée est Chicago, parce que nous avions besoin d'une police système en style gras. » À la sortie du Mac, plusieurs polices sont intégrées par défaut dans le système.

Jobs va approuver chaque icône, chaque élément dessiné par Susan. Elle rejoint pleinement la vision de Jobs sur l'ordinateur et sa passion pour la calligraphie.

1983 : définir l'environnement graphique et visuel du Mac

Début 1983, aucun logiciel ne permet à Susan de créer facilement des icônes (32 × 32) et des fontes. La création des icônes est un défi permanent : limitation de la taille, nécessité de trouver une représentation lisible et compréhensible. Il faut définir un code graphique et visuel pour les icônes système : la corbeille, l'icône des disques, la représentation d'un dossier, d'un nouveau document. Le Lisa possédait déjà un ensemble d'icônes, mais elles ne furent pas reprises in extenso dans le Macintosh.

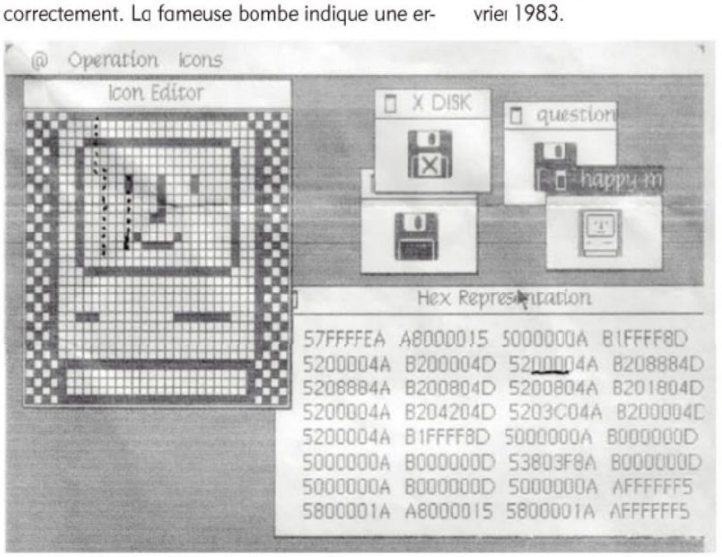


Le fameux sad Mac.

Susan part de zéro pour tout repenser et apporter sa vision d'artiste et de dessinatrice. Objectif : rendre l'ordinateur et le logiciel plus familiers, plus accessibles. La vision du développeur et de l'ingénieur n'était pas toujours compatible avec la philosophie de Jobs sur le projet Macintosh : le Mac est créé par des artistes.

Le Mac souriant du démarrage indique que tout va bien et que l'ordinateur et le système démarrent correctement. La fameuse bombe indique une erreur grave. Le Sad Mac symbolise un problème matériel ou système, accompagné d'un code d'erreur spécifique.

Pour aider Susan, Andy Herzfeld crée un logiciel dédé : Icon Editor. Ainsi, Susan peut créer et dessiner directement sur le Mac, et non plus sur papier. Ce fut une étape importante dans la définition de l'interface. Herzfeld travaille sur cet éditeur en février 1983.



L'éditeur d'icône d'Andy


Le drapeau pirate de Susan

SUSAN PEINT LE MYTHIQUE DRAPEAU PIRATE

Durant une retraite de l'équipe en janvier 1983, Steve avait déclaré qu'il valait mieux être un pirate que rejoindre la marine. Il encourageait l'esprit rebelle. En août, toute l'équipe emménage dans des locaux plus grands au Bandley 3. La « division » Macintosh rassemble alors plus de 80 personnes…

L'un des développeurs clés, Steve Capps, estime que si l'équipe du Macintosh est une bande de pirates, il doit y avoir un drapeau pirate flottant en haut du bâtiment. Peu après l'installation au Bandley 3 Capps arrive avec un morceau de tissu noir et demande à Susan de peindre l'emblème pirate, avec le logo Apple sur un œil.

Une petite expédition est menée pour l'installer. Quelques semaines plus tard, le drapeau disparaît. Après une rapide enquête, il apparaît que l'équipe du Lisa avait décidé de le voler. Il retrouve rapidement sa place. Il restera plus d'un an avant de disparaître de nouveau… pour de bon.

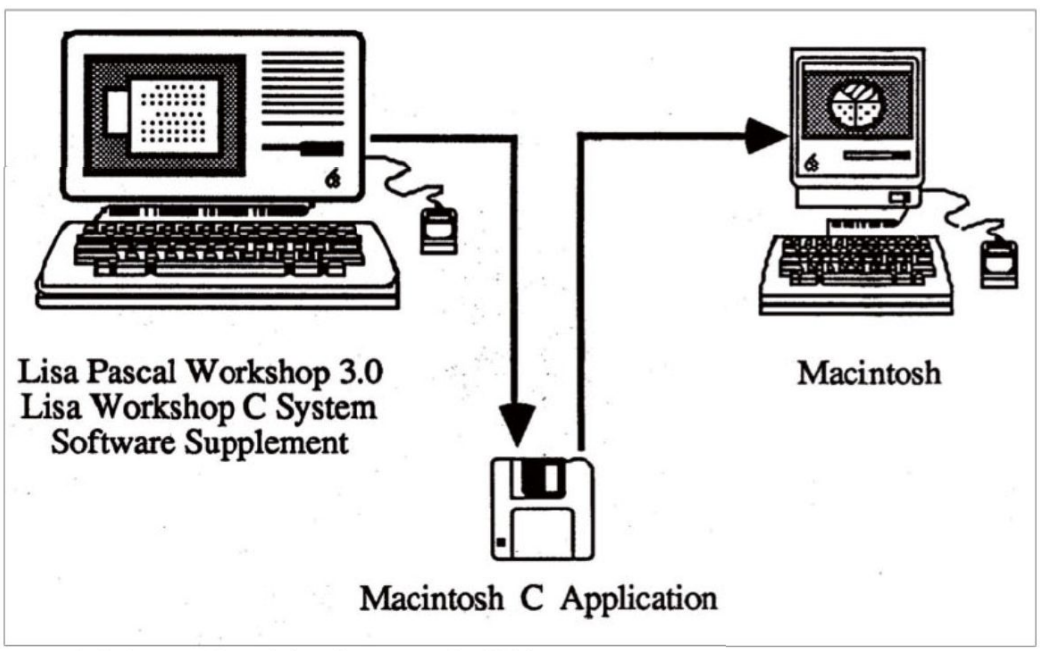
Il s'inspire du travail d'Atkinson sur le mode Fat Bits de MacPaint. Durant toute l'année, Susan crée des dizaines d'icônes parmi les plus connues du Macintosh : Happy Mac, la bombe, les icônes d'application, de document, d'imprimante, de police, la touche Command, etc. Elle forge ainsi l'identité visuelle de l'interface, qui influence encore aujourd'hui les interfaces graphiques des systèmes d'exploitation. Chaque icône est le fruit d'une réflexion : pourquoi utiliser une page avec un coin plié ? Dans la fenêtre de contrôle, comment représenter la rapidité du double-clic ? Ensuite, les développeurs intègrent les icônes en hexadécimal dans la ROM.

Susan travaille avec rigueur et recherche toujours la cohérence entre les icônes. Il en va de même pour les polices : cohérence et uniformité selon les styles. Elle crée également des visuels pour la documentation et les brochures, et intervient régulièrement dans les médias.

L'une de ses réalisations les plus emblématiques est DogCow, alias Clarus, ou encore Moof. Cet étrange animal apparaît courant 1983. DogCow est inclus dans la police Cairo et apparaît sur la page d'impression de la LaserWriter. Il devient rapidement une mascotte des développeurs Apple.

Quels modèles de développements pour les logiciels Macintosh ?

Steve Jobs sait qu'une des clés du succès est le logiciel. Sans logiciel, le Macintosh est une coquille vide. C'était l'erreur de l'équipe Lisa et de l'Apple III. L'Apple II est un succès grâce aux logiciels. *Très tôt dans la conception du Macintosh*, dès 1981, il n'hésite pas à aller voir les éditeurs avec un prototype. Dès que les prototypes sont complets, ils sont envoyés aux éditeurs. En parallèle, l'équipe doit fixer le modèle de développement, les outils nécessaires et rédiger la documentation technique.



Le développement Macintosh depuis l'environnement Lisa Workshop.

Paradoxalement, pour développer sur Macintosh, il faut utiliser un Lisa 2. Durant la conception de la machine, il n'est pas possible de développer directement sur la machine. Cette limitation se poursuit jusqu'à la sortie du modèle 512 Ko. La nécessité de passer par le Lisa sera définitivement supprimée en 1985, avec la disponibilité des environnements de programmation et des compilateurs.

Les limitations importantes du Macintosh 128 Ko ne permettent pas de le transformer en poste de programmation. Parmi celles-ci : le manque de mémoire vive et de stockage de masse, ainsi que l'absence de multitâche coopératif, il fallait quitter les logiciels en cours pour pouvoir exécuter un autre logiciel. Au printemps 85, le Mac 512 Ko et l'apparition du logiciel Switcher (permettant de basculer d'un logiciel à un autre sans quitter le Finder), rendent enfin le Macintosh indépendant du Lisa. D'autre part, les premiers environnements de développement sont exploitables sur un modèle 512 Ko !

Cependant, il était possible d'utiliser un éditeur de code sur le Macintosh malgré les limitations. Microsoft sort son éditeur Basic dès mars 84 (voir section sur le Basic).

La Toolbox : l'élément clé pour l'interface graphique

En 81, Andy Hertzfeld conçoit les premiers pilotes matériels et les premières couches d'Entrées/Sorties du Mac. Ce travail est indispensable pour pouvoir communiquer avec les couches matérielles et commencer à utiliser le logiciel. En 82, tout s'accélère avec la User Interface Toolbox. À partir de là, les développeurs peuvent s'attaquer au système et aux logiciels. La toolbox inclut toutes les routines de bases de l'interface et des interactions : fenêtres, boutons, menus, tous les éléments graphiques.

Par rapport aux démos faites à Microsoft à l'été 81, le système commence à se stabiliser. Andy développe la première version du bureau (Macintosh Desktop) en mars/avril 82. Cette couche permet de tester les routines graphiques et de les optimiser au fur et à mesure. Comme le dit si bien Andy : le gestionnaire de fenêtres est un des éléments les plus importants de la User Interface Toolbox et permet de montrer toute la puissance des régions définies par QuickDraw.

Andy implémente, sur Macintosh, le code de Bill Atkinson pour créer le gestionnaire de fenêtres, initialement conçu pour le système Lisa. Écrit en Pascal, langage historique d'Apple, Andy le réécrit en assembleur 68000, même si le portage initial est en Pascal. Une démo est même créée en février/mars, avec des balles qui rebondissent. Susan Kare crée des bitmaps 16 × 16 pour la démo.

La Toolbox est un des éléments clés des ROM du Macintosh. L'iMac (1998) introduit une nouvelle ROM et la Toolbox en est retirée. Elle est désormais sous la forme d'un fichier ROM directement accessible dans le dossier système. Ce fichier est chargé en RAM par le firmware au démarrage de la machine. Avec Mac OS X, la Toolbox n'est plus utilisée. Elle l'est uniquement dans l'environnement Classic.

L'environnement de développement

Pour développer des logiciels pour le Macintosh, il faut disposer de l'environnement suivant :

- 1 Macintosh complet
- 1 programmer switch (à installer sur le côté pour accéder aux boutons internes Reset et Interrupt)
- 1 Lisa 2 avec Lisa Workshop (environnement de développement)
- Des disquettes 3 ½
- Des disquettes 5 ¼ (Lisa)
- Les câbles pour relier le Lisa et le Mac

Le Lisa Workshop est un environnement de développement graphique, comprenant un éditeur de code, une ligne de commande, un compilateur Pascal, un assembleur 68000, un linker et l'outil MacCom, pour assurer la communication entre le Lisa et le Mac.

À cela, s'ajoute le Lisa Workshop Supplement. Il s'agit d'un complément au Lisa Workshop, avec sa documentation. Il inclut des outils, des librairies et des exemples de code. La version de février 85 se compose de 12 disquettes. Le complément évolue régulièrement. Il y a des outils côté Mac et côté Lisa. La dernière version date de 1986.

Le Lisa et le Mac utilisent le même processeur, le Motorola 68000. Le Pascal et l'assembleur sont les deux langages les plus utilisés. Il était donc relativement facile de concevoir un environnement multi-plate-forme. Le Lisa s'occupe de tout : éditeur de code, création des fichiers ressources pour le Macintosh, génération du code 68000, linker (entre le code et le compilateur Pascal), création du binaire exécutable Mac puis transport du fichier sur le Mac. Le logiciel pouvait s'exécuter sur le Lisa avec MacWorks.

Le processus de développement reste relativement facile pour les développeurs. Inconvénient : nécessité d'acheter 2 machines et l'environnement de développement Lisa.



L'assembleur était des langages proposés par Apple avec son Macintosh 68000 Development System

Macintosh Development System (1985)

Apple travaille sur un environnement dédié sur Macintosh : le Macintosh Development System (MDS). Il s'agit du 1er véritable environnement de programmation natif sur Macintosh. Il contient un éditeur, un éditeur assembleur, un linker pour la compilation, un compilateur, un outil d'exécution du code et un debugger. Il permettait de coder en Assembleur 68000. Il est livré sur 2 disquettes, avec un câble.

L'outil n'est pas développé directement par Apple, mais par Consulair. Bill Duvall en est le créateur. Il reçoit un prototype du Macintosh dès le printemps 83.

La version 2 supporte HFS, installation sur disque dur, éditeur avec fonction annulation, support des nouvelles ROM. L'environnement disparaît du catalogue d'Apple en 1987.

Macintosh Programmer Workshop (1985-86)

Dès fin 84, le Macintosh Programmer Workshop (MPW) fait parler de lui. L'équipe Macintosh veut proposer un environnement de développement complet et indépendant du Lisa, dont l'usage doit s'arrêter le plus tôt possible. Les équipes commencent à utiliser intensivement MPW fin 85. Mais ce n'est qu'en 86 que le MPW est officiellement commercialisé.

L'environnement doit :

- Fournir un environnement natif et intégré au Macintosh et supporter l'assembleur, le Pascal et le C
- Simplifier et améliorer le processus de développement et de compilation
- Environnement ouvert à des extensions externes

Les fondamentaux du développeur Macintosh

Les équipes Macintosh font bien les choses. Très tôt, la documentation technique est rédigée et suit l'évolution du projet. La Toolbox et les routines du système sont les fondamentaux, que tout développeur Mac doit connaître. Ces composants permettent d'accéder à l'ensemble des fonctionnalités du Mac.

Si on résume :

- Toolbox : ensemble des API et fonctions accessibles aux développeurs. Tout est documenté.
- QuickDraw : les routines graphiques que les développeurs utilisent pour créer les interfaces
- Inside Macintosh : documentation technique officielle et incontournable. La première version remonte à mars 82.

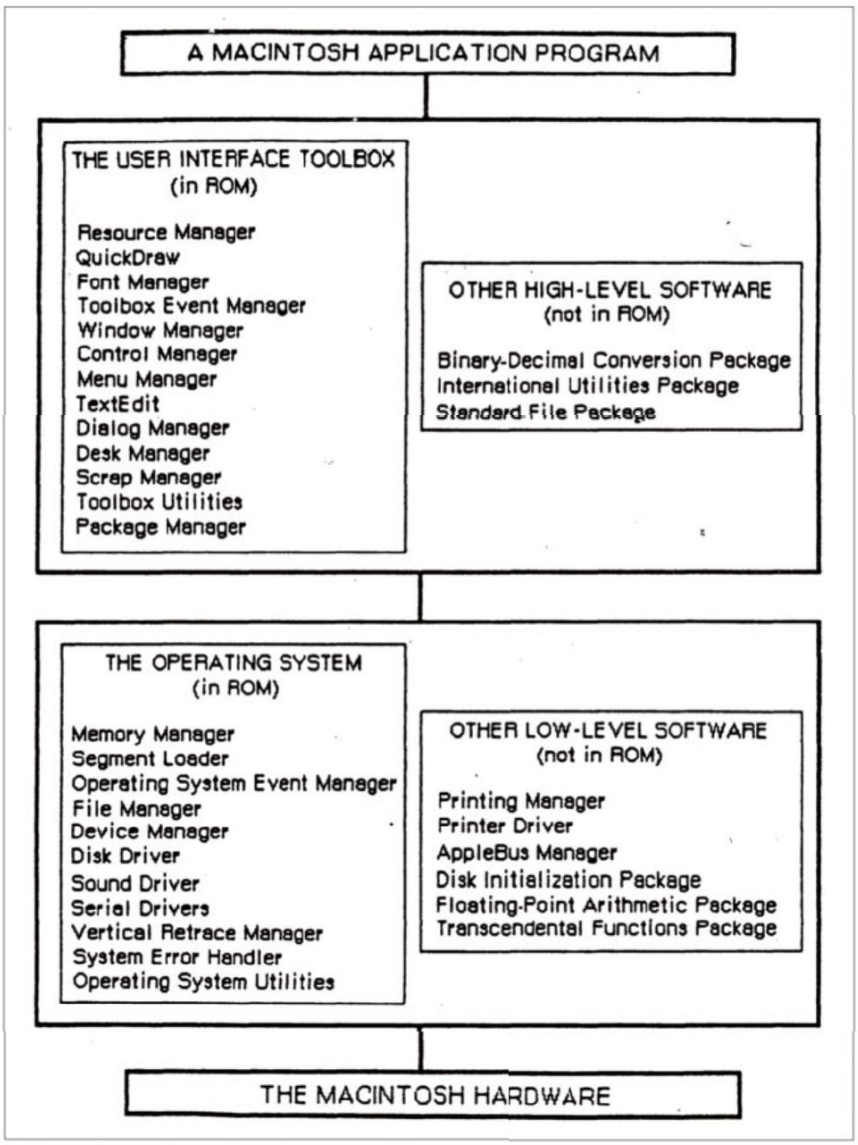
La documentation est initiée en 82 par Joanna Hoffman et Chris Espinosa. Elle est supervisée par

MACSBUG (1982)

Pour le debug bas niveau sur le processeur 68000, il faut un outil lui-même bas niveau. Il s'agit de MacsBug. Contrairement à ce que l'on pourrait penser, ce debugger a été initialement développé par Motorola. Il s'agit du Motorola Advanced Computer System Debugger. Advanced Computer System était le nom de code du 68000. Le nom restera.

Apple prend une licence en janvier 82, pour l'adapter aux besoins des développeurs Lisa puis Macintosh. Rich Page conçoit les premières versions (LisaBug, MacsBug). Le debugger évolue avec les développements des logiciels et des besoins des équipes.

En juillet 84, la fenêtre de debug est directement visible sur l'écran (10 lignes) et non dans une fenêtre séparée. Objectif : simplifier la visualisation des erreurs et interagir plus vite avec le système. Au départ, Macsbug fonctionne en mode terminal sur Macintosh, car il fallait une autre machine, un Lisa, pour pouvoir debugger et voir les erreurs. En 1985, Apple intègre MicroBug, moins puissant que MacsBug.



Architecture d'une application Macintosh typique. Inside Macintosh.

Caroline Rose, qui est embauchée à l'été 82. Reconnue par sa qualité et sa richesse, Inside Macintosh n'est pas toujours facile à manipuler ni à comprendre.

Autre fondamental : les User Interface Guidelines. Cette documentation est LE livre de référence pour comprendre l'interface graphique et comprendre le moindre composant, ce qui est autorisé ou pas, etc. L'autre référence incontournable est le Human Interface Guidelines.

Un autre environnement, moins connu, fut développé à partir de février 1985 : MacApp. Son créateur fut Larry Tesler, un des ingénieurs clés du Lisa puis du Macintosh. MacApp a été conçu pour « simplifier » l'accès aux différentes couches logicielles du Mac (Toolbox, QuickDraw, etc.). C'est une librairie orientée objet, utilisant le Pascal Objet. Même si MacApp ne rencontra pas le succès attendu, plusieurs logiciels phares furent développés dont Adobe Photoshop !

EMATE : L'HYBRIDE PORTABLE ET TABLETTE (1997)



L'eMate est un hybride entre un portable et le MessagePad. Il se destine avant tout au marché de l'éducation. C'est le seul matériel Newton à intégrer par défaut un clavier. Son design est novateur pour Apple : tout en rondeur, translucide. Il fut dessiné par Jonathan Ive.

L'eMate était audacieux pour son concept et l'OS utilisé. Quand Jobs reprend la direction d'Apple, à l'été 97, il doit faire des choix et le MessagePad est arrêté. Jobs évoque clairement son intérêt pour l'eMate, sur les usages et le design. Mais sa mission immédiate est de sauver Apple et malheureusement ni le MessagePad ni l'eMate ne génèrent suffisamment de ventes et il arrête ces projets.

Le succès fut limité et son potentiel ne fut jamais exploité : Apple est en crise, rachat de NeXT et retour de Jobs qui recentre les équipes et coupe des dizaines de projets. À la sortie de l'eMate, l'accueil avait été plutôt bon. Plusieurs déclinaisons de couleurs étaient prévues, mais seul le vert sortit.

ABONNEMENT NUMÉRIQUE
1 an ..... 39 €
Abonnez-vous sur :
www.programmez.com

FAITES VOTRE VEILLE TECHNOLOGIQUE AVEC
programmez!
Le magazine des dév - CTO - Tech Lead

ABONNEMENT PAPIER
1 an ..... 55 €
2 ans ..... 90 €
Voir page 5

Macintosh Plus : le modèle qui impose le Mac (1986)

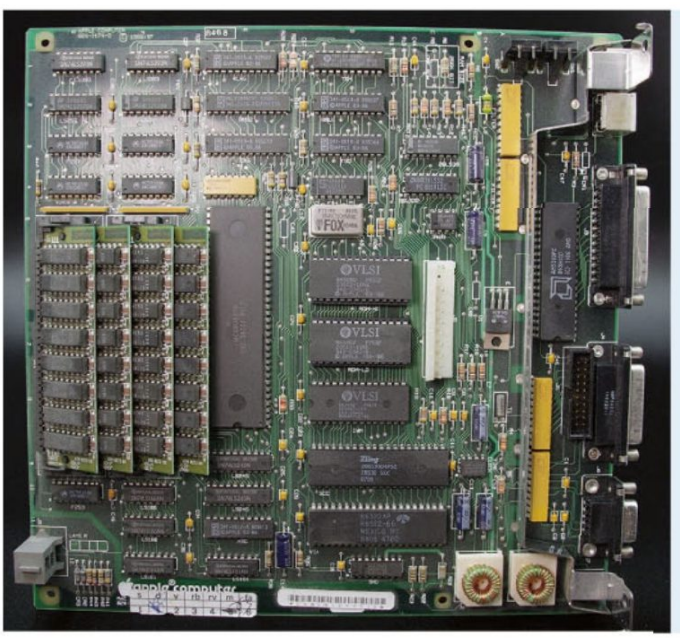
En janvier 1986, le Macintosh Plus arrive enfin sur le marché. Il reprend le design originel, mais propose enfin des caractéristiques à la hauteur des ambitions d'Apple. La machine embarque par défaut 1 Mo de RAM, un lecteur de disquette 800 Ko (contre 400 Ko auparavant) et un port SCSI pour les disques durs. Le Mac Plus permet enfin d'ajouter des barrettes sur la carte mère : jusqu'à 4 Mo. Le Mac Plus est aussi le premier modèle à sortir après le départ de Jobs.

Après la sortie du Macintosh 512 Ko, aucun modèle se réussit à dépasser le stade de prototype. Turbo Mac et Big Mac sont les deux prototypes les plus importants (1984-85). Le projet Mister T (1985) aboutit au Macintosh Plus.

Un développement chaotique

Jean-Louis Gassée s'installa à Cupertino pour aider Jobs à la division Macintosh qu'il dirigeait, avant de le remplacer. Il était partisan d'un Macintosh plus ouvert, plus extensible. C'est l'idée derrière le Mac Plus. Cette vision d'ouverture, même si cette direction fut rejetée, modifiée et relancée, servira à un autre projet : le Macintosh II.

Steve Sakoman joue un rôle central dans les Mac post-Jobs. Il fut embauché par Jobs en 84 pour aider Apple à créer un portable. C'était une des compétences de Sakoman chez HP. La démission de Jobs à la fin de l'été 85 bouleverse les plans des équipes.



Les caractéristiques

- Processeur : 68000 8 MHz
- Mémoire vive : 1 Mo, extensible à 4 Mo.
- ROM : 128 Ko
- Stockage : lecteur 3 1/2 800 Ko
- Affichage : 9'', 512x342
- Interfaces : SCSI, port lecteur de disquette, port DB9, ports imprimante / modem

Sakoman aide à concevoir le Macintosh Plus qui était en développement depuis plusieurs mois.

Les équipes développe une nouvelle ROM pour prendre en charge les nouveaux hardwares. L'arière a été redessiné pour rajouter les nouveaux comecteurs. La carte mère subit une profonde refonte. À noter que le Mac Plus est le dernier Macintosh à utiliser les connecteurs téléphone et le DB9 pour le clavier et la souris.

Succès ou échec ?

Indéniablement un succès. Il reste aujourd'hui un des Mac les plus appréciés. Il réussit à redonner un nouvel élan à la gamme et à Apple. Malgré un processeur 68000 vieillissant, le Mac Plus a su ouvrir le Macintosh avec les barrettes SIMM et le SCSI. Il est le dernier de la lignée directe du Mac 128k, mais aussi le premier d'une nouvelle gamme plus ouverte qui sera introduite avec les Macintosh SE et II. Le SCSI devient le standard du Mac.

LE SAVIEZ-VOUS ?

1 Du Happy Mac au logo MacOS

Nous vous avions parlé du Sad Mac, du Happy Mac. Le Mac 128 Ko impose donc le Happy Mac au démarrage, avec le "Welcome to Macintosh" et le « logo Picasso ». En mars 95, Apple change le "Welcome to Macintosh" par "Welcome to Mac OS". Ce changement s'explique par le lancement des clones, autorisés par Apple. C'est à ce moment-là que le logo Mac OS apparaît : deux visages souriants. Le Happy Mac sera retiré avec Mac OS X 10.2.

2 Une histoire de son

Le son du démarrage du Macintosh fait partie l'identité du Macintosh depuis son origine. Ce « gong » a été inspiré par les bips de démarrage de l'Apple II. Andy Hertzfeld et Charlie Kellner travaillent ensemble pour créer cette séquence. Dès 81, Andy raconte dans son ouvrage référence qu'il avait écrit une routine sonore pour générer un bip, augmentant progressivement. À cette date, les capacités sonores de la machine sont limitées. À l'été 82, avec l'ajout d'un véritable circuit audio, il était possible de créer une piste 8 bits. C'est là qu'intervient Charles Kellner, développeur dans l'équipe Apple II. Il développe, en dehors d'Apple, un synthétiseur exploitant les capacités de l'Apple II. Les deux développeurs vont travailler sur une séquence audio jusqu'à obtenir un carillon simple et audible. Après des ajustements, le son fut intégré au Mac. Charlie travaille chez lui le code avec un prototype complet, mais il s'aperçoit que le boîtier étouffe le son. Solution : des trous dans le boîtier. Il soumet l'idée aux designeurs. Quelques jours plus tard, Jobs écoute les deux propositions, avec et sans trou. Verdict : l'ajout du trou n'est pas assez significatif. Déçu, Charles retourne dans l'équipe Apple II, mais le son de démarrage reste. Il est intégré à la ROM dans la sous-routine boot beep. Ce son changera avec le Macintosh II. En 1987, Jim Reekes va reprendre et améliorer le son de démarrage et inventer d'autres sons d'ambiance dont le fameux Sosumi, le son d'alerte.

3 Sad Mac : l'alter ego du happy Mac

Le Sad Mac fait partie du quotidien des premières années du Mac. Il signifiait une erreur lors du démarrage de la machine. Il se compose : du Mac triste et d'un code hexadécimal. Durant le développement du Mac, ces codes d'erreur servaient à comprendre un problème matériel ou bas niveau système. Il apparaît début 83.

La programmation facile avec… HyperCard

Hypercard tient une place à part. Son auteur n'est qu'autre que Bill Atkinson un des plus talentueux développeurs du Lisa et du Macintosh. Son travail fut crucial pour l'interface graphique, la gestion de la souris ou encore les routines graphiques, QuickDraw. Il rejoint Apple en 1978. Il développe MacPaint qui devient très rapidement le logiciel référence du Macintosh. Durant une expérience sous LSD, Atkinson imagine la métaphore de la programmation, une autre manière de voir le logiciel et de les concevoir. Ce trip donne naissance à HyperCard. Voilà pour la légende. La réalité est un peu différente…

De Jobs à Sculley

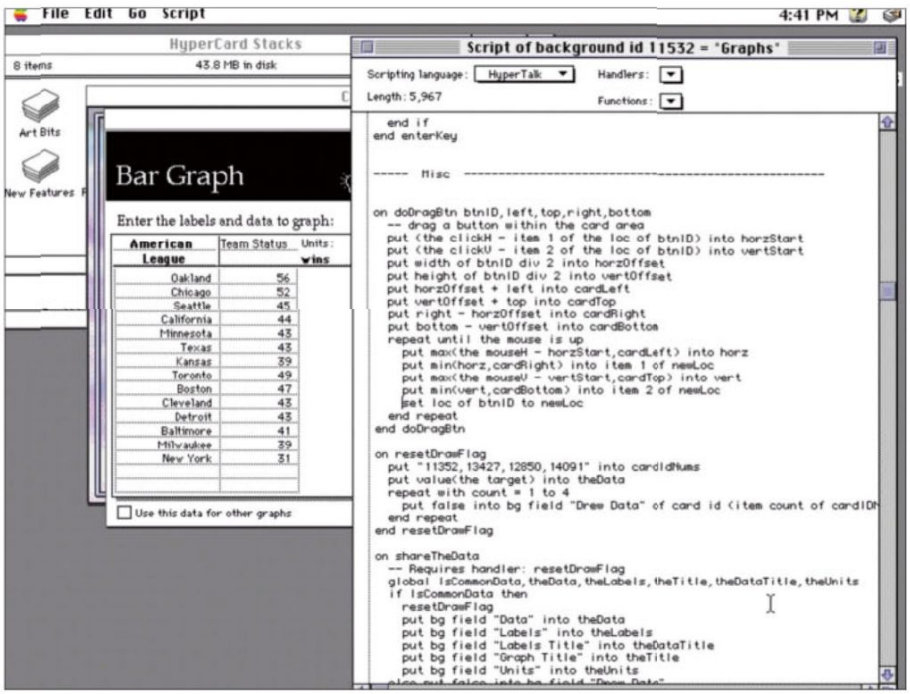
Atkinson fait une dépression après la sortie du Macintosh. La tension était telle durant les mois suivant une partie de l'équipe a besoin de souffler et de faire autre chose. Jobs avait évoqué, dès la reprise en main du projet Macintosh, que l'ordinateur pourrait tenir dans un livre dans les 5 ans.

Cette vision inspira Atkinson et il décida de la développer à partir de la seconde moitié de 1984. Il s'inspira du concept du Dynabook d'Alan Kay. L'idée du Dynabook est de créer un ordinateur portable grand public et particulièrement pour les enfants et les écoles. C'est le projet Magic Slate. Le projet est ambitieux : écran à cristaux liquides, stylet, écran tactile, notion de pages que l'on tourne pour les contenus et les applications, notion d'hyperlien pour naviguer entre les pages. Le format est celui d'une page papier. Le système est entièrement graphique. En 84-85, ces technologies sont embryonnaires et coûtent très cher. Pour animer le Magic Slate, Atkinson repose sur l'électronique du Mac 512 Ko.

À cette époque, Atkinson, comme plusieurs autres ingénieurs de la Pomme, est un Apple Fellow, le titre honorifique le plus important du constructeur. L'Apple Fellow a une liberté de réflexion et de projets, avec un budget dédié.

Cependant, la situation chez Apple a changé radicalement après l'automne 84. Jobs est mis à l'écart du Macintosh et quitte la société le 16 septembre 1985, mais cela fait plusieurs mois que son rôle exécutif n'est réduit à quasiment rien. John Sculley est le véritable patron. En 85, la direction d'Apple veut arrêter de livrer en standard MacPaint avec chaque Macintosh. Cette décision affecte Atkinson. Et la direction ne soutient pas le projet Magic Slate et refuse de produire un tel produit. Déçu, Atkinson garde l'idée des pages pour imaginer un logiciel de programmation révolutionnaire : WildCard. Cependant, il hésite à le soumettre à Apple suite aux déconvenues et il est tenté par quitter la société.

Le développement du projet démarre réellement dans les premiers mois de 1985. Après plusieurs



Page d'accueil de l'outil



mois de travail, il présente le WildCard à Sculley qui est séduit par l'idée. Atkinson pose des conditions pour rester et terminer le travail : que le logiciel soit livré avec tous les Macintosh. Sculley accepte. Une équipe est montée pour finaliser l'environnement, créer la documentation. Le produit est renommé HyperCard. Le langage, HyperTalk, est implémenté par Dan Winkler. Jobs, alors en pleine création de NeXT, veut attirer Atkinson, mais le développeur reste chez Apple pour terminer son projet. HyperCard est officiellement annoncé le 11 août 1987.

La base du logiciel est de créer un univers de programmation le plus accessible possible avec des notions de métaphores (les piles et les cartes) avec un langage de programmation en langage naturel, HyperTalk. Le succès fut immédiat.

HyperCard est une étape importante dans la programmation, car pour la première fois, tout le monde peut créer des logiciels rapidement. La programmation par événement est démocratisée. L'environnement peut être vu comme le précurseur de Visual Basic et de sa programmation dite Lego avec les composants.

Une gestion incertaine et de longues attentes

La version 2.0 fut dirigée par Kevin Calhoun. Elle sort en 1990. La v2 inclut un compilateur, des performances largement en hausse, un nouveau mode de debug. Une version Apple IIgs fut même développée. Au même moment, plusieurs développeurs d'Apple partent et créent General Magic dont Atkinson. La gestion de la couleur apparaît avec la version 2.3. Mais HyperCard est finalement un logiciel à part et Apple ne sait pas comment le gérer. Il fut confié à la nouvelle filiale logicielle : Claris et il n'est plus livré en standard avec les Mac. Il subit la concurrence de nouveaux outils comme Authorware. Le côté hypermédia, l'intégration et la manipulation de multiples médias et contenus attirent naturellement l'outil dans l'équipe QuickTime. La notion d'hyperlien est centrale dans HyperCard. C'est grâce à ce lien que l'on navigue d'une page à une autre, comme un hyperlien web. Hypercard est le premier logiciel à généraliser l'hyperlien. Ce concept inspira les travaux de Bernes-Lee et Caillau au Cern sur le WWW.

Un important travail de reboot fut entamé avec HyperCard 3.0. La version est officiellement présentée, en version alpha / bêta, à la WWDC de 1996. L'environnement supportait totalement QuickTime, QuickDraw 3D et pouvait générer des contenus compatibles pour les navigateurs internet. Les rumeurs se multiplient après l'été 96 pour une sortie en 1997 puis en 1998 ou 1999. La dernière version sort en avril 1998 : HyperCard 2.4.

1993-1997 : psychodrame chez Apple, le retour de Steve Jobs

La machine Apple se dérègle. Les ventes progressent fortement à la fin des années 80. Mais l'arrivée de Windows 3.x et la domination du Wintel touchent durement Apple. Et le début des années 90 est une accumulation de problèmes et d'erreurs : de gammes pléthoriques parfois sans logique de marché, des gammes illisibles à la qualité parfois médiocres, un OS qui n'arrive pas à se renouveler et un processeur à la traîne.

Sculley est viré à l'été 1993, Spindler le remplace. Ce dernier a un mérite : le processeur PowerPC. Pour le reste, il n'arrive pas à enrayer la chute des ventes ni à mener à bien le chantier du nouveau système. Les pertes s'accumulent et Spindler est renvoyé dès le printemps 1996. Gil Amelio, vétéran des composants électroniques, a une mission : sauver Apple et trouver des solutions.

NexStep comme solution miracle ?

Amelio prend acte de l'échec de Copland. En juillet ou août 96, il faut chercher un OS à l'extérieur. Plusieurs candidats sont possibles. L'un d'eux est BeOS de Be Inc., que Jean-Louis Gassée avait créé. Deux offres furent proposées par Apple, toutes deux refusées. Si le système était très prometteur, il n'était pas terminé et surtout il manquait cruellement d'utilisateurs et d'applications.

Jobs fut à la manœuvre mais, contrairement à Be Inc., il sut le faire intelligemment et avec patience : il impose peu à peu l'idée d'un rachat pur et simple de NeXT.

Le 20 décembre 1996, l'annonce fut faite : Apple rachetait NeXT pour 429 millions de dollars en liquidités et actions. Jobs pouvait respirer : il allait rembourser les dettes de NeXT et les investisseurs et surtout, il plaçait son équipe au cœur d'Apple. Jobs ne voulait pas que BeOS soit racheté ; il avait toujours des ressentiments envers J.-L. Gassée par suite des événements dans l'équipe Macintosh fin 84 / début 85.

Jobs devint conseiller spécial auprès d'Amelio. Si le retour partiel de Jobs était acté, ce n'est pas pour autant qu'il prit le pouvoir immédiatement. Il le fit après une longue réflexion et mit en action une stratégie de trois ans pour redynamiser une Pomme malade, réorganiser les gammes, couper dans les projets et réorienter les équipes. Il va minutieusement placer aux postes clés les fidèles de NeXT.

Le plus urgent était de réorganiser Apple, de donner une cohérence aux gammes. Amelio avait déjà procédé à des centaines de licenciements, mais ce n'était qu'une première étape. Jobs, avec ses équipes, va décider produit par produit ce qu'il garde, ce qu'il arrête.



Le 7 janvier 1997 : Gil Amelio présente la nouvelle stratégie système et invite sur scène Jobs et Woz. Il comprend très vite que Jobs lui vole la vedette avant de prendre sa place.

La fameuse matrice de produit élaborée en 1997-98. Elle permet de rencontre les efforts et de sortir uniquement des produits capables de se vendre rapidement et de répondre aux besoins du marché.

	Consumer	Professional
Desktop		
Portable		

En 1996, plus de seize gammes étaient commercialisées par Apple. Après plusieurs semaines de réflexion avec les équipes, il établit une matrice de produits sur dix-huit mois pour relancer Apple : quatre produits identifiables, les meilleurs possibles. Cela permet de ne pas multiplier les projets, de recentrer les efforts et de redonner aux utilisateurs une vision claire d'Apple.

Bref : faire moins mais beaucoup mieux.

Le traitement est brutal : 70 % des produits sont arrêtés, plus de 4 000 employés sont licenciés, arrêt

des licences Mac OS, nouveau système en développement basé sur NeXT. Au retour du cofondateur, Apple était à trois mois de la faillite avec une perte de 1 milliard de dollars.

1997 : Microsoft, le sauveur ?

Depuis de nombreuses années, Microsoft était accusé de position dominante sur le marché ou d'ententes avec les constructeurs. Les premières enquêtes remontent au système MS-DOS qui dominait le marché PC. La pression sur l'éditeur était particulièrement importante à partir de 1995-96 après la sortie de Windows 95 et d'Internet Explorer. La domination écrasante du Wintel était l'autre problème. Apple était le seul concurrent capable de rivaliser, mais le constructeur était en chute libre.

L'annonce du partenariat Apple – Microsoft est faite le 6 août 1997 durant la conférence Macworld Expo de Boston. Par cet accord, les deux sociétés mettent un terme aux conflits de licences, confirment le développement de la suite Office sur Macintosh et font d'Internet Explorer le navigateur par défaut (ce qui fut mal accueilli par la salle), avec une collaboration autour de Java. Microsoft investit également 150 millions de dollars dans Apple.

Les relations entre les deux compagnies ne furent jamais simples. Mais il était temps de passer à autre chose. Et pour Microsoft, le procès antitrust se profilait. Surtout, durant les 500 jours de Gil Amelio, les relations avec Microsoft s'étaient fortement dégradées : l'arrêt du développement d'Office sur Macintosh était un mauvais signal pour le marché. Mais pour Microsoft, comment développer des logiciels sur Macintosh si le futur OS n'était pas défini ? Amelio et Jobs, peu de temps après le rachat de NeXT, rencontrent Bill Gates, mais la situation restait confuse. Gates ne savait pas trop qui dirigeait réellement Apple ni sur quel OS développer…

Une fois Amelio hors-jeu, Jobs relance les contacts avec Gates et Microsoft, comme le rappelle Walter Isaacson. Il veut mettre fin aux procès et avoir la garantie que Microsoft développera ses logiciels pour Macintosh. En quatre semaines, l'accord est réglé. Les ultimes détails furent fixés à peine quelques heures avant la keynote de Jobs à la Macworld Expo. Les ultimes détails furent finalisés durant les répétitions de la keynote après un appel d'une heure de Gates.

La mise en scène fut malheureuse : Bill Gates sur l'écran géant de la scène — il n'était pas sur place — et Jobs apparaissant tout petit à côté…

Malgré tout, Jobs avait obtenu ce qu'il voulait : le soutien de Microsoft, calmer les marchés et pouvoir travailler sereinement sur la suite : l'iMac et le futur Mac OS X.

Chaque CEO veut son Macintosh !

Chaque CEO cherchait son « Macintosh », le produit phare qui marquerait son règne. Cette obsession a parfois aveuglé la direction et les projets se sont empêtrés dans des problèmes sans fin.

1 / MessagePad, tardivement récupéré par Sculley

Dès 1991, Sculley était persuadé que l'informatique ultra-mobile et le tactile allaient révolutionner les usages. Il fallait donc développer une nouvelle *génération de terminaux autonomes. Aux origines* du projet, il y a Steve Sakoman, chez Apple depuis 84. Mais après trois ans, il veut quitter la Pomme pour créer un ordinateur ultra-mobile et communicant, capable de gérer la reconnaissance d'écriture.

J.-L. Gassée souhaite garder l'ingénieur et l'autorise à lancer le projet, avec l'aval de la direction. Sakoman accepte le défi et rassemble autour de lui une petite équipe, dont Kapor (inventeur de Lotus 1-2-3) et Kaplan, un développeur historique d'Apple. Un peu plus tard, c'est un des développeurs de génie du Lisa et du Macintosh qui rejoint le projet, Steve Capps. Sakoman exige une totale autonomie, dans son propre laboratoire, à l'écart. À l'été 87, la direction donne son feu vert. Newton n'était pas l'unique projet cherchant à créer une nouvelle manière d'utiliser l'ordinateur. Un autre projet est mené par Atkinson et Hertzfeld, deux développeurs du premier Mac. Quand Apple mise sur le Newton, ce projet sort d'Apple pour devenir la société General Magic.

En 1990, Newton est un fiasco : un terminal totalement inutilisable pesant plusieurs kilos ! La direction veut y mettre fin et Sakoman quitte la société. Le projet n'avait aucune direction claire.

Quand Sculley s'intéresse enfin sérieusement au Newton, personne ne sait ce qu'il est possible d'en faire. Larry Tesler, autre développeur de génie du Lisa et du Macintosh, est chargé de packager un produit commercialisable. Après plusieurs semaines à récupérer ce qui peut l'être, un matériel commence à émerger, mais la reconnaissance d'écriture est l'un des points faibles du système.

Pour sortir de l'impasse technique, il fallait trouver une solution externe. Le choix se porta sur la technologie Graffiti de la société russe Paragraph. Durant deux ans, tout le projet fut remanié en profondeur : design, matériel, processeur, etc. Pour le processeur, Apple utilise un des premiers processeurs ARM, mais il fallait redévelopper partiellement le système ! L'intégration des technologies Paragraph et des briques logicielles d'Apple exigea de nombreux mois de travail. Mais ce travail n'a pas pu être stabilisé à la sortie du MessagePad. Tant bien que mal, Sculley sort le MessagePad à l'été 1993 après plusieurs reports. Mais pour fabriquer le MessagePad, Apple savait qu'il lui fallait un partenaire maîtrisant totalement la miniaturisation et l'intégration de ce type d'électronique. Le choix se porta sur Sharp.

Le projet fut tellement intense qu'il usa des dizaines d'ingénieurs. Sculley voulait SON Macintosh. Les équipes travaillèrent jour et nuit, ou presque, durant un an pour finaliser le matériel et le système. Au printemps 93, plus de 3 700 bugs avaient été remontés. Impossible de lancer la production avant le milieu de l'été. Le projet laissa des traces en interne et les démissions se succédèrent *après la sortie du MessagePad.*

Mais à l'été 93, Sculley quitte précipitamment Apple en juin après une nouvelle crise et de très mauvaises ventes. Les équipes Newton continuent le travail dans la tempête. Début août, la première version exploitable du NewtonOS fut intégrée au MessagePad 100.

Sur la partie design de l'appareil, les premiers travaux réels débutèrent en 87 avec le projet Figaro. De nombreux mockups et prototypes furent produits pour tester tel ou tel concept. Les premières idées ne sont pas un PDA tel que nous le connaissons avec le MessagePad 100, mais une tablette tactile. On notera que le boîtier du MessagePad évoluera à chaque version.

2 / Le Macintosh des 20 ans : l'ultime Macintosh pour Amelio

Cette machine doit célébrer les vingt ans d'Apple. L'ambition était de rompre avec tout ce que faisait Apple dans les années 90 : le design, le boîtier, le type d'écran, etc. Pour Amelio, ce projet est, semble-t-il, la machine qui doit marquer son « règne ». Si la machine est superbe, avec un design novateur et un système audio haut de gamme, le prix de vente était tellement extravagant (9 000 $) que l'échec commercial était inévitable avant même sa sortie. Pour Amelio, c'est plus une question de prestige que de ventes. Il découvre le projet quelques semaines après sa nomination comme patron d'Apple (début 1996). Il comprend que deux clans s'affrontaient depuis des mois : les partisans et les opposants à Spartacus. Le projet, très ambitieux, se heurte à de multiples problèmes techniques : fabrication de la caisse, intégration hardware, énorme bloc externe.

Les premiers prototypes fonctionnels remontent à 1995. Mais en réalité, il faut remonter bien plus loin : automne 1992 avec le projet Pomona. Ce projet sera l'une des fondations du futur Spartacus. Pomona avait l'ambition de créer un ordinateur dédié et conçu pour la maison. Le projet n'aboutit pas. Mais il était audacieux sur le design et le hardware : LCD par défaut, design plus proche d'une chaîne B&O que d'un ordinateur.

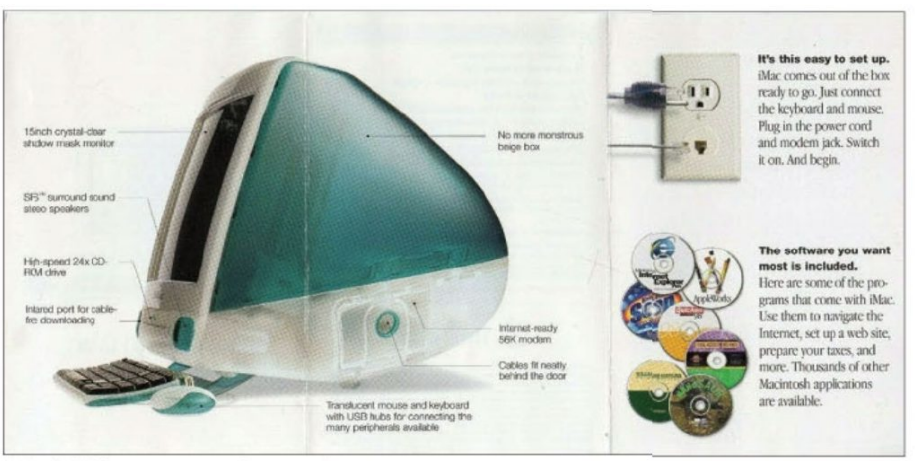
Pomona ne réussit pas à dépasser le stade des prototypes. Jonathan Ive fut recruté lorsque l'équipe



Bill Gates à la keynote de la MacWorld Boston d'août 1997.



Un ces concepts du projet Pomona. Ces designs aboutissent au Macintosh des 20 ans



L'iMac de 1998, une autre vision du micro-ordinateur

design planchait sur cette nouvelle approche du Macintosh. Le design du Pomona aurait dû être la base du futur Spartacus. Au printemps 93, l'un des concepts est de reprendre les courbes des systèmes audio B&O. Ive travaille sur cette approche qui sera l'une des caractéristiques du Spartacus. En octobre 94, le design est approuvé et le projet peut réellement être développé avec une équipe hardware dédiée. Il faut plus de deux ans de travail pour fixer définitivement tous les éléments du Spartacus. Une des difficultés est de concevoir le support permettant de tenir l'écran contenant l'électronique.

La particularité du Spartacus est sa verticalité. Idée géniale, mais qui compliqua la conception matérielle. Il fallait repenser la carte mère, la connectique, créer une alimentation externe, trouver un lecteur de CD-ROM capable de fonctionner verticalement, etc. Il faut tout concevoir : clavier (dérivé du PowerBook), système audio avec sa télécommande, redessiner tous les boutons. Pour l'alimentation et l'audio, les équipes conçoivent un impressionnant cylindre. On peut aussi noter l'extrême finesse du lecteur de disquette intégré à droite de l'écran. La carte mère est dérivée de l'architecture Gazelle et reprend le design du PowerMac. Le manque de place à l'arrière de l'écran explique le peu d'extensibilité de la machine. L'architecture Gazelle date de février 1997 et utilise un PowerPC 603e.

Échec complet : 12 000 exemplaires produits, beaucoup trop cher (7 490 $).

3 / L'iMac sauve Apple

Quand Jobs définit les quatre produits prioritaires, il manque alors un véritable Macintosh passe-partout, capable d'être utilisé à la maison, à l'école, voire dans les petites entreprises. Facile à configurer, facile à installer, accès réseau simplifié, etc.

Jobs trouve dans l'équipe de design des projets et des esquisses intéressantes. Il met l'équipe au défi. Rapidement, la synergie Jobs – Ive va se mettre en place.

De ces réflexions et défis, le projet iMac va peu à peu émerger. L'idée est de proposer une machine qui, visuellement, peut s'intégrer partout et offrir un beau design. On rompt avec les câbles multiples et le lecteur de disquettes (dont la fin est actée par l'iMac). Et tout doit être fait pour simplifier la connectique. Ce dogme de départ va s'imposer au marché : lecteur de CD-ROM par défaut, connecteur USB, prise Ethernet pour l'accès à Internet. Fini les multiples connecteurs et le lecteur de disquette ! La rondeur apporte de la chaleur, et non une informatique standard et froide. Le translucide du plastique est là pour donner une ambiance. C'est le même translucide que celui de l'eMate. Le design peut rappeler des appareils des années 1960.

L'iMac est dévoilé en mai 1998. Il est souvent dit que des projets novateurs étaient en développement, notamment un embryon d'iMac. Même si Amelio comprenait qu'il manquait des machines puissantes et nouvelles, il restait dans la continuité des gammes existantes. Il fallait aller vite, très vite, pour espérer sauver Apple. L'usage des logiciels 3D fut primordial pour réduire le temps de développement.

En réalité, l'idée de créer un Macintosh moins cher et grand public était au cœur des réflexions de Mitch Stein (directeur des interfaces homme-machine) dès 1994-95. Le projet My First Apple est lancé. Puis le projet prend le nom de Macintosh JFW et des designers, dont Jonathan Ive, y sont affectés. Le projet évolue, mais rien n'était abouti quand Gil Amelio fut nommé patron d'Apple. Amelio voulait un Macintosh peu cher pour dynamiser les ventes, mais quand il quitte la direction, rien n'est prêt.

En 1997, la première idée était de construire un Network Computer. Le NC était défendu par Ellison d'Oracle, ami de Jobs. Mais cette vision était trop éloignée du Macintosh pour certains membres de la direction. Jobs réorienta son idée en misant sur un ordinateur totalement compatible avec Internet. Ce sera l'iMac. L'enjeu était de faire mieux que les constructeurs PC : plus simple, plus cool, réduire les câbles et la configuration. Le projet C1, alias MacMan, alias iMac, était né. Pour gagner du temps, la décision fut prise de s'appuyer sur le hardware G3 déjà utilisé par Apple.

Le choix du plastique pour l'enveloppe est rapidement acté, mais il a fallu contrôler le processus de fabrication et proposer plusieurs couleurs à la sortie de la machine. Il fallait créer les moules et vérifier que le plastique se comportait bien. En plus de la machine, les équipes ont développé la fameuse souris plate et ronde ainsi que le clavier.

8 échecs (1980-2017)

Durant l'Histoire d'Apple, les flops ont été nombreux, certains très importants, d'autres moins. Voici une petite sélection.

1980 / Apple III et les limites organisationnelles du constructeur

C'est le premier échec commercial d'Apple. Il a été mal aimé dès sa sortie. Les multiples problèmes et instabilités ont plombé la machine. Plusieurs correctifs sont apportés, mais rien ne sauvera la machine.

Le projet Sara a montré les limites du constructeur. Le manque d'organisation des équipes a eu des conséquences désastreuses. L'Apple III fut le premier modèle à intégrer un lecteur de disquette et à proposer un disque dur externe. Apple introduisait un rupture de gamme alors qu'en interne, deux autres projets étaient en développement : le Lisa et le Macintosh.

Le projet Sara est initié en 1978. Pour la direction et Jobs, il fallait regarder au-delà de l'Apple II et proposer une évolution radicale. Woz n'y participe pas. Le design est assuré par Jerry Manock qui avait fait la caisse de l'Apple II. Le boîtier est plus haut et anguleux. L'obsession de Jobs pour le silence a été une difficulté supplémentaire : ne pas avoir de ventilateur pour ne pas perturber l'utilisateur. Le design nécessite environ 10 mois de travail. Côté hardware, le III est très différent de l'Apple II : plus rapide, plus de mémoire, lecteur de disquette par défaut, écran 80 colonnes, clavier intégré.

Mais les designers travaillent un peu à l'aveugle, car même si les dimensions de la carte mère furent fixées très tôt, ils ne savent pas exactement ce qu'il y aurait à l'intérieur, et de nombreuses maquettes furent réalisées selon les évolutions des spécifications. Et il a fallu prendre en compte les réglementations sur l'émission d'ondes qui devaient évoluer aux Etats-Unis, cela explique la poids de la châssis. Les équipes design et hardware manquent parfois de coordination.

Quelques mois plus tard, un réalité émerge dans les équipes : Sara va dans le mur. Jobs, très présent au début, l'est de moins en moins car il travaille sur le projet Lisa. La cohérence se perd. Lorsque la production de l'Apple III est lancé, les ingénieurs savent que tout n'est sec sur le hardware et le logiciel est toujours en développement.

Le retard logiciel est la conséquence de l'absence de machines complètes. Elles arrivent à peine 3 mois avant la production. Or, il faut terminer le système SOS beaucoup plus complexe que l'OS de l'Apple II. Il doit gérer le disque dur, le futur ProFile. Et il faut aussi terminer les manuels. Et l'Apple III ne propose pas de killer app. C'est la conséquence de garder secret les projets, Jobs voulant garder le contrôle. Cette erreur stratégique est corrigée avec le Macintosh. Mais pour l'Apple III, c'est trop tard.



Apple III avec son écran. © Benoît Prieur

Et ce n'est que le début du chaos. Les pré-séries cumulent les erreurs système et les dysfonctionnements. Le hardware n'est pas stable. Après enquête, les ingénieurs découvrent des défauts de câbles et dans plusieurs connecteurs. Jobs découvre ce fiasco lors d'une visite de l'usine de production. Sa colère explose. Il faut arrêter la production et fixer chaque défaut. Mike Scott lance une réorganisation interne : les équipes s'organisent par lignes de produits. Ces ajustements obligent à repousser la date de sortie. Pis, quelques semaines après la commercialisation, le constructeur retire la machine pour fixer d'autres dysfonctionnement. L'horloge temps réel réapparut officiellement, mais en option. Cependant, malgré toutes les corrections, une nouvelle carte mère et des améliorations des flux thermiques, l'Apple III resta un échec.

Personne chez Apple n'imaginait, comme l'écrit si bien Paul Kunkel dans le livre référence AppleDesign (1997), qu'un échec était possible.

1992 : OpenDoc : une approche composable trop complexe

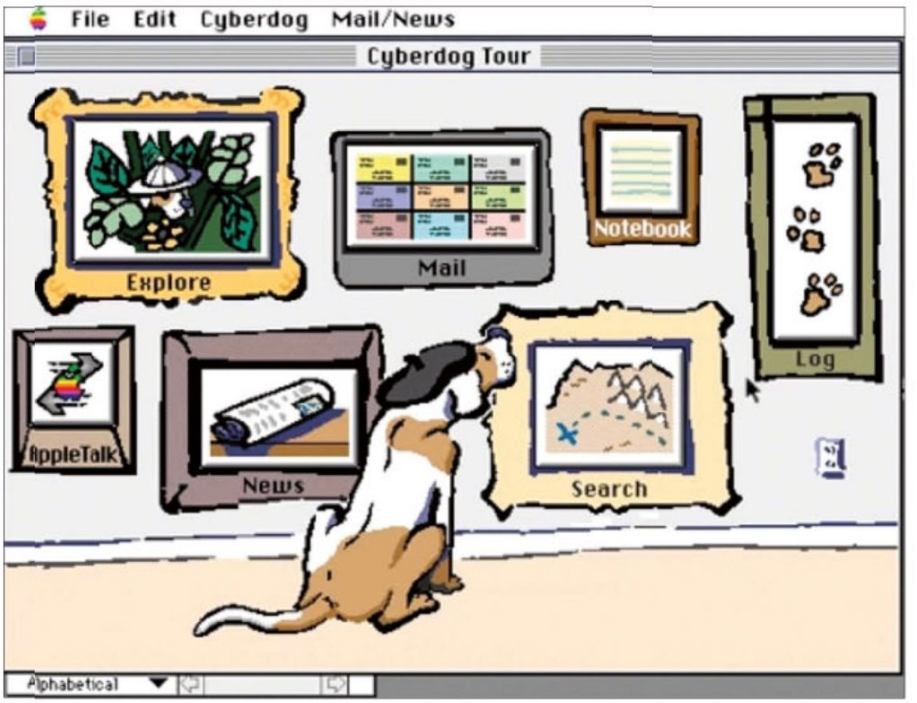
OpenDoc constitue un des chantiers majeurs d'Apple dans les années 1990. OpenDoc permet de créer des applications sous forme de composants que l'on ajoute dans un conteneur. L'application pouvait être « composée » par l'utilisateur. Cette technologie très prometteuse s'est révélée d'une grande complexité. Peu d'éditeurs s'y intéressent. La technologie devait être par défaut dans Copland et disponible sur Windows, OS/2 et les systems Unix. Elle se veut une alternative plus puissance que OLE de Microsoft.

OpenDoc apparait en 1992. Le développement est réalisé en collaboration entre Apple, IBM et WordPerfect.

Le Système 7.5 reçoit la 1ere version. OpenDoc est alors orienté document. Les documents intègrent différents composants permettant d'utiliser de types de contenus hétérègones.



CD-Rom d'OpenDoc. Photo : Jaxermd



Interface de CyberDog

Une des applications phares était le navigateur CyberDog.

CyberDog illustre la modularité offerte par OpenDoc. Le navigateur est entièrement modulaire avec des briques de bases pour le réseau, la communication. L'internet rompt avec tous ce que l'on avait l'habitude de voir via des icônes réalistes. On pouvait accéder aux pages HTML, à la messagerie, à un moteur de recherche. Bref, CyberDog devrait être le centre nerveux Internet de son Macintosh.

L'équipe de développement supporte les plugins Netscape et Java était envisagé. CyberDog n'était pas là pour affronter les navigateurs du marché mais plutôt comme un démonstrateur technologique pour convaincre les utilisateurs et surtout les développeurs.

La version 1.0 est disponible en 1996. La version 2 date de juin 1997 mais le projet fut arrêté en mars au retour de Jobs.

Apple n'avait ni le temps ni les moyens de continuer à développer OpenDoc et d'autres modèles applicatifs apparaissaient plus simples.

1994 : eWorld

En janvier 1994, Apple annonce eWorld, une sorte de monde virtuel avec des métaphores où les bâtiments permettent d'accéder aux différents services. Apple espère voir une communauté se créer et prospérer. Les services les plus utilisés sont le centre eMail et le centre communautaire. Mais la direction voit que eWorld n'a pas une croissance suffsante et coûte cher à maintenir. En face, AOL est le service le plus utilisé.

Un des gros défauts du service : les tarifs beaucoup trop élevés. Il y a une double facturation : l'abonnement mensuel et un prix horaire. Les sites Internet vont vite dépasser eWorld. Apple autorise un accès depuis un navigateur et même le MessagePad, mais rien ne sauve le service. Ce monde se vide de ses utilisateurs. Et le 31 mars 1996 à 12h01, eWorld est coupé.

1996 : Copland (1996)

La direction d'Apple sait que l'OS prend trop de retard sur les concurrents. La sortie de Windows 95 illustre tous les manques du Système repose toujours sur les versions de 84/85... Les ambitions étaient immenses mais le projet, mal maîtrisé, deviant incontrôlable.

Devant l'impossibilité de version complete et stable, la decision tombe en août 1996 : Copland est arrêté pour chercher un OS extérieur. Copland mobilisait les ressources vitales d'Apple: 500 développeurs et 300 millions de $ investis... Au même moment, le constructeur traverse une grave crise : baisse des ventes, pertes.

Copland doit au Système sous l'appellation Mac OS 8. Dès 1993, des équipes internes commencent à développer un nouveau système pour Macintosh et les processeurs PowerPC.
Les ambitions autour du futur OS sont :

- 100 % natif PowerPC
- utilisation massive de la technologie OpenDoc, co-développée par Apple et IBM
- réécriture et développement de toutes les briques système
- nouvelle interface graphique
- nouvelles couches de communication
- compatibilité élargie avec DOS et Windows

Copland repose sur une plateforme technique (Next-Generation Platform) en rupture avec l'existant. Elle contient toutes les fondations bas niveau : noyau et services PowerPC, composants système, les couches de communication, l'interface et le support multimédia. Dessus se greffent les supports Mac, OEM (pour les clones) et la compatibilité Windows. Puis la troisième pile concerne le réseau (client universel, Internet, etc.).

L'un des intérêts de Copland était la nouvelle fondation technique :

- système natif PowerPC / RISC
- multitâche préemptif
- réécriture de la gestion des I/O pour booster les performances
- protection mémoire pour améliorer la stabilité système
- réduction de l'empreinte système en mémoire
- nouveau mécanisme d'extension système
- abstraction matérielle
- micro-noyau
- support de TCP/IP, IPX
- nouveaux moteurs graphiques : QuickDraw GX, QuickDraw 3D
- support d'Unicode
- nouveau modèle de développement

Face à l'OS historique, la concurrence offrait des systèmes modernes :

- BeOS est en développement depuis 1990-91 : d'abord lié à la BeBox puis comme OS indépendant. Il faudra attendre 1995 pour voir une première version opérationnelle, mais le système tarde à se stabiliser et à sortir sur PowerPC.
- Windows 95 sort en été 1995. En développement depuis 3 ans, cet OS va être une claque pour Apple.
- NextStep : l'OS créé par NeXT (fondé par Steve Jobs) sort en 1990, après plusieurs années de retard, et propose déjà des fondations particulièrement robustes, une interface radicalement différente.
- Linux : le kernel Linux existe depuis 1991 et évolue rapidement et les premières distributions apparaissent dès 1992/93.

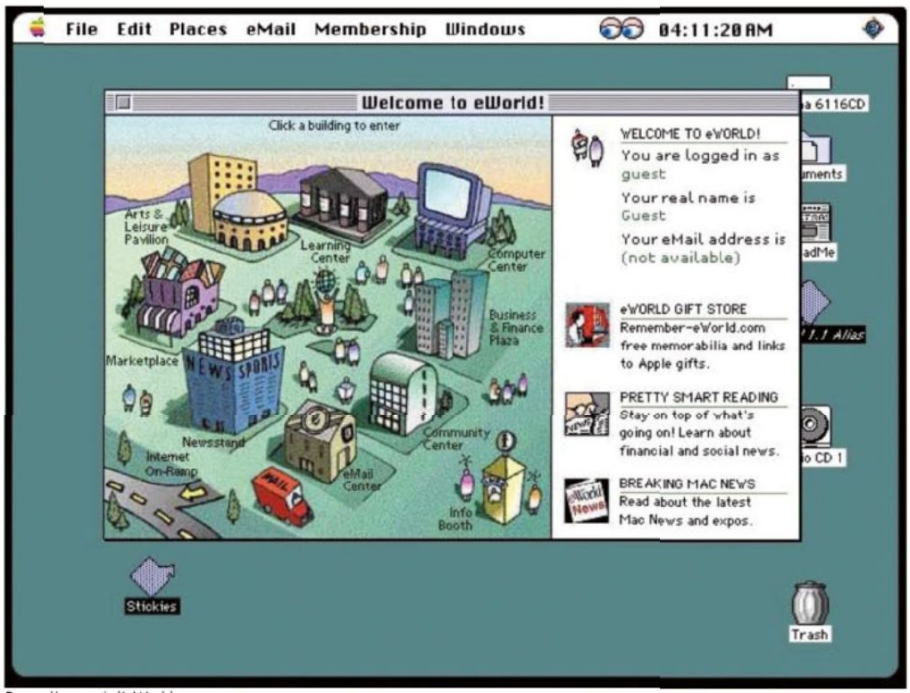
L'un des aspects les plus marquants de Copland est son interface entièrement nouvelle : nouvelle ergonomie avec effet « 3D », personnalisation et surtout l'utilisation de thèmes. Le thème permettait de changer l'aspect des éléments graphiques, l'un des plus connus est le thème Gizmo. L'idée était de généraliser l'aide active aux utilisateurs avec le concept « montre-moi ». Bref, moderniser l'ensemble visuel sans toucher aux grands principes Mac.

Copland autorisait le fonctionnement des applications existantes, mais les premières versions du système utilisaient un émulateur et sans profiter des nouvelles fonctions du système. Les applications 100 % natives à la nouvelle architecture étaient attendues pour la version suivante connue sous le nom de code Gershwin.
Durant la conférence développeur du printemps 1995, Apple a commencé à en parler officiellement. Les premières versions, disons des versions plus ou moins Delta / Alpha commençaient à circuler. Une version finalisée était attendue pour début 1996. Les mois passèrent sans que Copland se stabilise. Et les versions successives dédiées aux développeurs étaient instables et difficile à manipuler.

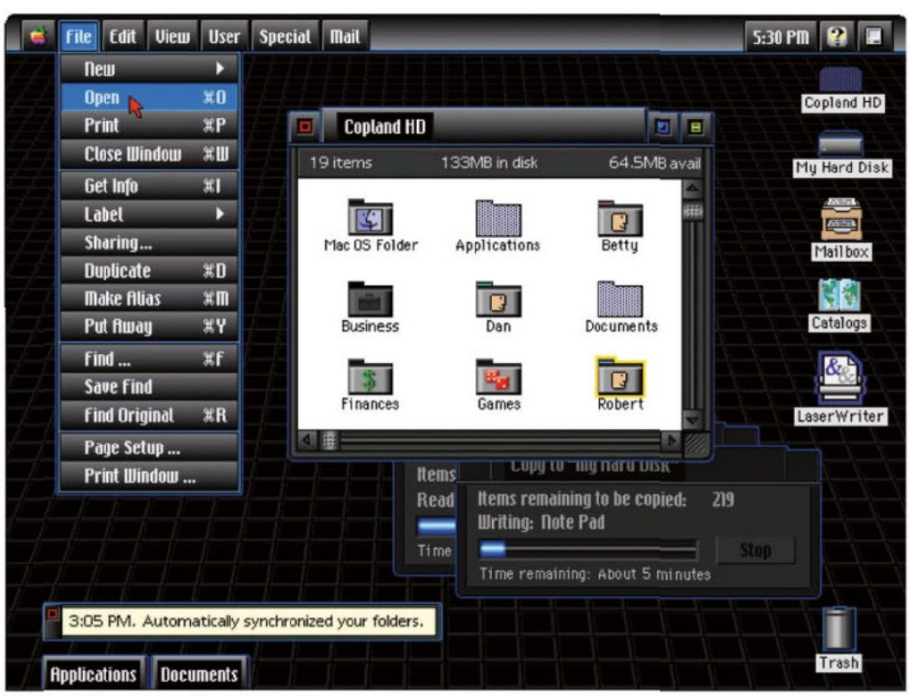
Pour pouvoir utiliser Copland, il fallait avoir un Power Mac compatible, et la liste était (très) réduite. Il fallait formater le disque dur, de préférence avec l'outil disque dur livré avec Copland puis y installer une version complète de Mac OS 7.5.x puis Copland depuis Mac OS 7.5.3.

La dernière build connue est la D11E4 datant de juin 1996. Le projet fut stoppé quelques semaines plus tard.

Trop ambitieux ? Mauvaise gestion des équipes ? Manque d'une vision globale du projet ? Une réécriture sous-estimée ? Copland était un projet radical avec une réécriture complète de nombreuses briques techniques par x équipes. Chaque équipe, chargée d'un ou plusieurs composants système, travaillait dans son coin, rendant l'intégration difficile. L'état de la D11E4 (juin 1996) était loin d'une véritable version bêta.



Page d'accueil d'eWorld



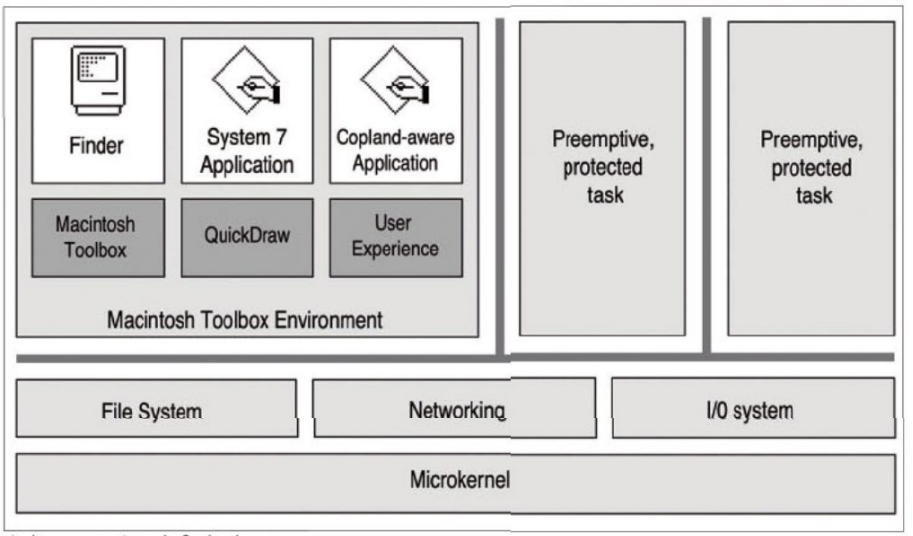
Un thème d'interface possible de Copland

Toutes les technologies de Copland ne sont pas jetées. La refonte de Mac OS, connue sous le nom de Mac OS 8, utilisera des technologies Copland, celles qui étaient les plus abouties. Cette base sera préservée avec Mac OS 9.

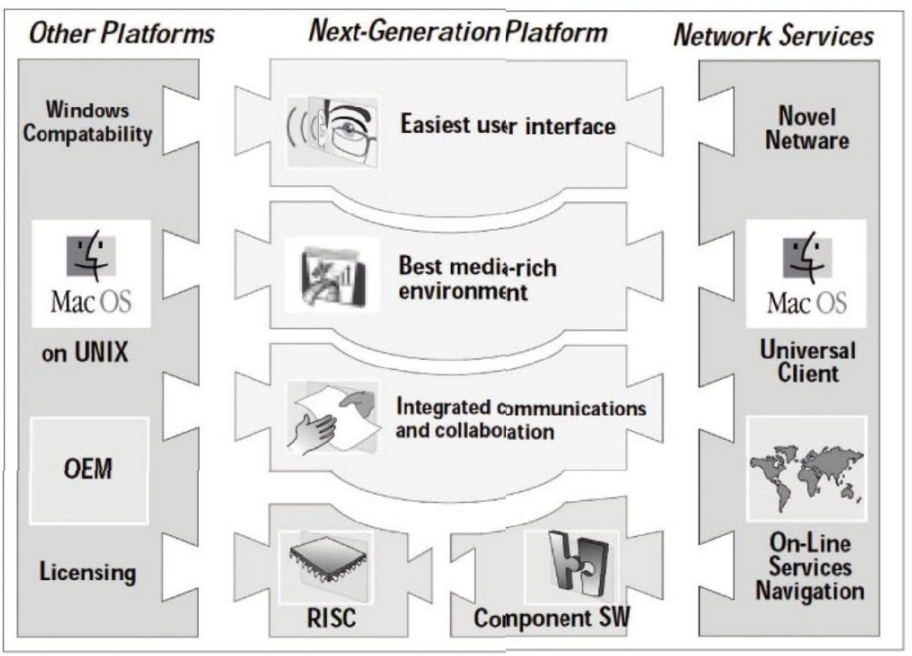
En parallèle, Apple commence à chercher un OS à l'extérieur.

1996 : Pippin, la malédiction du jeu !

Au début des années 1990, Apple tente d'attirer les éditeurs de jeux. Pour dynamiser les ventes du Mac et attirer de nouveaux utilisateurs, le jeu semble être la solution la plus évidente. Des responsables d'Apple se rapprochent de Bandai pour créer une console basée sur le PowerMacintosh : le projet Pippin. Apple fournit l'architecture de référence, le design hardware. A Bandai de faire le reste. A l'origine, le processeur devait être un 68030. Le changement de processeur fut nécessaire pour supporter les fonctionnalités attendues par les utilisateurs.



Architecture système de Copland



Au-delà de Copland, c'est une nouvelle architecture logicielle et hardware qui se préparait



Pippin, modèle le plus connu, la version Atmark. ©Evan-Amos

Bandai avait l'ambition de créer une console basée sur le Macintosh. Le constructeur exigea plusieurs limitations pour ne pas concurrencer le Macintosh ni de créer un véritable clone. Un des avantages techniques fut la disponibilité d'un modem et la possibilité d'installer un accès Internet. Apple ne voulait pas faire cet effort, mais céda devant la pression de Bandai.

La base technique est un PowerMacintosh utilisant un processeur 603. Le choix est purement économique : le 604 est très puissant, mais très cher. Mais les conflits internes gênent la conception de la plateforme. Pippin est une véritable plateforme Macintosh. Mais il ne faut pas que la console puisse nuire aux ventes des vrais Macintosh…

Au printemps 1996, Gil Amelio annonce la Pippin 1997 Reference Platform. Cette spécification aurait pu devenir la Pippin 2.0. Le processeur reste le PPC 603. Mais le matériel subit une nette évolution : 8 Mo de RAM par défaut, lecteur DVD, support audio AC-3, accélération 2D et 3D, port FireWire, ouverture vers les réseaux (notion Network Computer).

L'OS a été modifié pour s'adapter à la plateforme. Le port ADB sera lui aussi modifié et spécifique à la console. Comme il s'agit d'un véritable PowerMacintosh, on peut booter sur un système standard. La console possède un port d'extension pour installer des cartes. Il est aussi possible d'utiliser le bus SCSI interne. La carte mère dérive de la seconde génération des cartes mères PowerMacintosh (architecture PCI).

Le design de la console fut terminé en à peine 6 mois. En décembre 1994, un accord avait été signé avec Bandai, alors que le projet était en développement depuis quelques mois. C'est un designer de Lunar Design qui s'occupa de définir les grandes lignes. Les manettes évoquent un boomerang. Un trackball était envisagé, mais l'idée ne fut pas retenue. L'arrivée d'Internet bouleversa les spécifications. Cette connectivité se fait grâce au Geoport. Elle fait de Pippin une des premières consoles à pouvoir accéder à Internet.

Les premiers prototypes montrés publiquement reprennent le design des premiers PowerMac 6100. Les premières démonstrations publiques eurent lieu à Cannes durant le Milia de janvier 1995 puis au CeBIT de mars. Durant l'automne, les premières unités fonctionnelles sont expédiées aux partenaires. Pour les développeurs, les outils de développement, la Pippin SDK, arrivèrent à partir de décembre 1995 (DR 0). La version 1.0 fut distribuée en février 1996. Au début de l'année, les ROM de la console furent stabilisées. A noter que des prototypes sont équipés d'un disque dur et/ou d'un lecteur de disquettes.

Apple introduisit une protection logicielle pour éviter les développements non contrôlés. Les CD-ROM contenaient un fichier caché authentifiant le CD. Ce fichier était généré par Apple pour l'éditeur.

Pippin utilise donc un CD-ROM dit pippinized. C'est à dire, que le CD-ROM utilise un OS Pippin dédié et des fichiers permettant de démarrer et de lire le CD. Le Pippin OS est dérivé du System 7.5.2 provenant directement du PowerMacintosh. La ROM 1.3 supporte le lecteur de disquette, les disques SCSI et le slot PCI de la carte mère. La console boote uniquement avec un CD-ROM. Le slot d'extension est très utile, car il permet d'étendre (théoriquement) la Pippin. On peut installer une carte pour lire des disquettes !

Bandai livrait avec la Pippin un bundle de 6 CD-ROM dans l'Internet Kit : navigateur Netscape, kit de connexion, Pippin Networks. L'absence de stockage interne est un handicap, car il fallait une carte mémoire supplémentaire.

Cinglant échec : la Pippin subit la concurrence des consoles japonaises. Et Bandai ne fera pas la pro-

motion prévue. Le manque de jeux va rapidement pénaliser la plateforme ainsi que les piètres performances en affichage 3D. Bandai arrête la vente des consoles Pippin en 1997. Au retour de Jobs, la Pippin fait partie des projets non prioritaires : on ferme.

2010 : l'antennagate de l'iPhone 4

Depuis quelques mois, l'iPhone est englué dans une polémique liée à des plaintes de réception. C'est alors que Jobs réalise l'une des démonstrations de mauvaise foi les plus mémorables : « vous n'avez qu'à bien le tenir ». Bref, ce ne serait pas un problème technique, mais l'utilisateur qui tient mal son iPhone… Pour répondre à l'antennagate, Apple organise un événement spécial. Le constructeur n'a pas réellement présenté d'excuses, ni procédé à un échange massif. En revanche, il publie une page sur les problèmes de réception de la concurrence et dévoile une petite partie des laboratoires dédiés à l'iPhone et aux antennes. Problème réel, mais finalement sans impact à long terme sur l'iPhone.

En 2014, la gamme iPhone 6 traverse une autre polémique : le bendgate. En cause : des iPhone 6 qui se pliaient trop facilement. La justice américaine juge en 2018 qu'Apple connaissait le problème. Des documents internes semblaient montrer que cette génération se pliait bien plus facilement que la génération précédente. Un programme de réparations, facturé 150 $, fut lancé.

2014 : Apple Car, projet Titan

Dix ans de R&D, jusqu'à 5 000 personnes mobilisées et recrutées et plus de 10 milliards de budget : le projet Titan fut annulé en interne fin février 2024. D'autres chantiers plus urgents attendaient et la direction actait le retard considérable de développement et les difficultés technologiques à surmonter.

Apple n'ayant jamais communiqué sur ce projet, nous ne savons pas réellement où en était le développement de la voiture et des systèmes embarqués. La voiture semble avoir intéressé Jobs dès 2008. Un futur marché où Apple se devait d'être présent. Elon Musk évoquait en 2016 la possibilité qu'Apple travaille sur une voiture électrique à conduite autonome. La même année, un vétéran du hardware, Bob Mansfield, reprit du service pour prendre la direction du projet.

L'un des axes technologiques était la conduite autonome (juin 2017, Tim Cook), mais pas forcément pour une voiture Apple. De nombreuses rumeurs évoquent des changements de design et des ambitions trop grandes dès la première génération : pas de volant, système totalement autonome, suppression des pédales. Quelques dizaines de tests réels ont été effectués sur route. En juillet, un employé, Xiaolang Zhang, est arrêté par le FBI avec des données confidentielles sur le développement de Titan. En août 2018, Doug Field est débauché de Tesla pour diriger le projet. En 2019, la startup Drive.ai est achetée pour renforcer l'expertise en conduite autonome.

Au début des années 2020, Apple cherchait des usines et des partenaires pour la fabrication et les batteries. Le responsable du projet change une nouvelle fois. Les rumeurs de négociations au Japon et en Corée avec des constructeurs se multiplient en 2021. Puis le silence.

Si le projet de construire une voiture est abandonné, une équipe continue à travailler sur les systèmes automobiles et le projet pourrait se relancer si la bonne idée est trouvée. Mais construire une voiture est très complexe. Sans doute que la volonté de tout contrôler n'a pas facilité le travail des équipes de Titan avec des partenaires potentiels.



Steve Jobs démine le problème de réception durant une conférence spéciale



AirPower

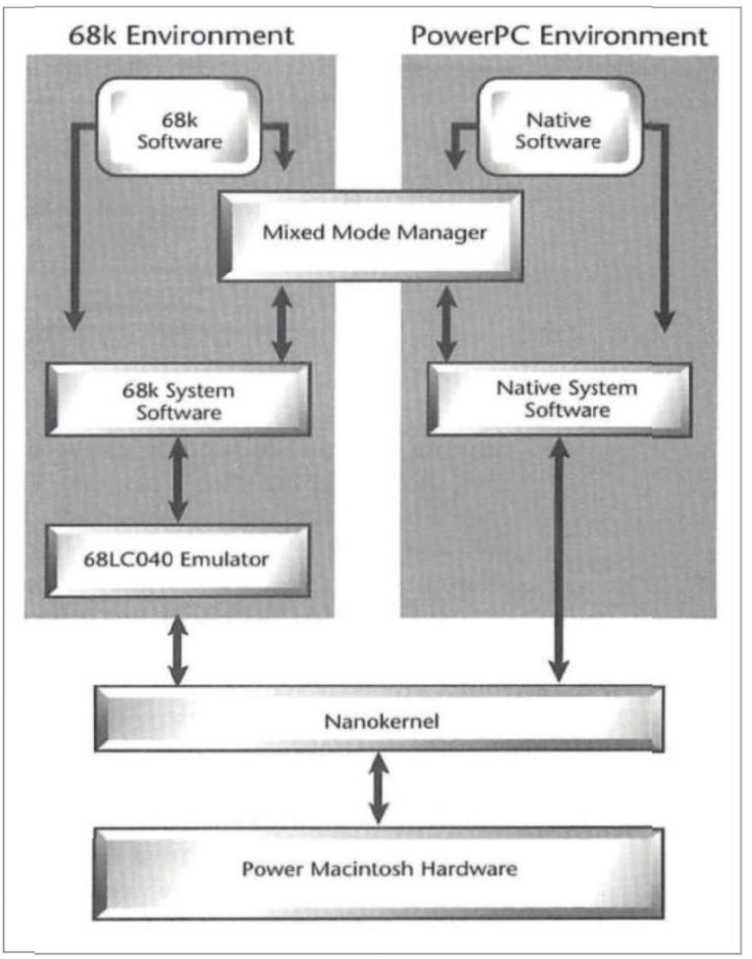
2017 : AirPower

Dévoilé le 12 septembre 2017, l'AirPower était la station de recharge sans fil pour iPhone, AirPods et Apple Watch. La sortie était promise pour 2018. Le produit fut retardé, puis finalement annulé en mars 2019.

Lors de l'annonce, des prototypes étaient visibles. Quelques semaines après, les mentions d'AirPower commencent à disparaître. Les problèmes techniques se cumulent et les ingénieurs ne trouvent pas de solutions efficaces : gestion de la chaleur, chargement simultané de plusieurs appareils, interférences entre les bobines, prise en charge de deux normes de charge.

Les transitions matérielles : Apple ou l'art de la maîtrise

Le micro-processeur a toujours été un sujet stratégique pour Apple. Lorsque le Lisa est en développement, les ingénieurs se posent des questions : créer un micro-processeur Apple ou utiliser une architecture du marché ? Après l'échec d'une architecture processeur maison, les équipes hardware optent pour le Motorola 68000. Mais dix ans plus tard, Motorola a du mal à faire évoluer son architecture et Apple ne peut rivaliser avec la concurrence. Intel et AMD proposent bien mieux.
La première transition sera la fondation du PowerPC avec IBM et Motorola. La seconde étape sera l'architecture x86 d'Intel et l'étape actuelle, Apple Silicon, basée sur les designs ARM. Revenons sur les étapes clés d'une histoire parfois peu connue.



Le fonctionnement des environnements 68k et PowerPC sur PowerMacintosh

Du Motorola 6800x0 au PowerPC (1991-1994)

Devant la stagnation du processeur 6800x0, Apple décide d'agir. Acheter Sun Microsystems ? Utiliser l'architecture POWER d'IBM ? Spindler, le futur patron du constructeur, négocie avec IBM dès la fin de 1989. L'option POWER devient crédible mais IBM hésite à s'engager. Motorola rentre dans le projet pour la fabrication. C'est ainsi que l'alliance AIM (Apple IBM Motorola) est dévoilée au printemps 1991. L'objectif : concevoir le PowerPC. Mais il faut un OS capable de tourner sur ce hardware. Deux pistes : porter l'OS actuel du Macintosh ou miser sur le projet Pink, un développement interne pour créer un OS plus moderne.
Après de multiples revirements, la décision est de porter l'OS actuel sur PowerPC, même si au lancement des premiers PowerMacintosh, ce sera plutôt un bricolage technique le temps de porter nativement l'OS. Il faudra trois ans de travail pour finaliser l'architecture hardware, préparer la transition et assurer le fonctionnement des applications actuelles. Les premiers échantillons des PowerPC arrivent en 1992. La documentation technique commence à parler de la transition vers le PowerPC dès l'été 1992.

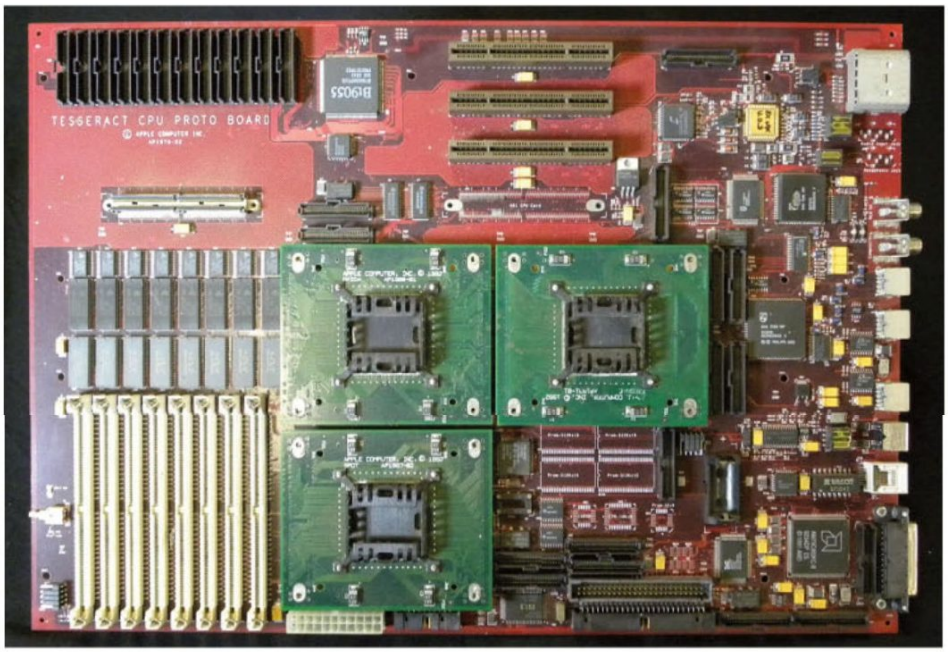
Les premiers PowerMacintosh sont commercialisés en 994. System 7.1.2 introduit le premier support du PowerPC mais uniquement en 32 bits. Un des enjeux est d'utiliser les applications 6800x0 sur les PowerMacintosh. Pour ce faire, Apple développe une couche de compatibilité pour exécuter les applications non PowerPC en toute transparence. Le System 7.1.2 de 1994 gère les deux types de binaires (68k et PPC). La ROM du PowerMacintosh contient l'émulateur 6800x0. Pour assurer le bon fonctionnement entre les environnements 68k et PPC, Apple introduit le Mixed Mode Manager. Il s'occupe de déterminer les appels système et comment un code 68k peut utiliser du code natif, dans des cas très précis, par exemple la partie graphique en utilisant QuickDraw natif et non QuickDraw émulé, ce qui aurait fortement pénalisé les performances.

1992 : PROJET STAR TREK

Dans la mythologie Apple, Star Trek est un des projets les plus fascinants. Il faut noter que les informations concrètes sur ce développement sont limitées. Le projet était mené par des équipes d'Apple et de Novell. Dans la plus grande discrétion, il fut décidé de tester la portabilité de System 7 sur des PC x86. Les ventes d'Apple baissent, ouvrir le monde x86 pourrait être une solution.
Les délais sont très courts pour créer un prototype fonctionnel de System 7. Le projet fut torpillé par le départ de Sculley et l'arrivée de Spindler. Ce dernier pouvait depuis quelques années privilégier une solution avec IBM, le PowerPC. Il arrêta le développement et les équipes s'occupèrent de migrer l'OS sur PowerPC.

Mac OS X : comment utiliser l'ancien Mac OS ? (1997-2007)

Lorsqu'Apple rachète NeXT, le développement d'un nouvel OS démarre immédiatement : Rhapsody. C'est le premier OS moderne et natif PowerPC d'Apple, et surtout capable de rattraper les concurrents. Il reprend les solides fondations techniques de NeXTSTEP / OpenStep.



Prototype PowerPC et sa carte mère : le Tesseract (1992). Chez Apple, les cartes mères des prototypes sont le plus souvent rouges. Photo : bitsaver



Carte mère du Mac Mini DTK 2020. Annonce eBay

Pour faire fonctionner les applications Mac OS 9, les équipes créent le Blue Box qui prendra le nom de (Mac OS) Classic. Classic agit comme un émulateur de Mac OS 9 sur Mac OS X. Quand on lance une application Mac OS 9, Classic se lance et exécute l'application. Les performances sont au rendez-vous et permettent une transition en douceur en attendant que les développeurs migrent vers Mac OS X. Cet environnement sera retiré en 2007.

PowerPC vers Intel (2000-2006)

Au retour de Jobs (1997), la question du processeur redevient un enjeu stratégique. Jobs connaît bien Intel. Depuis la fin des années 1960, Andy Grove était devenu un des mentors de Steve et celui-ci admirait profondément le co-fondateur d'Intel.

Le projet Star Trek, portage de l'OS sur x86, avait montré sa faisabilité, mais le rachat de NeXT changea radicalement la donne. Les développeurs de NeXT connaissaient et maîtrisaient les processeurs Intel.

La transition vers le x86 fut annoncée en juin 2005. Le PowerPC évoluait assez lentement et Apple rencontrait aussi des difficultés techniques pour intégrer les dernières générations PPC dans les portables. En 2003, Apple espérait sortir un portable G5 le plus rapidement possible, mais ce projet resta lettre morte.

Régulièrement, des rumeurs évoquaient une migration sur x86, plusieurs années avant l'annonce officielle. Un Macintosh x86 circulait en interne et chez les développeurs : le DTK de 2005. Il s'agit d'une machine dédiée aux développeurs à des fins de test. Il utilise un Pentium 4, CPU jamais utilisé en production par Apple. Ce modèle est très limité : pas de FireWire, absence de DVI.

Les premiers Macintosh Intel sortent en 2006 : iMac et MacBook. L'iMac Intel est dévoilé en janvier 2006. Puis, les mois suivants, Apple sort le Mac mini et les premiers portables x86.

Dès son retour, Jobs arrêta les clones, mais pouvait accepter ceux de Motorola si le fondeur adaptait les PowerPC aux portables. Las d'attendre, Jobs exclut Motorola de l'équation d'Apple. Mais visiblement, les débats au conseil d'administration furent longs et animés : faut-il le faire, quand le faire, comment le faire. Dans le plus grand secret, des ingénieurs d'Intel et d'Apple travaillèrent ensemble pour concevoir l'architecture et redévelopper toutes les couches du système, et particulièrement la compatibilité avec les applications PowerPC. Il était indispensable que Mac OS X version Intel puisse exécuter les applications PPC sans heurts et en toute transparence. Techniquement, il fallait concevoir une couche d'émulation complète.

En réalité, il semble bien que les équipes de Mac OS X n'eussent jamais perdu de vue la plateforme Intel et régulièrement le système était compilé et adopté. Dès fin 1999, courant 2000, Mac OS X fut porté sur x86 pour tester la faisabilité. Fin 2001, la version semble suffisamment avancée pour la présenter aux responsables système. Jobs fera des démos de l'OS à Sony. En 2002, des ingénieurs supplémentaires sont mobilisés. Ce sera le projet Marklar. Cela signifie que la direction, et donc Jobs, préparait le terrain soit pour migrer vers x86 soit pour licencier le système sur PC.

Le développement matériel fut sans doute lancé plus tardivement. Lorsque la décision fut prise de basculer sur x86, il fallut accélérer le développement de l'OS, intégrer l'émulation PPC et créer les futurs Mac Intel.

Comme pour les anciennes transitions, Apple intègre une émulation du code PowerPC : Rosetta. Il permet la traduction à la volée du code. La première version apparaît en octobre 2005 avec Mac OS X 10.4.3 qui supporte les Mac x86.

Du x86 à Apple Silicon (2020)

La dernière transition en date est Apple Silicon. En juin 2020, durant la WWDC, l'annonce est faite : Apple va abandonner l'architecture x86 pour les processeurs Apple Silicon. Pour les développeurs, un kit de transition est proposé. Il s'agit d'un Mac mini équipé d'un processeur ARM, un Apple A12Z. Pour assurer l'exécution des applications x86, Apple annonce Rosetta 2. La transition est lancée dès 2018 avec le projet Kalamata.

Rosetta 2 émule l'architecture x86 et ne nécessite pas de modifier le code x86. Il s'occupe de traduire les instructions en instructions ARM. Il n'est pas installé par défaut. L'OS détecte si l'application a besoin de Rosetta. Apple introduit aussi un modèle de développement fonctionnant sur Apple Silicon et x86 : les binaires universels. Dans ce cas, pas besoin de Rosetta.

2007 : l'iPhone révolutionne le téléphone mobile



Un des multiples prototypes du projet P1

Rumeurs, contre-rumeurs, nous étions nombreux à nous attendre à quelque chose d'important durant la keynote de Steve Jobs en janvier 2007. C'est sans doute l'une des plénières les plus mémorables du patron d'Apple.

Ce premier iPhone n'est que partiellement l'iPhone qui sera le point de départ de la révolution mobile. Il pose les fondations qu'Apple va peu à peu consolider avec l'arrivée des applications mobiles (les apps) et en ouvrant la plateforme aux développeurs. Jobs était tout d'abord réticent, mais il s'est laissé convaincre de la nécessité d'ouvrir le système.

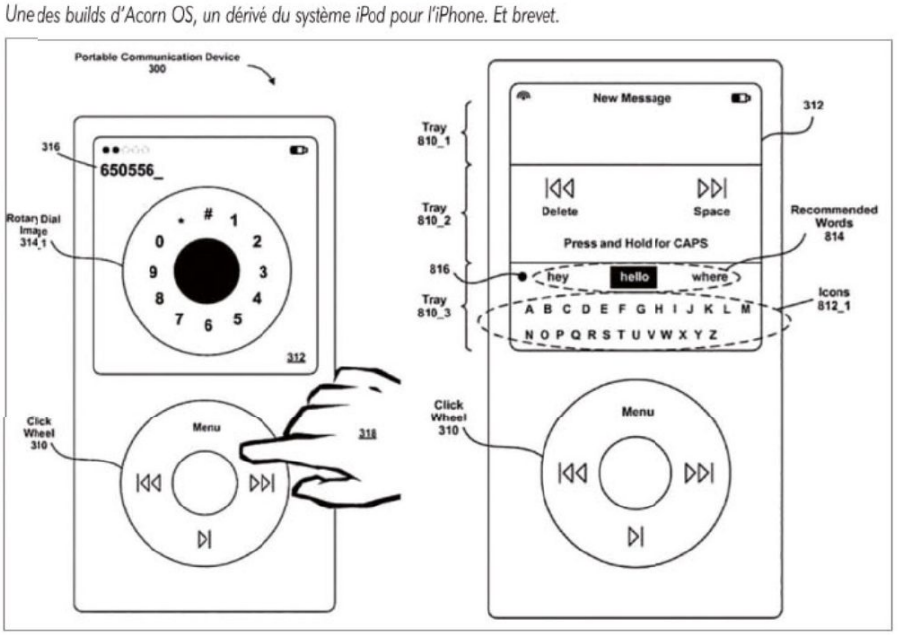
Aux origines du projet, Apple ne cherchait visiblement pas à créer un téléphone, mais plutôt une tablette, puis, en 2004, le projet change de direction avec le Project Purple. Une des clés de la réussite du projet fut le Français Jean-Marie Hullot. Il dirigeait une équipe à Paris, dans le plus grand secret, pour concevoir les services de téléphonie et les contours de l'iPhone. Il suggère l'idée du téléphone à Jobs dès 2001, même si Steve n'était pas forcément emballé. Le Français travaillait déjà chez NeXT. Hullot quitte Apple en 2005.



Une des builds d'Acorn OS, un dérivé du système iPod pour l'iPhone. Et brevet.

L'idée était révolutionnaire : pas de stylet, pas de clavier (ce qui, à cette époque, était inconcevable sur un téléphone), un écran entièrement tactile et un système embarqué. Les défis techniques étaient grands : comment optimiser l'électronique, quel processeur choisir, comment créer l'écran et le gérer. La question du système ne sera pas simple : un système inspiré de l'iPod, un autre système ou partir sur une variante de l'OS X ? Après de nombreux mois de recherches et d'hésitations, ce sera le système OS X qui sera choisi : le futur iOS.

Le jour de la présentation, le prototype de l'iPhone est loin d'être stable. Seules les actions que Jobs doit montrer sur scène sont jugées utilisables et strictement balisées (on ne sort pas du scénario). Tout se passe bien et la magie iPhone va pouvoir prendre. Il faudra plusieurs mois de développement supplémentaires pour intégrer le système et l'ensemble des services de communication.



Boutique du Magazine PROGRAMMEZ!

Les anciens numéros de programmez!

Le magazine des dév - CTO - Tech Lead



266



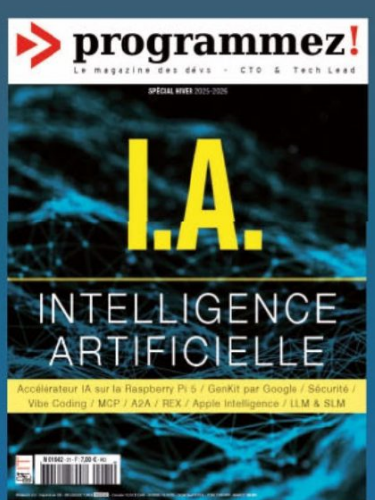
267



270



272



HS21



273

Tarif unitaire 8 € (6,99€ + 1,01€) (frais postaux inclus)

☐ 266	: ___ ex	☐ 270	: ___ ex	☐ HS21	: ___ ex
☐ 267	: ___ ex	☐ 272	: ___ ex	☐ 273	: ___ ex

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

soit ___ exemplaires x 8 € = _____ € ..... soit au **TOTAL** = _____ €

☐ M. ☐ Mme Entreprise : _____ Fonction : _____

Prénom : _____ Nom : _____

Adresse : _____

Code postal : _____ Ville : _____

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

PROG HS22

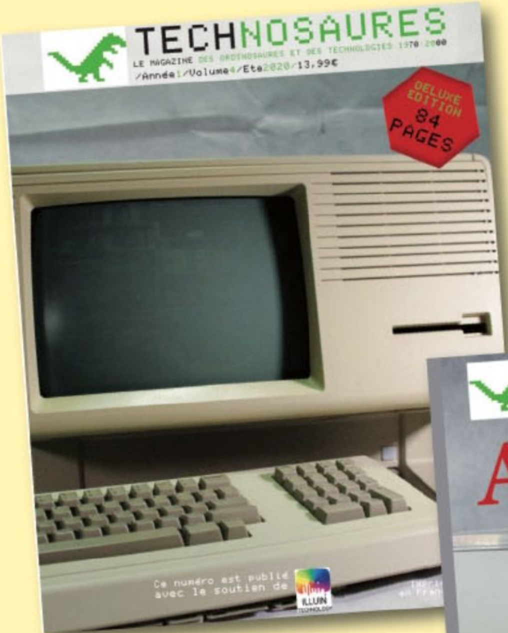
Offres pouvant s'arrêter à tout moment, sans préavis

Boutique du Magazine TECHNOSAURES

La trilogie Apple :

3 numéros exceptionnels

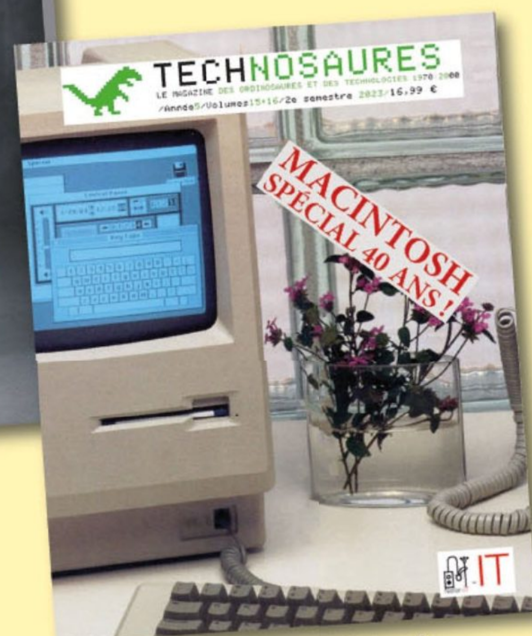
TECHNOSAURES
LE MAGAZINE DU VINTAGE COMPUTING



Apple Lisa*



Applemaniac



Les 40 ans du Macintosh

49€
(Tarif france)

** version standard ou deluxe (84 pages) selon les stocks disponibles*

Offre Spéciale : trilogie Apple

☐ 49 € x .... |__| ex = |__|__|__|__|__| €

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

☐ M. ☐ Mme Entreprise : |__|__|__|__|__|__|__|__| Fonction : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Prénom : |__|__|__|__|__|__|__|__|__|__|__|__|__| Nom : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Adresse : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Code postal : |__|__|__|__|__| Ville : |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

PROG HS22

Offres pouvant s'arrêter à tout moment, sans préavis

Steve Ballmer, Microsoft, préfère en rire. Blackberry et Nokia n'y croient pas. Pourtant, en 18 mois, ils vont disparaître.

Le multitouch n'est pas une technologie inconnue : les projets P1 et P2

Jobs rejette le clavier immédiatement : peu pratique, encombrant sur le device et surtout des centaines de pages de contraintes. Apple a déjà acquis une solide expérience dans le tactile avec le Newton. Certes avec un stylet, mais les bases sont là. Ensuite, vers 2003, une petite équipe travaille sur le vrai multitouch, plusieurs doigts simultanés pour manipuler l'écran, faire un zoom, etc.

Le prototype 035, un iPad avant l'heure, est conceptualisé entre 2002 et 2004. Ive est à la manœuvre pour le design. Le travail autour de la tablette aboutit au projet iPhone quand Jobs voit un prototype d'interface. Pour la partie logicielle, Scott Forstall sera un des responsables clés.

Deux prototypes sont développés : Purple 1 alias P1, basé, sous la responsabilité de Fadell, sur un iPod nano, et P2, partant du prototype 035 de l'équipe d'Ive. Le P1 apparaît limité dans l'interactivité et les fonctionnalités et l'usage de la molette de sélection n'est ni pratique ni intuitif. Et l'iPod nano limite le hardware. Après 6 mois d'efforts, P1 est purement et simplement arrêté.

P2 concentre les efforts. Ive s'occupe du design industriel, Fadell de l'ingénierie, Forstall doit adapter Mac OS X au P2 ! Ive travaille sans l'OS réel et l'équipe logiciel ne travaille pas sur le design défini… *Mais Ive est tenu au courant des évolutions* logicielles et chaque responsable échange.

L'équipe de Forstall travaillait dans un étage entièrement dédié et interdit à toute personne non autorisée. Rapidement, Ive mise sur le tout écran : limiter au maximum les boutons, l'écran tactile est au cœur du device.

Fin 2004, Ive travaille sur deux concepts de design : Extrudo et Sandwich. Le premier a la taille d'un iPod mini. Le second a une forme rectangulaire, aux coins arrondis. Mais Extrudo pose des problèmes de découpe et les angles trop vifs peuvent blesser l'utilisateur. Malgré tous les prototypes et les efforts pour corriger les défauts, Sandwich est choisi comme design final.

Côté logiciel, tout s'accélère à l'été 2005 quand Forstall travaille activement sur l'architecture de l'OS pour le futur iPhone. Un des défis est de créer un clavier virtuel pour remplacer le clavier physique. Fin 2006, les couches logicielles sont suffisamment avancées pour être intégrées aux *derniers prototypes*.

En février 2006, les designers n'arrivent pas à définir un concept. Ive veut d'autres idées. Il demande à Shin Nishibori, un designer japonais, de proposer d'autres réflexions. Durant plusieurs semaines, il va reprendre les conceptions minimalistes de Sony pour concevoir le téléphone.

En mars, rien n'est résolu. Pour sortir de l'impasse, Ive écarte les designs et décide de repartir sur un des premiers prototypes. Ce design ressemble beaucoup au design du premier iPhone. Et finalement, le design va rapidement pouvoir se fixer : la face avant est dépouillée, sans nom, sans logo, un unique bouton. L'écran domine visuellement.

L'autre défi est l'écran et sa structure. Les écrans utilisant une couche plastique n'étaient pas satisfaisants ; le verre fut alors choisi. Il fallait trouver un verre résistant et capable de laisser passer le tactile. Il a fallu plus de 6 mois pour trouver le fournisseur et définir tous les aspects de l'écran et les différentes couches.

En automne 2006 : rien n'est prêt, l'OS ne fonctionne pas, le téléphone est une catastrophe. Pour Jobs : il n'y a aucun produit ! Problème : en janvier, l'iPhone doit être dévoilé. Les équipes ont 3 mois pour résoudre les problèmes, fixer le hardware, stabiliser l'OS. Il fallait gérer l'écran, les différents capteurs (accéléromètre, capteur de proximité, etc.). Tout était nouveau pour les ingénieurs et les développeurs. Décembre 2006 : un prototype suffisamment « stable » est présenté à AT&T, qui est impressionné.



Le design général se fixe à partir d'avril 2006



Le 1er iPhone (2007)

L'App Store : l'autre réussite de l'iPhone

Steve Jobs ne voulait pas d'App Store et pas d'applications tierces sur le téléphone. Mais il finit par accepter cette ouverture. L'App Store est une fonctionnalité clé de l'iPhone. Pour contourner la limitation, il était possible de développer des apps web sur le moteur WebKit embarqué dans le téléphone. Finalement, Jobs confirme, en octobre 2007, la disponibilité d'un SDK dédié pour écrire des applications natives.

Il sera disponible en février 2008. L'App Store ouvre officiellement le 10 juillet 2008 avec environ 500 apps.

AppleScript : les applications s'ouvrirent au monde extérieur…

AppleScript est un langage de programmation intégré au système qui permet d'écrire des scripts en langage naturel déclenchant depuis l'extérieur, des actions dans les applications. Il est fourni avec un éditeur de scripts qui est partie intégrante de MacOS.



Christine Buttin

Faisant partie du support développeurs d'Apple Europe dès 1989 puis d'Apple France, j'ai eu l'honneur d'introduire cette technologie au cours de multiples conférences développeurs, ma mission étant d'engager ceux-ci à l'implémenter dans leurs applications. Par la suite, en tant que free-lance, j'ai donné de nombreuses formations sur le langage et réalisé des scripts d'automatisation pour des entreprises avant de consacrer mon temps aux formations sur le développement Cocoa et WebObjects et à la réalisation de sites intranet en WebObjects.

Mac OS et les AppleEvents

En 1991, Apple délivre une nouvelle version du système du Macintosh, Système 7. Le multitâche a été amélioré pour permettre à plusieurs applications d'être actives en même temps. Le système comporte une nouvelle API, les AppleEvents, qui met en place l'échange de messages entre applications selon le schéma suivant : **IMAGE 1**

Le message comporte des informations concernant l'action à déclencher dans l'application 2 et des paramètres complémentaires pour désigner les objets concernés et fournir des précisions sur celle-ci. La réponse est optionnelle, elle contient un objet ou une liste d'objets résultant de l'exécution de l'action.

Jusque-là, rien de très nouveau dans le monde du logiciel. Ce qui est nouveau, c'est qu'Apple entreprend la définition d'une suite d'événements standards qui comporte toutes les actions que la plupart des applications réalisent en général : créer, supprimer, ouvrir, fermer, imprimer, quitter, … Et tous les objets qu'elles gèrent : document, fenêtre, application. Durant l'année 1990, Apple réunit les développeurs par famille d'applications pour créer via un vocabulaire commun des suites propres à leur domaine : suite Texte, suite Tableur, suite Graphique , suite Base de données… dans l'idée qu'une application, par exemple un traitement de texte puisse communiquer de la même manière avec le tableur X ou le tableur Y. En outre, les développeurs peuvent ajouter leurs propres suites pour donner accès à des services spécifiques à leur application.

De nombreuses applications vont implémenter les AppleEvents à commencer par Apple avec le Finder, ClarisWorks…, par Microsoft avec Excel et Word, par QuarkXPress, Adobe et aussi par l'implication des supports développeurs qui vont encourager les développeurs à mettre en œuvre cette technologie.

Plusieurs idées se profilent derrière les AppleEvents :

- réaliser des applications centrées sur leur cœur de métier et qui utilisent les services des autres pour éviter à l'utilisateur de se retrouver avec des boîtes à outils complexes qui contiennent de plus en plus de fonctionnalités et sont de plus en plus lourdes, le contraignant à travailler dans un monde fermé. En d'autres termes pour l'utilisateur, passer de la philosophie tout-en-un à une philosophie « je choisis mes outils ».
- fournir via les Suites qui sont publiées, un dictionnaire de vocabulaire standardisant les échanges d'une application à l'autre.
- permettre aux utilisateurs de concevoir des scripts via AppleScript et son éditeur, qui est livré avec le Système, pour piloter les applications.

IMAGE 2

AppleScript **IMAGE 3**

C'est un langage de programmation orienté objet, complet, accompagné d'un éditeur de scripts, pour réaliser des scripts de commandes destinées aux applications, sous la forme de phrases en langage naturel. Une instruction permet d'indiquer à quelle application on s'adresse et les actions et objets qu'on utilise sont ceux définis dans les Suites des AppleEvents. Par exemple :

```
tell application "Finder"
get name of every folder of window 1
end tell
```

Lors de l'exécution du script, AppleScript traduit les actions en AppleEvents, les envoie aux applications concernées et récupère les résultats éventuels.

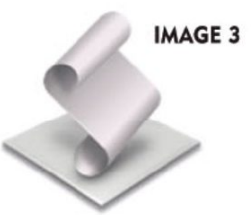


IMAGE 3

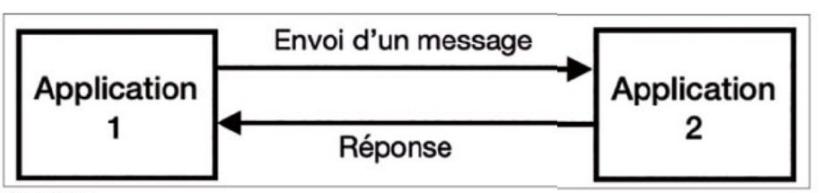


IMAGE 1

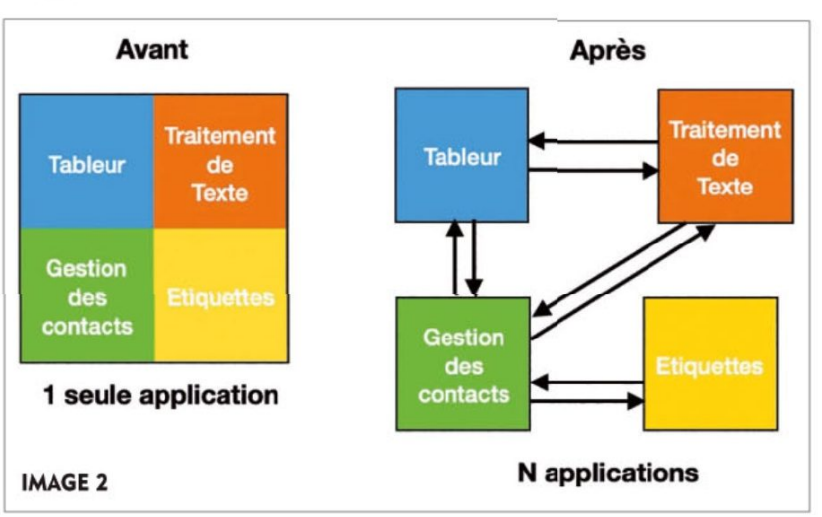


IMAGE 2

La force d'AppleScript repose sur le fait que le vocabulaire qu'on utilise est :

- simple, ce sont les mots du langage naturel,
- défini, le dictionnaire de chaque application est visible avec l'éditeur de scripts,
- extensible. Nouvelle application veut dire nouveau dictionnaire et enrichissement du vocabulaire. D'ailleurs Apple continuera à étendre le vocabulaire en créant une Suite d'événements Système et des suites propres aux applications fournies avec le système.

Lors de la première version d'AppleScript, il est possible *d'écrire des scripts en anglais, français...* Finalement seul l'anglais restera.

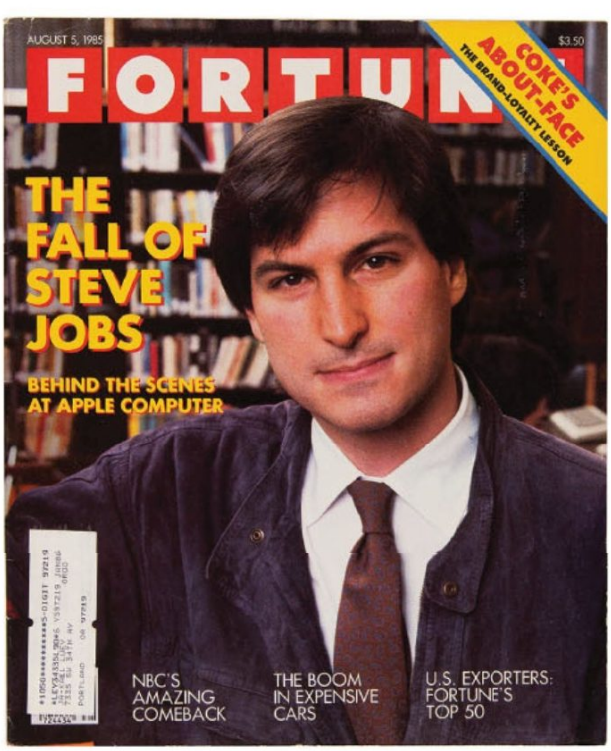
AppleScript permet :

- d'automatiser des tâches entre applications, par exemple l'envoi automatique de mails à des destinataires différents comportant le texte d'un article issu d'un traitement de texte et des photos stockées dans un dossier,
- d'effectuer des tâches difficiles à réaliser manuellement comme compter les caractères d'une page de cet article ou calculer son temps de travail dans un projet à partir des événements de iCal,
- d'ajouter des fonctionnalités complémentaires aux applications en y attachant des scripts qui pilotent d'autres applications,
- d'enregistrer les actions de l'utilisateur pour générer les scripts. Cette dernière fonctionnalité n'est pas généralisée à toutes les applications.

Avec AppleScript on peut commencer à rêver à des possibilités tout à fait nouvelles : on peut mettre en œuvre des processus métier de circulation d'informations. On peut automatiser des tâches répétitives, par exemple changer le nom de tous les documents avant archivage. On pourrait même imaginer, à condition que les applications implémentent les AppleEvents, que Mac OS observant tous les matins, via les AppleEvents émis par les applications, que vous faites systématiquement les mêmes actions vous propose de les faire pour vous automatiquement au démarrage de votre ordinateur. Cette fonctionnalité n'a jamais été réalisée.

Conclusion

AppleScript fait partie des joyaux de la couronne conservés par Steve Jobs lors de son retour chez Apple et qui ont été réimplémentés dès la première version de MacOS X : l'éditeur de scripts était bien là et les applications fournies par Apple étaient scriptables. 36 ans plus tard, on peut dire qu'AppleScript est une technologie bien résistante : dans MacOS Tahoe, TextEdit, Pages, Numbers, iCal, iMovie, Mail, Safari... sont toujours scriptables, le Finder est scriptable et enregistrable et cerise sur le gâteau les dictionnaires ont été revus pour y ajouter du vocabulaire JavaScript et Objective-C depuis MacOX 10.11, permettant d'écrire des scripts dans ces langages qui interagissent directement avec les applications. Enfin, une interaction entre AppleScript et Terminal a été développée pour pouvoir lancer des scripts Unix depuis l'éditeur de scripts. En outre, Automator fourni avec MacOS permet la mise en œuvre simple de processus automatiques sans utiliser de scripting, il est extensible en utilisant AppleScript, JavaScript ou des commandes du shell. Les utilisateurs avertis disposent ainsi d'une boîte à outils très complète pour étendre les possibilités des applications.



16 SEPTEMBRE 1985 : STEVE JOBS PART !

Le 12 septembre, Jobs annonce au conseil d'administration qu'il pensait, avec quelques salariés sans grande importance, monter une nouvelle société pour créer une machine 3M. Il demande si Apple pourrait lui accorder une licence du Macintosh. Il met en balance sa démission de son poste de président et garantit que la nouvelle société ne concurrencerait pas Apple. Le conseil décide de ne pas accepter la démission et demande d'attendre quelques jours.

Le lendemain matin, Jobs présente à Sculley la liste des personnes sans importance, qui pourraient partir avec lui. Parmi les 5 noms, il y a Crow, Page et Tribble, tout sauf des salariés sans importance. Ces ingénieurs travaillaient au projet Big Mac. Sculley s'insurge et le conseil d'administration veut lui retirer la présidence du constructeur. Le 16 septembre, Jobs démissionne et il envoie des copies de sa lettre de démission à plusieurs journaux.

La situation ne se calme pas. Une semaine plus tard, le constructeur porte plainte contre Jobs et Page et veut interdire toute utilisation d'informations internes dans leur future société. La plainte évoque des manquements de Jobs à son poste de président. La direction se retrouve dans une situation paradoxale : d'un côté elle a critiqué Jobs pour son incompétence à gérer des équipes et là, au contraire, elle donne du crédit au cofondateur. Trois mois plus tard, l'affaire est enterrée.

Du Copier/Coller aux Writing Tools d'Apple Intelligence

L'invention du copier/coller au Xerox PARC n'était pas seulement une commodité d'interface : elle introduisait une abstraction durable où le contenu devient indépendant de l'outil qui l'a produit. En séparant la donnée de son origine, cette idée a ouvert un espace de transformation et de circulation entre logiciels. Quarante ans plus tard, les Writing Tools prolongent exactement ce principe : le système peut désormais non seulement transporter un texte, mais aussi le comprendre, le reformuler et l'enrichir.



Etienne Vautherin

Précédemment :

- *"Technology Evangelist" chez Apple,*
- *"iOS Evangelist" chez Orange,*

Actuellement :

- *CTO pour OPlife*

Du Note Pad aux Writing Tools

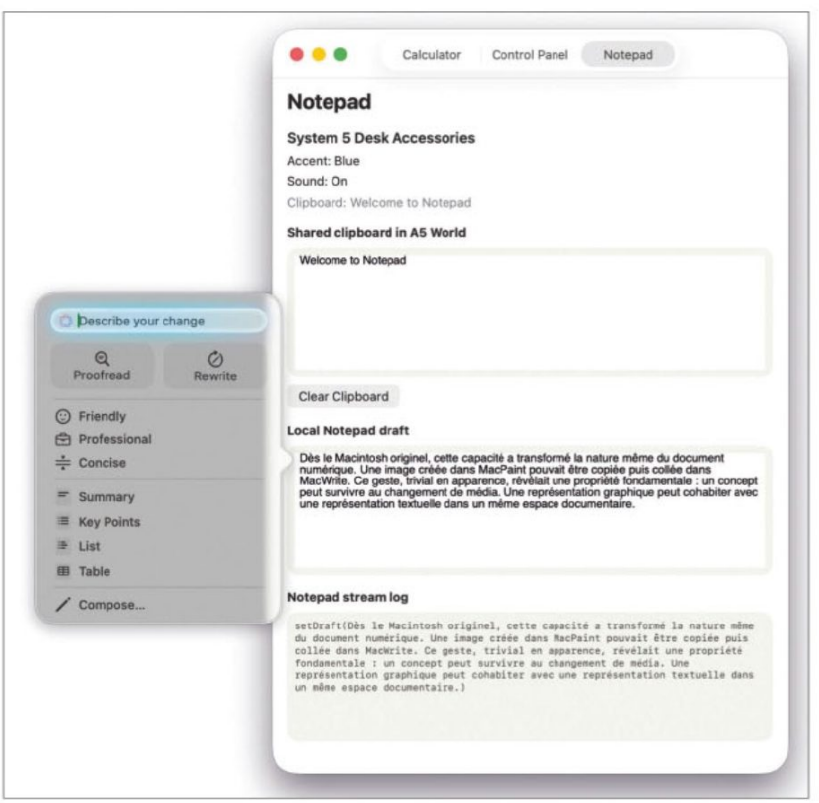
Imaginons un instant le Note Pad du Macintosh original (cet accessoire minimaliste toujours disponible depuis le menu Pomme) doté des Writing Tools d'Apple Intelligence. L'expérience serait étonnamment cohérente avec l'esprit du Macintosh de 1984. Le Note Pad n'était pas un traitement de texte complet comme MacWrite ; c'était un espace de pensée rapide, un endroit où l'on pouvait noter une idée avant de la structurer ailleurs. Si cet accessoire avait aujourd'hui accès aux capacités d'Apple Intelligence, il deviendrait un véritable laboratoire d'écriture. Une simple note pourrait être reformulée, condensée ou développée en quelques secondes. L'utilisateur pourrait changer le ton d'un texte, corriger son style ou produire un résumé directement depuis ce petit carnet. On pourrait imaginer un menu contextuel proposant plusieurs reformulations, comme si le petit accessoire historique recevait soudain la puissance sémantique d'un système contemporain. L'outil resterait simple, fidèle à sa vocation d'origine, mais il deviendrait capable d'accompagner la pensée plutôt que de simplement l'enregistrer. Cette projection imaginaire ne relève pas seulement du jeu de style : elle révèle une continuité profonde entre les abstractions logicielles du Macintosh initial et les mécanismes modernes d'Apple Intelligence.

Copier / Coller : une abstraction fondatrice

La brochure originale du Macintosh mettait en scène une démonstration devenue emblématique : un dessin créé dans MacPaint pouvait être copié puis collé dans MacWrite. Pour l'utilisateur, le geste semblait presque magique. Pourtant, derrière cette simplicité apparente se cachait une idée décisive : le contenu devenait partiellement indépendant de l'application qui l'avait produit. Techniquement, cette capacité reposait sur le Scrap Manager, qui permettait de déposer dans une zone partagée une ou plusieurs représentations d'un même contenu. L'application qui recevait l'opération de collage n'avait pas besoin de connaître l'application source ; elle devait seulement reconnaître un format qu'elle savait interpréter. Cette abstraction introduisait une forme de contrat d'échange entre logiciels. MacPaint et MacWrite n'étaient plus seulement deux applications distinctes : ils devenaient deux participants d'un même espace documentaire. C'est là une idée extraordinairement moderne. Les Writing Tools prolongent exactement ce principe. Lorsqu'un utilisateur sélectionne un texte dans une vue éditable et demande un résumé, une reformulation ou un changement de ton, l'application hôte n'a pas besoin d'embarquer elle-même toute l'intelligence de transformation. Elle expose un contenu sélectionnable ; le système applique ensuite une opération de haut niveau sur ce contenu. Hier, le système transportait des données d'une application à l'autre ; aujourd'hui, il peut aussi les transformer.

L'objet prêt à l'emploi

Pour un développeur, la vraie élégance d'une plateforme ne réside pas seulement dans ses abstractions théoriques, mais dans la manière dont elles deviennent immédiatement utilisables. Très tôt, MacApp a incarné cette idée en proposant

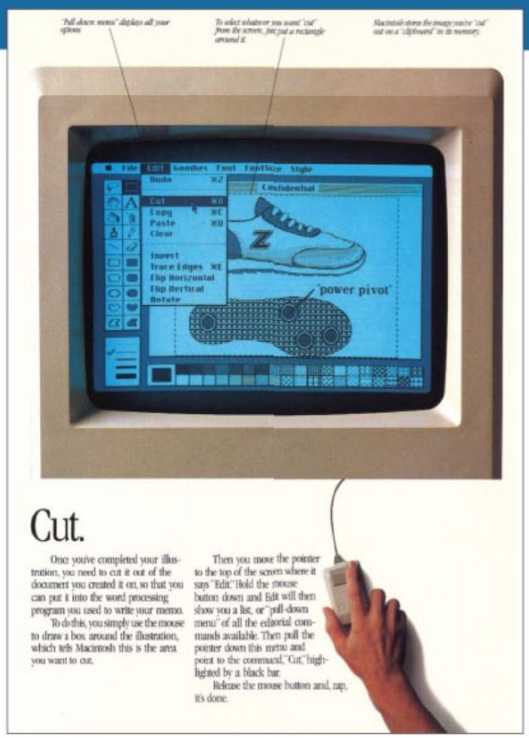


des objets prêts à l'emploi intégrant déjà les comportements standards attendus sur Macintosh. Un développeur n'avait pas à reconstruire manuellement la sélection, l'édition ou l'intégration au mécanisme de copier/coller : le framework fournissait des classes capables de prendre en charge ces usages. Cette logique a traversé les décennies. Avec UITextView, puis avec les composants textuels modernes des plateformes Apple, le développeur bénéficie automatiquement de la sélection, du menu d'édition, du presse-papiers système et, désormais, des Writing Tools. L'intérêt architectural est majeur : l'abstraction d'échange et de transformation n'est pas codée à la main dans chaque application, elle est *héritée à travers un composant* standard. Autrement dit, le développeur n'implémente pas l'intelligence ; il adopte l'objet qui sait dialoguer avec elle. C'est exactement la promesse du "prêt à l'emploi" telle qu'Apple l'a progressivement raffinée depuis MacApp.

Une continuité de plateforme

Imaginer le Note Pad historique dialoguant avec Apple Intelligence n'est donc pas une fantaisie sans portée. C'est une manière très concrète de lire quarante ans de conception logicielle chez Apple. Le copier/coller montrait déjà qu'un système bien conçu pouvait fournir une abstraction générale d'échange entre applications. MacApp, puis UITextView, ont montré qu'un framework mature pouvait encapsuler cette abstraction dans un objet standard, immédiatement exploitable par les développeurs. Les Writing Tools ajoutent une étape supplémentaire : le système ne se contente plus de faire circuler le contenu, il devient capable de l'analyser et de le transformer. Mais le principe, lui, reste remarquablement stable. Pour le développeur logiciel, la leçon est précieuse : lorsqu'une plateforme propose les bons objets, l'innovation la plus puissante devient souvent accessible sans effort supplémentaire.

Retrouvez plus de contenu, et des liens sur ce sujet, sur le site https://murmurations.dev/fr/

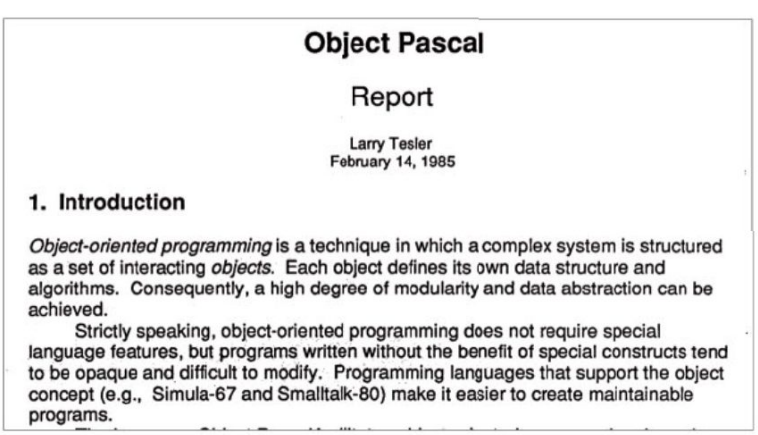






LARRY TESLER, LE COPIER/COLLER EN OBJECT PASCAL

Après avoir conçu au Xerox PARC l'une des interactions les plus durables de l'informatique moderne — le copier/coller — Larry Tesler rejoint Apple au tournant des années 1980, au moment où le projet Lisa prend forme. Chez Apple, son rôle ne se limite pas à l'interface utilisateur : il contribue aussi à structurer les outils conceptuels permettant de construire des applications complexes. Dans ce contexte, Tesler formalise Object Pascal, une extension objet du Pascal utilisée sur Lisa puis sur Macintosh. Ce langage introduit des notions d'encapsulation et d'héritage adaptées à la construction d'interfaces graphiques riches. Object Pascal devient ainsi le socle de MacApp, l'un des premiers frameworks applicatifs destinés aux développeurs Macintosh. Grâce à lui, la programmation de l'interface passe d'un assemblage de procédures à une véritable architecture d'objets, ouvrant la voie à une nouvelle génération d'outils logiciels.



Object Pascal

Report

Larry Tesler
February 14, 1985

1. Introduction

Object-oriented programming is a technique in which a complex system is structured as a set of interacting *objects*. Each object defines its own data structure and algorithms. Consequently, a high degree of modularity and data abstraction can be achieved.

Strictly speaking, object-oriented programming does not require special language features, but programs written without the benefit of special constructs tend to be opaque and difficult to modify. Programming languages that support the object concept (e.g., Simula-67 and Smalltalk-80) make it easier to create maintainable programs.

Switcher et Swift 6

LA CRÉATION DE MONDES ISOLÉS

Cet article propose un pont entre deux générations de développeurs. Il permet aux ingénieurs d'aujourd'hui de comprendre le rôle qu'a joué l'A5 World dans l'architecture du Macintosh original, tout en offrant à ceux qui ont connu ces machines une manière d'aborder les concepts modernes de Swift Concurrency. En rapprochant ces deux univers, il montre que les abstractions contemporaines prolongent souvent des intuitions techniques déjà présentes dans les premières architectures du Mac.



Etienne Vautherin

Précédemment :

- *"Technology Evangelist" chez Apple,*
- *"iOS Evangelist" chez Orange,*

Actuellement :

- *CTO pour OPlife*

Réinterpréter le Macintosh avec Swift Concurrency

Le Macintosh original reposait sur trois idées structurantes : un monde global par application (A5 World), une boucle d'événements centrale (GetNextEvent) et un mécanisme de changement de contexte introduit par Switcher puis MultiFinder. Le code présenté ici propose une lecture moderne de ces principes : un actor pour représenter l'espace global, des AsyncStream pour reproduire la boucle d'événements et des snapshots cohérents pour simuler la restauration de contexte. L'objectif est de montrer comment ces intuitions historiques peuvent être reformulées avec les abstractions de Swift Concurrency.

L'actor AppA5World : une île partagée

Dans ce code, le type AppA5World est déclaré comme un actor. En Swift Concurrency, un actor est un type de référence conçu pour protéger l'accès concurrent à un état mutable. Contrairement à une classe ordinaire, son état interne n'est accessible que par une seule tâche à la fois. Toute interaction avec cet état doit passer par des méthodes asynchrones (await), ce qui garantit qu'aucune condition de course ne peut se produire.

Conceptuellement, un actor agit comme une petite île de cohérence. Les tâches concurrentes peuvent envoyer des requêtes vers cette île, mais elles ne peuvent jamais modifier son état simultanément. Swift confie au runtime la responsabilité d'ordonner ces accès de manière sûre. Ce modèle remplace avantageusement les mécanismes traditionnels de synchronisation comme les verrous (NSLock) ou les files sérialisées.

```
actor A5World {
  struct SharedGlobals: Sendable {
    var appName: String
    var accentColorName: String
    var soundEnabled: Bool
    var clipboardText: String
    var openedAccessories: Set<HistoricalAccessory>
  }

  private var globals: SharedGlobals

  init(appName: String = "System 5 Desk Accessories") {
    self.globals = SharedGlobals(
      appName: appName,
      accentColorName: "Blue",
      soundEnabled: true,
      clipboardText: "Welcome to Notepad",
      openedAccessories: [.calculator, .controlPanel, .Notepad]
    )
  }
    ...
}
```

L'actor AppA5World stocke une structure interne appelée SharedGlobals, qui regroupe les variables globales partagées par l'application : nom de l'app, couleur d'accentuation, état du son, contenu du presse-papiers et liste des accessoires ouverts. Toutes les mutations passent par des méthodes telles que setAccentColorName, setSoundEnabled ou setClipboardText. Une méthode snapshot() permet également d'obtenir une copie cohérente de cet état global.

Cette idée de « monde partagé » fait directement écho à un mécanisme historique du Macintosh original : l'**A5 World**. Sur les machines 68k des années 1980, le registre processeur **A5** était utilisé comme pointeur vers les variables globales d'une application. Lorsqu'une application devenait active, le système chargeait son pointeur A5, donnant ainsi accès à son ensemble de données globales. Lorsque l'on passait à une autre application, le système sauvegardait l'ancien A5 et chargeait celui de la nouvelle application.

Chaque application possédait ainsi son propre **A5 World**, c'est-à-dire un espace cohérent contenant toutes ses variables globales et ses structures de données partagées. Les accessoires de bureau, bien qu'intégrés au système, pouvaient également accéder à cet espace lorsqu'ils s'exécutaient dans le contexte d'une application.

L'actor AppA5World reproduit conceptuellement ce mécanisme dans un environnement moderne. Il joue le rôle d'un espace global partagé, accessible depuis plusieurs composants — ici les accessoires Calculator, Control Panel et Notepad. Chacun peut lire ou modifier l'état commun en envoyant des requêtes à cet actor.

La différence majeure réside dans la manière dont la cohérence est assurée. Dans le Macintosh original, cette cohérence reposait essentiellement sur un modèle mono-tâche : un seul morceau de code s'exécutait à la fois, ce qui évitait les conflits d'accès. Dans un système moderne, plusieurs tâches

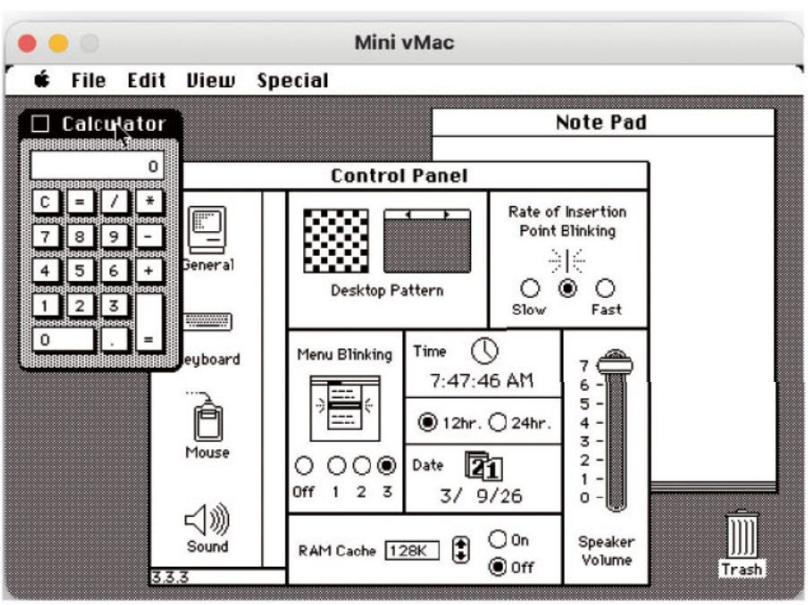
peuvent s'exécuter en parallèle. L'actor remplit alors le rôle qu'assurait implicitement le moteur unique du Macintosh : garantir qu'un seul accès modifie l'état partagé à un instant donné.

On peut donc voir AppA5World comme une réinterprétation contemporaine de l'A5 World historique. Dans le Macintosh classique, le registre A5 pointait vers un bloc mémoire contenant les variables globales de l'application. Dans ce code Swift, l'actor encapsule cet espace global et en régule l'accès. Le principe reste le même : un lieu unique où réside l'état commun du système, auquel les différents composants viennent se connecter pour lire ou modifier les informations partagées.

ANDY HERTZFELD, ACCESSOIRES ET SWITCHER



Parmi les ingénieurs clés de l'équipe Macintosh originale, Andy Hertzfeld a joué un rôle déterminant dans la conception de mécanismes qui ont profondément marqué l'expérience utilisateur du système. Il participe à la mise en place des Desk Accessories, de petits programmes intégrés au système — calculatrice, carnet, tableau de bord — capables de s'exécuter à l'intérieur d'une application sans la quitter. Pour rendre cela possible, le système exploite le contexte global de l'application, le fameux A5 World, auquel ces accessoires peuvent accéder. Quelques années plus tard, Hertzfeld pousse plus loin cette idée avec Switcher, un utilitaire permettant de passer rapidement d'une application à une autre en sauvegardant et restaurant leur contexte. Switcher introduit ainsi une première forme de multitâche coopératif sur Macintosh et prépare directement l'arrivée du MultiFinder. Par cette double contribution — accessoires et commutation d'applications — Hertzfeld a façonné une architecture simple mais extrêmement élégante, où le changement de contexte devient une opération légère et quasi instantanée.



AccessoryEventChannel : recréer une boucle d'événements

Le type AccessoryEventChannel repose sur AsyncStream, une primitive de Swift Concurrency qui permet de produire et de consommer une suite asynchrone de valeurs. Un AsyncStream agit comme un canal : une partie du programme peut y publier des événements grâce à une Continuation, tandis qu'une autre partie peut les lire avec une boucle for await. Cette construction permet de représenter naturellement des flux d'événements produits au fil du temps, comme des interactions utilisateur ou des messages provenant d'un système.

Ce modèle rappelle fortement l'architecture du Macintosh original. Sur les premières versions du système, les applications étaient structurées autour d'une boucle d'événements appelée GetNextEvent. Cette fonction interrogeait le système pour savoir si un événement était disponible : clic souris, frappe clavier, mise à jour d'une fenêtre, etc. L'application exécutait alors une boucle infinie qui appelait GetNextEvent, puis dispatchait l'événement vers le code approprié. Toute la logique interactive du programme était organisée autour de cette boucle centrale.

```
@MainActor
final class AccessoryEventChannel<Event: Sendable> {
  let stream: AsyncStream<Event>
  private let continuation: AsyncStream<Event>.Continuation

  init() {
    let (stream, continuation) = AsyncStream.makeStream(of: Event.self)
    self.stream = stream
    self.continuation = continuation
  }

  func post(_ event: Event) {
    continuation.yield(event)
  }

  func finish() {
    continuation.finish()
  }
}
```

Le code de AccessoryEventChannel reproduit conceptuellement ce mécanisme, mais avec les outils modernes de Swift Concurrency. La méthode AsyncStream.makeStream crée deux éléments : le flux (stream) et sa continuation. Les vues SwiftUI publient des événements dans le flux en appelant post, qui invoque continuation.yield. De l'autre côté, le runtime de l'accessoire exécute une boucle for await event in channel.stream. Cette boucle joue exactement le rôle de la boucle GetNextEvent historique : elle attend le prochain événement puis déclenche le traitement correspondant.

La différence essentielle est que l'attente n'est plus active. Dans le Macintosh original, l'application appelait répétitivement GetNextEvent dans une boucle. Avec AsyncStream, la suspension est gérée par le runtime asynchrone : la tâche se met en attente jusqu'à ce qu'un événement soit produit. On obtient

ainsi l'équivalent moderne de la boucle d'événements classique, mais intégré dans le modèle de concurrence structuré de Swift.

Simuler Switcher et MultiFinder avec un snapshot d'actor

Dans le Macintosh classique, l'arrivée de Switcher puis du MultiFinder a introduit une forme de coexistence entre plusieurs applications. Le système conservait pour chacune son contexte d'exécution, notamment son A5 World qui contenait les variables globales et l'état partagé de l'application. Lorsqu'un utilisateur passait d'une application à une autre, le système restaurait simplement le pointeur A5 correspondant. L'illusion du multitâche reposait ainsi sur une opération très simple : sauvegarder un état global puis en restaurer un autre.

Dans ce code Swift, l'actor AppA5World joue le rôle de ce monde global. Toutes les données communes sont stockées dans la structure SharedGlobals. La méthode snapshot() retourne une copie cohérente de cet état. Comme l'accès à l'actor est sérialisé par Swift Concurrency, ce snapshot représente un instant stable du monde partagé, exactement comme le bloc mémoire pointé par le registre A5 à un moment donné.
Le modèle devient particulièrement intéressant lorsque plusieurs composants accèdent à ce monde partagé. Chaque accessoire peut modifier l'état de l'actor, puis la vue principale récupère régulièrement un snapshot à l'aide de await a5World.snapshot(). Cette opération équivaut conceptuellement à la restauration d'un contexte lors d'un changement d'application : l'interface utilisateur se met à jour pour refléter l'état courant du système.

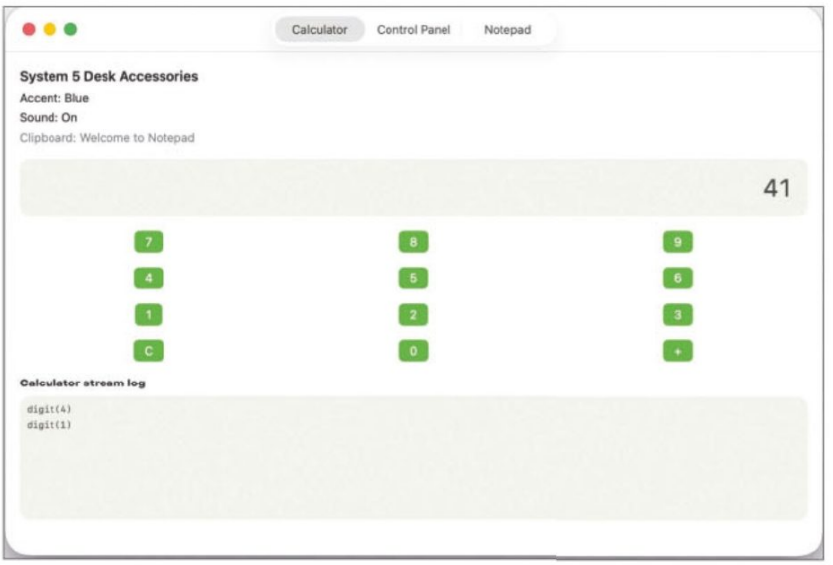
Autrement dit, le snapshot agit comme une photographie du monde global. Lorsqu'il est injecté dans le modèle observable de l'interface (AppModel.sharedGlobals), SwiftUI redessine les vues avec cet état. On retrouve ainsi une idée très proche du fonctionnement de Switcher et du MultiFinder : plusieurs composants existent simultanément, mais l'utilisateur voit toujours une représentation cohérente de l'état global au moment où le système lui redonne la main.

```
actor A5World {
    ...
  func snapshot() -> SharedGlobals {
    globals
  }

  func setAccentColorName(_ name: String) {
    globals.accentColorName = name
  }

  func setSoundEnabled(_ enabled: Bool) {
    globals.soundEnabled = enabled
  }

  func setClipboardText(_ text: String) {
```




```
    globals.clipboardText = text
  }

  func openAccessory(_ accessory: HistoricalAccessory) {
    globals.openedAccessories.insert(accessory)
  }

  func closeAccessory(_ accessory: HistoricalAccessory) {
    globals.openedAccessories.remove(accessory)
  }
}
```

La différence tient à la nature du multitâche. Dans le Macintosh original, un seul programme s'exécutait réellement à la fois et le changement de contexte consistait à restaurer un pointeur mémoire. Dans ce modèle moderne, plusieurs tâches peuvent travailler en parallèle, mais l'actor garantit que l'état global reste cohérent. Le snapshot fournit alors l'équivalent contemporain du changement de contexte historique : une reconstruction sûre et instantanée du monde partagé.

Continuité d'idées, évolution des outils

Les mécanismes du Macintosh original — A5 World, boucle GetNextEvent, changement de contexte avec Switcher ou MultiFinder — reposaient sur une architecture simple où la cohérence du système provenait d'un moteur unique et d'un état global bien défini. Avec Swift Concurrency, ces mêmes idées réapparaissent sous d'autres formes : actors pour protéger l'état partagé, AsyncStream pour modéliser les flux d'événements et snapshots cohérents pour reconstruire le monde visible de l'application. Les outils ont changé, mais l'intuition architecturale demeure étonnamment proche.
Retrouvez plus de contenu, et des liens sur ce sujet, sur le site https://murmurations.dev/fr/

De QuickDraw à SwiftUI : l'algèbre invisible des interfaces

Cet article offre au développeur d'aujourd'hui une double lecture précieuse : il lui permet de redécouvrir, sous la syntaxe de SwiftUI, l'intuition mathématique qui structurait déjà les régions QuickDraw. En suivant ce fil, on voit apparaître la composition non comme un simple procédé graphique, mais comme une véritable algèbre de formes, où union, intersection, différence et xor deviennent les opérations fondatrices d'une pensée visuelle que SwiftUI prolonge avec une élégance nouvelle.



Etienne Vautherin

Précédemment :
- *"Technology Evangelist" chez Apple,*
- *"iOS Evangelist" chez Orange,*

Actuellement :
- *CTO pour OPlife*

La primitive géométrique

La conception de QDRegionView reprend cette idée en introduisant une structure QDRegion décrivant une expression de région. Chaque région est représentée comme un arbre d'opérations : une primitive géométrique (rectangle, ellipse, chemin) ou une combinaison de deux régions selon une opération booléenne. Cette représentation récursive reflète la logique de QuickDraw : une région peut être simple ou résulter d'une suite d'opérations appliquées à d'autres régions.

```
struct QDRegion {
  indirect enum Op {
    case primitive(Path)
    case union(Op, Op)
    case intersection(Op, Op)
    case difference(Op, Op)
    case xor(Op, Op)
  }

  let op: Op

  static func rect(_ rect: CGRect) -> QDRegion {
    QDRegion(op: .primitive(Path(rect)))
  }

  static func ellipse(in rect: CGRect) -> QDRegion {
    QDRegion(op: .primitive(Path(ellipseIn: rect)))
  }

  func union(_ other: QDRegion) -> QDRegion {
    QDRegion(op: .union(op, other.op))
  }

  func intersect(_ other: QDRegion) -> QDRegion {
    QDRegion(op: .intersection(op, other.op))
  }

  func subtract(_ other: QDRegion) -> QDRegion {
    QDRegion(op: .difference(op, other.op))
  }

  func xor(_ other: QDRegion) -> QDRegion {
    QDRegion(op: .xor(op, other.op))
  }
}
```

Les méthodes union, intersect, subtract et xor ne réalisent pas immédiatement le calcul géométrique. Elles construisent plutôt une nouvelle expression de région. Cette approche différée correspond à une forme d'algèbre déclarative : la région est décrite comme une composition logique avant d'être matérialisée graphiquement. Le rôle de QDRegionView est précisément d'interpréter cette expression.

Le rendu avec un contexte moderne

Le rendu est assuré par un Canvas, qui fournit un contexte graphique moderne dans SwiftUI. Lors du dessin, l'arbre d'opérations est parcouru récursivement. Les primitives produisent des Path, tandis que les opérations composées utilisent des couches de dessin et des opérations de clipping pour reproduire les effets d'union, d'intersection ou de différence. Cette étape correspond à la matérialisation visuelle de l'expression algébrique.

```
struct QDRegionView: View {
  let region: QDRegion.Op

  var body: some View {
    Canvas { context, size in
```

Regions

Unlike most graphics packages that can manipulate only simple geometric structures (usually rectilinear, at that), QuickDraw has the ability to gather an arbitrary set of spatially coherent points into a structure called a region, and perform complex yet rapid manipulations and calculations on such structures. Regions not only make your programs simpler and faster, but will let you perform operations that would otherwise be nearly impossible.

You define a region by calling routines that draw lines and shapes (even other regions). The outline of a region should be one or more closed loops. A region can be concave or convex, can consist of one area or many disjoint areas, and can even have "holes" in the middle. In Figure 5, the region on the left has a hole in the middle, and the region on the right consists of two disjoint areas.

The Mathematical Foundation of QuickDraw I-141

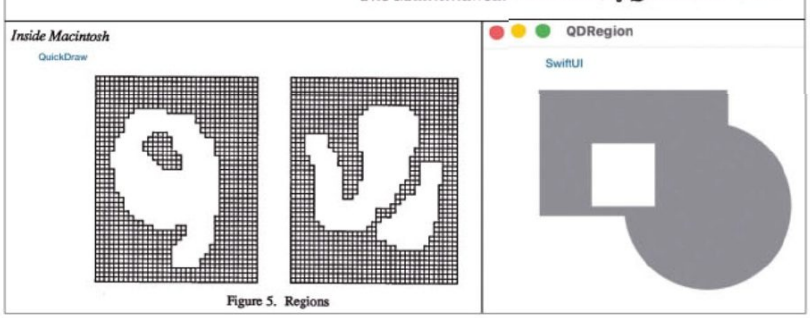


Figure 5. Regions

```
        let rect = CGRect(origin: .zero, size: size)
        draw(region, in: rect, context: &context)
      }
    }

    private func draw(
      _ op: QDRegion.Op,
      in rect: CGRect,
      context: inout GraphicsContext
    ) {
      switch op {
      case .primitive(let path):
        context.fill(path, with: .foreground)

      case .union(let a, let b):
        draw(a, in: rect, context: &context)
        draw(b, in: rect, context: &context)

      case .intersection(let a, let b):
        context.drawLayer { layer in
          layer.clip(to: materialize(a, in: rect))
          layer.clip(to: materialize(b, in: rect))
          layer.fill(Path(rect), with: .foreground)
        }

      case .difference(let a, let b):
        context.drawLayer { layer in
          layer.clip(to: materialize(a, in: rect))
          layer.clip(to: materialize(b, in: rect), options: .inverse)
          layer.fill(Path(rect), with: .foreground)
        }

      case .xor(let a, let b):
        draw(.union(.difference(a, b), .difference(b, a)), in: rect, context: &context)
      }
    }

    private func materialize(_ op: QDRegion.Op, in rect: CGRect) -> Path {
      switch op {
      case .primitive(let path):
        return path

      case .union(let a, let b):
        var p = materialize(a, in: rect)
        p.addPath(materialize(b, in: rect))
        return p

      case .intersection, .difference, .xor:
        // SwiftUI does not expose full path boolean ops like QuickDraw regions.
        // For true geometric boolean composition, bridge to Core Graphics / CGPath
        // or use a custom rasterization/vector boolean engine.
        return Path(rect)
      }
    }
  }
```

Cette architecture présente deux avantages. D'une part, elle conserve l'esprit du modèle QuickDraw : une région reste un objet abstrait composable indépendamment de son rendu. D'autre part, elle s'intègre naturellement dans SwiftUI, où la vue devient l'interpréteur d'une description déclarative de la géométrie. La région n'est plus seulement un artefact graphique : elle devient une structure algébrique que la vue transforme en image.



BILL ATKINSON, CATALYSEUR DE GÉNIE

Lorsque Steve Jobs recrute Bill Atkinson aux prémices du projet Lisa, celui ci est encore étudiant en neurosciences à l'Université de Washington. Jobs perçoit pourtant immédiatement chez lui une capacité rare : la faculté de penser les structures profondes d'un système plutôt que de simplement écrire du code. Dans l'environnement créatif qu'il façonne autour du Macintosh, Jobs pousse Atkinson à dépasser son domaine initial. Les modèles de perception visuelle étudiés en neurosciences se transforment peu à peu en intuitions mathématiques sur la géométrie, les transformations et la représentation de l'image. Atkinson apprend alors à traduire ces idées en algorithmes concrets et en structures logicielles élégantes. De cette rencontre entre intuition scientifique, rigueur mathématique et exigence d'ingénierie naissent QuickDraw, les régions, ou encore l'HyperCard originel. Steve Jobs n'a pas seulement recruté un programmeur : il a créé les conditions dans lesquelles un jeune chercheur pouvait devenir l'un des architectes les plus visionnaires du logiciel moderne. La vue SwiftUI `QDRegionView` s'inspire directement du modèle conceptuel des régions QuickDraw. Dans QuickDraw, une région n'est pas seulement une forme graphique : c'est un ensemble de points du plan sur lequel on peut appliquer des opérations algébriques. Les appels `UnionRgn`, `SectRgn`, `DiffRgn` ou `XorRgn` permettent de composer des régions pour produire une nouvelle région, tout en restant dans le même espace d'objets. Autrement dit, l'ensemble des régions est fermé par ces opérations. Une région combinée reste elle-même une région, susceptible d'être réutilisée dans d'autres compositions.

L'algèbre silencieuse

En revisitant les régions QuickDraw à la lumière de SwiftUI, on découvre qu'une grande part de la modernité des interfaces graphiques était déjà présente dans l'algèbre silencieuse du Macintosh originel. Union, intersection ou différence n'étaient pas seulement des opérations graphiques, mais les briques d'une pensée compositionnelle. SwiftUI n'en est pas la rupture : il en révèle la continuité, en donnant aujourd'hui une forme déclarative et lisible à cette géométrie du logiciel. Retrouvez plus de contenu, et des liens sur ce sujet, sur le site https://murmurations.dev/fr/



Dominique BULTEZ

Ingénieur radioélectricité et électronique de l'INPG de Grenoble, puis ingénieur Brevet chez Thomson, puis employé No 20525 d'Apple France, j'ai rejoint France Télécom/Orange jusqu'à la retraite qui me permet de me consacrer aux applications des microcontrôleurs aux trains miniatures chez Locoduino.org (voir **www.bultez.fr**).

Apple et les télécommunications

J'ai eu le très grand honneur d'être recruté chez Apple Computer France de 1988 à 1997. Après avoir été l'un des créateurs/animateurs du premier club de micro-informatique à Paris (AFIN-CAU) et développé sur Mac, en MPW, un logiciel de traduction de tous types de documents en Télétex grâce au génial « bottleneck » qui permettait d'adresser l'écran graphique et l'imprimante par les mêmes appels de fonctions, j'ai été remarqué pour avoir rendu Mac friendly la communication interne des Macs avec le système informatique (IBM) de gestion.

Ma mission consistait à permettre aux développeurs de produits de télécommunication, alors quasiment tous sur PC, de rejoindre le Mac pour le développement de leur chiffre d'affaires et la fierté de meilleurs produits.

Les modems et applications de communication

On a vu les vitesses des modems évoluer de 300 bits/seconde (V21), à 14400 (V32) bits/seconde, en passant par 1200 (V22), 2400 (V26), 9600 (V29). Le catalogue des modems s'est ainsi étoffé avec une offre française florissante (COM1, Kortex, Olitec, PNB, SAT, Alcatel) à côté d'Apple, Hayes, Motorola, US Robotics, etc.

Ainsi est né le en 1989 le Macintosh PowerBop équipé d'un modem radio CT2 (conçu par la R&D Europe) pour le réseau Bi-Bop, grâce auquel j'ai eu mon premier téléphone portable (mais pas complètement mobile).

Avant l'Internet, les BBS

Avant que le Web et l'Internet grand public n'existent, les gens se connectaient déjà. Les **BBS** et les premiers services en ligne comme **AOL, eWorld ou CompuServe** ont été les ancêtres de nos réseaux sociaux, forums et messageries actuelles.

Comment cela marchait? L'application dédiée composait, via le modem, un numéro de téléphone. Elle se connectait à un seul serveur.

On pouvait lire / poster des messages (forums primitifs). Télécharger des fichiers et un peu plus.
On demandait gratuitement des disquettes ou un CDRom permettant d'installer et configurer tout ce qu'il faut

eWorld a été le meilleur service de type BBS made in Apple. Il a existé de **1994 à 1996.**

Les Kits de communication Internet

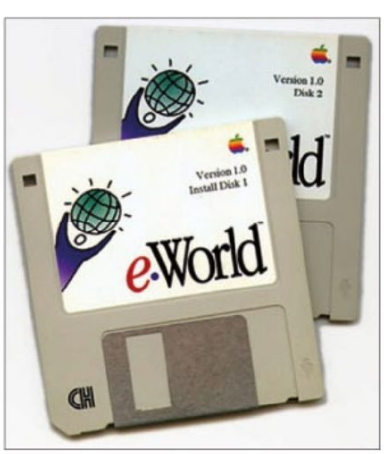
Ce n'est qu'à partir de System 7.5 (sorti en 1994) que MacTCP a commencé à être fourni gratuitement avec le système, ce qui a ouvert la connectivité Internet à un public beaucoup plus large de propriétaires de Mac.

À partir de 1995-1996, Apple a progressivement remplacé MacTCP par Open Transport, une nouvelle pile réseau intégrée directement dans le système, plus moderne et plus complète.

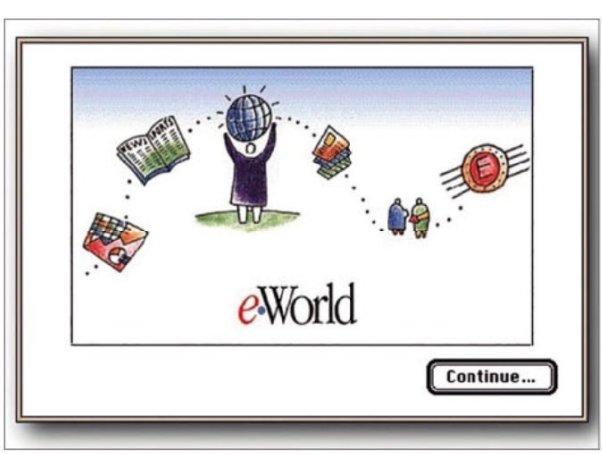
Le premier kit Internet pour Macintosh / grand public en France ne vient pas directement des opérateurs télécoms historiques, mais de FAI indépendants comme Calvacom, FranceNet en 1994, avec des disquettes et CD-ROM de connexion à l'Internet via ligne téléphonique.

Les opérateurs comme Wanadoo/France Télécom ont rejoint ce marché un peu plus tard, à partir de 1996–1997, avec des kits modem + abonnement plus packagés.

Alors Apple a fourni son propre Kit.
Aux USA, avec seulement 2 opérateurs, la configuration était







simple. Mais en Europe, avec une dizaine de FAI par pays, il a fallu ruser.

J'ai eu la chance et le bonheur de pouvoir piloter le développement et l'intégration du kit internet Apple inclus dans Mac OS 8 en 1997, avec « carte blanche » de Cupertino. Ce kit a d'ailleurs intégré Wanadoo et les autres opérateurs français en tant qu'option d'accès Internet par défaut pour la France ainsi que les opérateurs des autres pays européens, fourris par les autres Apple d'Europe.

Un assistant graphique guidait l'utilisateur pour choisir son FAI (ex. *Wanadoo, Club Internet, World NET pour la France*), entrer le numéro de téléphone, login et mot de passe, installer et configurer MacPPP si nécessaire pour une connexion modem PPP. Clients et utilitaires étaient inclus.

L'Assistant de configuration Internet de Mac OS 8 est un excellent exemple de l'esthétique d'Apple à la fin des années 90, avec son interface "Platinum" grise et ses icônes colorées.

Le RNIS (ISDN) et la visioconférence

À côté de l'X25, réseau numérique Transpac de France Télécom utilisé pour les services bancaires et Télétel, il nous fallait une offre adaptée à Numéris, le réseau RNIS français et étranger (2x64 + 16 kb/s) qui ouvrait la porte aux services sophistiqués de transferts de fichiers, d'impressions à distance, d'interconnexions de réseau local de téléphonie et de visiophone.

Grâce aux JDD (journées du développement, version française de la WWDC), tous les développeurs de télécom étaient présents : à ceux cités plus haut s'ajoutaient Accès Privilège, Planet-ISDN (Euronis), OST, SCII Télécom, etc..).

L'ambiance et la camaraderie étaient formidables, ce qui m'aidait à faire comprendre et appliquer les « Human Interface Guidelines » si nécessaires pour configurer et utiliser facilement et intuitivement ces produits complexes.

Grâce à eux et mon lobby effréné, accompagné de France Télécom, à cette époque du Macintosh II, les cartes RNIS ont commencé à fleurir, y compris chez Apple à Cupertino. Pour la petite histoire, l'acronyme américain ISDN est passé de « I Still Don't Need » à « I Smell Dollars Now » et c'est parti !

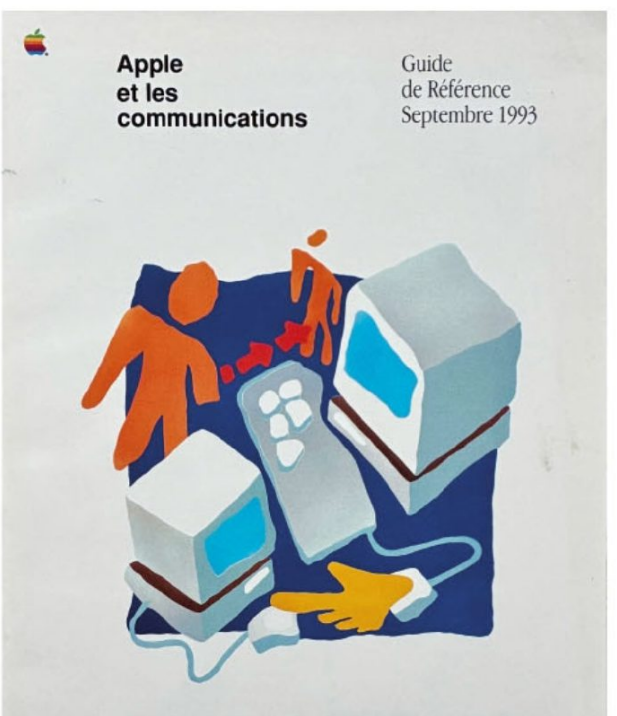
Aider les développeurs externes et internes sur le plan technique doit s'accompagner d'un support marketing très puissant comme les conférences à Apple Expo et ailleurs.

Michael Spindler alors président d'Apple Inc. m'a demandé une démonstration de visioconférence à la Cité de la Réussite (Sorbonne) le 22 octobre 1994 dans le grand amphithéâtre. Avec la société Access Privilège de Philippe Boulanger et Numéris, une première visioconférence entre Paris et Sophia-Antipolis a eu lieu en plein écran du nouvel iMac.

Devant le succès et la liesse dans l'amphi, personne n'a pensé à garder une copie d'écran et je cherche encore quelques images de presse.







Le Guide Apple et les Communications

Conçu par Pascal DELANNOY, Chef de Produits Réseaux et Telecoms, l'édition de 1993 du Guide « Apple et les communications » au format « Inside Macintosh » nous a pris un an (avec David Fischer et Pierre Allain) pour répertorier tous les produits ayant bénéficié de notre support développeur entre autres. Cette édition reflète bien le travail acharné des développeurs et de notre support technique et marketing, dans une formidable ambiance de coopération.

50 ans d'Apple : quelle place pour les développeurs ?

Depuis cinquante ans, Apple ne fabrique pas seulement des ordinateurs. L'entreprise construit des plateformes complètes qui reposent sur une combinaison étroite entre matériel, logiciel et expérience utilisateur. Dans cette équation, les développeurs jouent un rôle central. Les applications sont le lien direct entre les technologies de la plateforme et les usages quotidiens des utilisateurs.



Baptiste Dajon

Software Engineer on Apple Platforms

Apple Account Owner & Technology Evangelist chez Decathlon

Développeur sur les plateformes Apple depuis 2008, je travaille aujourd'hui sur des applications mobiles et des expériences spatiales pour les plateformes Apple. Passionné par cet écosystème depuis mon premier iPod, je m'intéresse particulièrement à l'évolution des langages, des outils de développement et aux nouvelles capacités offertes par l'intelligence artificielle et le spatial computing.

Contact :
https://www.linkedin.com/in/baptistedajon

L'histoire d'Apple est souvent racontée à travers ses produits emblématiques. Pourtant, derrière chaque génération d'appareils se trouve également une évolution profonde du modèle de développement. Les langages, les frameworks et les outils changent, mais la philosophie reste étonnamment constante : fournir aux développeurs des briques suffisamment puissantes pour construire des expériences cohérentes avec l'ensemble de l'écosystème. Pour ma part, cette relation avec Apple a commencé bien avant d'écrire la moindre ligne de code. Mon premier contact avec un produit Apple remonte à un iMac G3 rouge que possédait un ami. À une époque où la majorité des ordinateurs étaient des boîtes beige relativement impersonnelles, voir un ordinateur coloré, transparent et presque ludique était une expérience marquante. Mais ce n'était pas seulement l'objet qui impressionnait. L'interface, la fluidité et l'attention portée au détail donnaient l'impression que l'ordinateur pouvait être à la fois puissant et amusant.

Mon premier produit Apple personnel a été un iPod mini rose de deuxième génération. Ce produit m'a fait découvrir quelque chose de fondamental : le logiciel pouvait être simple, utile et beau à la fois. Cette idée ne m'a jamais quitté et explique en grande partie pourquoi j'ai choisi de devenir développeur sur les plateformes Apple.



Mes débuts dans le développement Apple

J'ai commencé à développer pour les plateformes Apple en 2008, au moment où l'App Store venait d'ouvrir. C'était une période fascinante, car tout restait à inventer. Les développeurs découvraient une nouvelle plateforme mobile qui combinait capteurs, connectivité et interface tactile.

Ma première application professionnelle était une application permettant de localiser les distributeurs de billets autour de l'utilisateur afin de s'y rendre facilement. Ce type d'application paraît aujourd'hui banal, mais à l'époque l'intégration entre le matériel et les frameworks logiciels était déjà impressionnante. L'application utilisait CoreLocation pour récupérer la position GPS et MapKit pour afficher les distributeurs sur une carte. Dans le code, la récupération de la position passait par un CLLocationManager configuré avec un delegate.

```
- (void)locationManager:(CLLocationManager *)manager
didUpdateLocations:(NSArray<CLLocation *> *)locations {

  CLLocation *location = [locations lastObject];

  NSLog(@"%f %f",
      location.coordinate.latitude,
      location.coordinate.longitude);
}
```

Ces coordonnées étaient ensuite utilisées pour interroger un service distant et afficher les distributeurs les plus proches.

Mon premier projet indépendant était une application appelée PKM Card Helper, destinée aux joueurs du jeu de cartes Pokémon. L'application permettait de suivre les scores d'une partie et de gérer certaines mécaniques du jeu.

Ce projet m'a permis de comprendre concrètement :

- le cycle de vie d'une application iOS
- la navigation entre view controllers
- la persistance de données
- la gestion des états d'interface

C'était aussi la première fois que je livrais une application directement aux utilisateurs.

Objective-C : comprendre la machine

À cette époque, tout reposait sur Objective-C. Ce langage était pour moi une découverte totale, car il était presque exclusivement utilisé dans l'écosystème Apple. Durant mes études supérieures, j'avais choisi mon école en partie parce qu'elle proposait des modules dédiés au développement Objective-C et à Mac OS X Server. Je me souviens avoir

acheté un énorme livre pour apprendre ce langage et ses concepts. Objective-C reposait sur un système d'envoi de messages et un runtime dynamique très puissant.
Un appel de méthode simple ressemblait à ceci :

```
[self.tableView reloadData];
```

La gestion de la mémoire était également un sujet central. Avant ARC, il fallait gérer explicitement les appels à retain et release.

```
NSString *title = [[NSString alloc] initWithString:@"Hello"];
[title release];
```

Ces contraintes obligeaient les développeurs à comprendre profondément le cycle de vie des objets et la manière dont les composants interagissaient entre eux. Même si cette approche était plus complexe, elle formait une excellente école pour comprendre les bases du développement logiciel.

La plomberie du développement iOS

Une grande partie du travail ne concerne pas uniquement l'écriture du code, mais toute la plomberie qui entoure le développement. La signature des applications et la gestion des certificats font partie des premières difficultés rencontrées par les développeurs iOS. Lorsqu'un build est produit dans Xcode, plusieurs étapes techniques se produisent.
Le code Swift ou Objective-C est compilé en binaire natif. Les ressources sont intégrées dans un bundle applicatif. L'ensemble est ensuite signé avec une identité de développement. Le résultat est un bundle .app qui peut être installé sur un appareil. Lors de la distribution vers l'App Store, ce bundle est empaqueté dans une archive .ipa.
Cette chaîne garantit l'intégrité de l'application et constitue l'un des piliers de la sécurité de la plateforme. Aujourd'hui, cette plomberie est largement automatisée par Xcode, mais comprendre ces mécanismes reste essentiel pour diagnostiquer certains problèmes de build ou de distribution.

Swift : une révolution pour les développeurs Apple

En 2014, Apple annonce Swift. Pour moi, cette annonce a été un moment particulier, car je travaillais exclusivement en Objective-C depuis plusieurs années. C'était le premier changement de langage majeur de ma carrière. Swift introduisait une syntaxe moderne et des mécanismes de sécurité qui réduisaient considérablement les erreurs.
La gestion des optionals en est un bon exemple.

```
if let user = user {
  print(user.name)
}
```

Swift a également introduit Codable pour la sérialisation JSON.

```
struct ATM: Codable {
  let name: String
  let latitude: Double
  let longitude: Double
}
```



Le parsing devient alors trivial :

```
let atms = try JSONDecoder().decode([ATM].self, from: data)
```

Plus récemment, Swift Concurrency a profondément simplifié la gestion des opérations asynchrones.

```
let (data, response) = try await URLSession.shared.data(from: url)
```

Cette évolution a permis de simplifier énormément les architectures applicatives et de réduire la dépendance à certaines bibliothèques tierces comme RxSwift.

SwiftUI et le développement déclaratif

SwiftUI introduit un modèle déclaratif pour la construction des interfaces.

```
struct ScoreView: View {

  @State private var scoreA = 0
  @State private var scoreB = 0

  var body: some View {
    VStack {
      Text("A: \(scoreA) B: \(scoreB)")
      Button("Point A") {
        scoreA += 1
      }
    }
  }
}
```

L'interface devient une fonction de l'état de l'application. Ce changement de paradigme peut être déstabilisant au début, mais il permet de prototyper extrêmement rapidement et de partager une grande partie du code entre les différentes plateformes Apple.

Apple Silicon : une transition historique

L'arrivée d'Apple Silicon représente également un moment important dans l'histoire récente de la plateforme. Pour les développeurs Apple, cette transition a été étonnamment fluide. Grâce aux outils fournis par Apple, la majorité des appli-





cations ont pu être recompilées simplement pour l'architecture ARM64. Cette transition a également permis d'unifier davantage l'écosystème. Les applications iOS peuvent désormais fonctionner sur Mac, ce qui renforce l'idée d'une plateforme logicielle cohérente.

Mon rôle aujourd'hui chez Decathlon

Aujourd'hui, je suis Apple Account Owner et Technology Evangelist chez Decathlon. Mon rôle consiste à accompagner les équipes qui travaillent sur les sujets Apple afin de garantir la meilleure expérience possible pour les utilisateurs.
Cela implique de suivre les évolutions technologiques, d'aider les équipes à adopter les nouveaux frameworks et d'anticiper les changements à venir dans l'écosystème Apple. C'est également un rôle de liaison avec Apple afin de mieux comprendre les évolutions des plateformes.

IA et spatial computing

Les Foundation Models annoncés récemment représentent une évolution importante. Chez Decathlon, nous avons pu intégrer une fonctionnalité d'intelligence artificielle dans l'application Decathlon Outdoor sans avoir besoin de déployer une infrastructure serveur complexe. Le développement pour Vision Pro introduit également un nouveau paradigme. Les applications ne sont plus limitées à un écran, mais peuvent être placées dans l'espace autour de l'utilisateur. RealityKit permet par exemple de charger un modèle 3D et de l'ajouter dans une scène.

```
let entity = try await ModelEntity(named: "Tent")
let anchor = AnchorEntity(world: .zero)
anchor.addChild(entity)
```

La difficulté consiste alors à concevoir des interactions naturelles pour l'utilisateur tout en évitant de multiplier les éléments visuels dans l'espace.

Les 50 prochaines années

Depuis mes débuts en 2008, le développement sur les plateformes Apple a énormément évolué. Les langages, les frameworks et les outils ont changé, mais la philosophie reste la même : proposer une plateforme cohérente qui permet aux développeurs de créer des expériences de qualité. Si les cinquante premières années d'Apple ont été celles de l'ordinateur personnel puis du smartphone, les prochaines pourraient être celles de l'intelligence embarquée et du spatial computing.
Think Different and make something wonderful.

LE SAVIEZ-VOUS ?

Nom de code

Apple attribue généralement des noms de code à chaque projet, chaque machine en développement. Le projet peut changer de nom à chaque itération du projet.

Rouge

Habituellement les prototypes matériels intègrent des cartes mères de couleur rouge. Le rouge indique qu'il s'agit d'un prototype. Parfois, nous trouvons le nom de code de la carte mère ou la machine en développement et la version du prototype ainsi qu'un numéro de série. Cette signalétique n'est pas systématique. Les versions PVT sont généralement vertes et indiquent que la machine est prête à être produite. On peut aussi trouver des cartes mères bleues DVT.

EVT (Engineering Verification Test)

Prototype servant à vérifier le bon fonctionnement des composants.

DVT (Developer Verification Test)

Prototype servant essentiellement aux développeurs pour vérifier le bon fonctionnement des logiciels. Prototypes les plus courants.

PVT (Production Verification Test)

Version finale de la machine qui doit valider la fabrication et le bon fonctionnement de l'ensemble de la machine.

Apple fait (aussi) de l'open source



François Tonic

Nous critiquons facilement Apple : c'est tout fermé, c'est propriétaire, etc. La Pomme fait aussi de l'open source mais communique très peu dessus. Une partie de macOS est en open source. Historiquement, les fondations de MacOS X sont des briques ouvertes. D'autres projets sont disponibles sur GitHub.

Comme nous l'avons dit, depuis les 1eres version de Mac OS X, Apple a rendu disponible la fondation de l'OS sous le nom de Darwin (DarwinOS). Ce nom sera en vigueur jusqu'en janvier 2023. Ce sont des dizaines de librairies, d'API, et de couches bas niveau qui sont sur le GitHub officiel. Cela concerne macOS et iOS.

Les principaux projets open source

1. Swift : le langage de programmation est en open source. Apple met aussi en open source des outils spécifiques, par exemple, Swift Build (février 2025). Voir les articles dédiés
2. Container : Apple met en open source certaines technologies récentes de macOS. Container est une couche légère pour créer, gérer et exécuter des conteneurs nativement sur l'OS. Il est écrit en Swift et optimisé pour les processeur Mx. Container s'appuie sur Containerization, un package Swift pour exécuter des conteneurs Linux sur macOS. Lui-même s'appuie sur le framework Virtualization, qui lui n'est pas open source
3. WebKit : le moteur de rendu est au cœur de Safari et des navigateurs s'appuyant dessus. Il est utilisé par Mail, Safari et de multiples apps. Initialement, Chromium s'appuyait dessus avant que Google n'en fasse un fork
4. *Developer Tools* : Apple ouvre une série d'outils de développement.

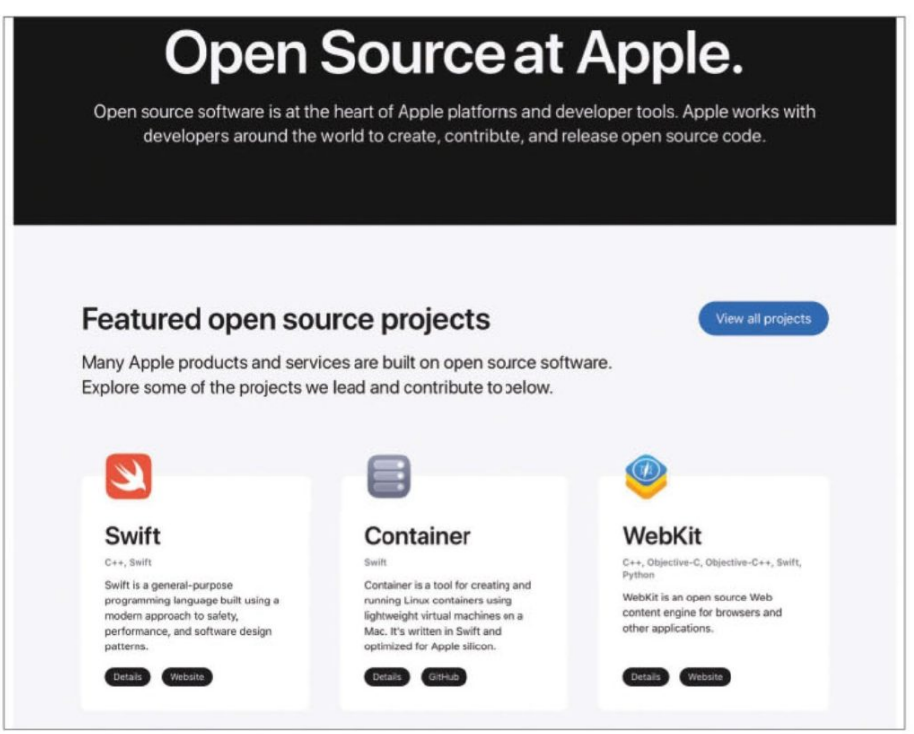
Les autres projets open source importants :

- MLX : framework pour le machine learning
- FoundationDB : base clé valeur
- ResearchKit : framework pour les recherches médicales et la santé en général
- Core ML Tools : paquets Python pour intégrer les Core ML

Le GitHub Apple OSS distributions répertorie tous les projets, toutes les contributions des équipes Apple à l'Open Source : plus de 500 référentiels sont disponibles. Il y a les projets actifs et les projets arrêtés.

Sans forcément le dire, Apple contribue à divers projets : Kubernetes, Apache Cassandra, LLVM & CLang, Apache Spark, Open Policy Agent, Apache Flink, Jupyter.

Sur la partie IA, LLM, Apple n'hésite pas à publier des contributions techniques sur des usages, des techniques de traitements autour de la vision, sur l'halucination des modèles, etc. Le constructeur peu même publier des LLM.



FOCUS SUR CONTAINER

macOS 26 propose un outil natif au système : container. Il permet de créer et d'exécuter des conteneurs Linux directement sur un Mac sous macOS 26 et basé sur un processeur Apple Silicon. Il est écrit en Swift. Il est compatible avec les images OCI et toutes les images standards.

L'outil s'appuie sur le package Containterization qui gère les conteneurs et le framework de virtualisation de macOS. Tout se passe dans le Terminal. Il suffit de démarrer le serveur : container system start. Il vérifie que tout est bien installé. On dispose des commandes de base : création, suppression, run, start, build une image, etc.

Pour l'utiliser, il faut :

- installation de l'outil
- accès un référentiel de conteneurs

Le conteneur n'est pas exécuté directement, mais dans une machine virtuelle légère créée pour chaque conteneur. Containerization isole ainsi chaque conteneur et la VM ne monte que les données nécessaires au conteneur. Il est possible de faire fonctionner l'outil container sous macOS 15, avec des limitations.

Pour lancer un conteneur d'un serveur, on peut taper la commande container run -d- —name (nom image) :latest. S'il s'agit d'une image Docker, Container charge l'image depuis le Docker Hub. Une des promesses est la performance grâce à l'intégration native de container au système.

Un exemple réussi de package proposé par Apple en open source

Swift Async Algorithms apporte à Async Sequence l'élégance que la bibliothèque standard Swift offre aux Sequence. Les flux asynchrones gagnent de nouveaux opérateurs familiers dans le monde synchrone (chain, merge, zip, debounce, etc.), tout en conservant la suspension du temps. Cette symétrie entre synchrone et asynchrone simplifie la compréhension, facilite les conversions entre collections et flux, et offre un modèle clair pour écrire ses propres opérateurs. *Etienne Vautherin*

Site : https://github.com/apple/swift-async-algorithms/

Une base, plusieurs plateformes : partager le code, adapter l'expérience



Etienne Vautherin

Précédemment :

- *"Technology Evangelist" chez Apple,*
- *"iOS Evangelist" chez Orange,*

Actuellement :

- *CTO pour OPlife*

Les plateformes Apple donnent l'impression d'un univers très fragmenté : iPhone, Mac, Apple Watch, Apple TV ou visionOS. Elles reposent sur des interactions et des usages très différents. Pourtant, pour les développeurs, ils partagent de plus en plus une architecture commune. SwiftUI, Swift Package Manager, Multiplatform, Catalyst ou Swift Playgrounds dessinent progressivement un socle unifié qui permet de partager le cœur d'une application tout en respectant les spécificités de chaque plateforme.

Une première constante : la distribution

Malgré cette diversité des plateformes, la distribution des applications repose sur un principe commun. Toutes les applications destinées aux utilisateurs finaux passent par l'App Store et par l'infrastructure de distribution associée. La signature cryptographique, la gestion des certificats, les provisioning profiles et la validation des builds constitue un socle technique partagé.

Ce système implique également une responsabilité juridique claire. Le titulaire du compte développeur reste responsable du logiciel publié, de la conformité aux règles de la plateforme et de la gestion des données personnelles. Même dans un contexte où les outils de génération de code deviennent plus puissants, cette responsabilité reste humaine et ne peut pas être déléguée à un outil automatisé.

Dans la pratique, cette constante influence profondément l'architecture des pipelines de développement. Les systèmes d'intégration continue, les scripts de build et les outils de test s'articulent tous autour de cette étape finale : produire une archive signée et valide pour la distribution.

Partager du code vs partager une cible

Avant d'aborder la cible Multiplatform proposée par Xcode, il est utile de distinguer deux notions souvent confondues dans les projets multi-plates-formes : le partage de code et le partage de cible. Ces deux approches poursuivent des objectifs différents et ne répondent pas aux mêmes contraintes.

Le partage de code consiste à mutualiser les éléments indépendants de la plateforme. Il s'agit généralement de la logique métier, des modèles de données, des services réseau ou des algorithmes. Dans l'écosystème Apple moderne, cette mutualisation est souvent réalisée à l'aide de Swift Package Manager. Un module commun peut alors être consommé par plusieurs applications — iOS, macOS, tvOS ou watchOS — sans duplication de code.

Le partage de cible correspond à une approche différente. Une cible Multiplatform dans Xcode représente une application unique capable de fonctionner sur plusieurs destinations. Dans ce modèle, une partie de la configuration du projet, des ressources et parfois des vues SwiftUI est mutualisée directement au niveau de la cible. Le projet décrit alors un produit unique qui peut être construit pour plusieurs plateformes.

Comprendre la différence entre ces deux stratégies est essentiel. Le partage de code reste toujours possible et souvent souhaitable, même lorsque les cibles sont distinctes. À l'inverse, le partage de cible n'est pertinent que lorsque les plateformes concernées possèdent des comportements suffisamment proches pour qu'une partie de l'application puisse réellement être commune. Cette distinction explique pourquoi certaines plateformes Apple peuvent être intégrées dans une cible Multiplatform tandis que d'autres, comme watchOS, restent généralement isolées dans des cibles spécifiques.

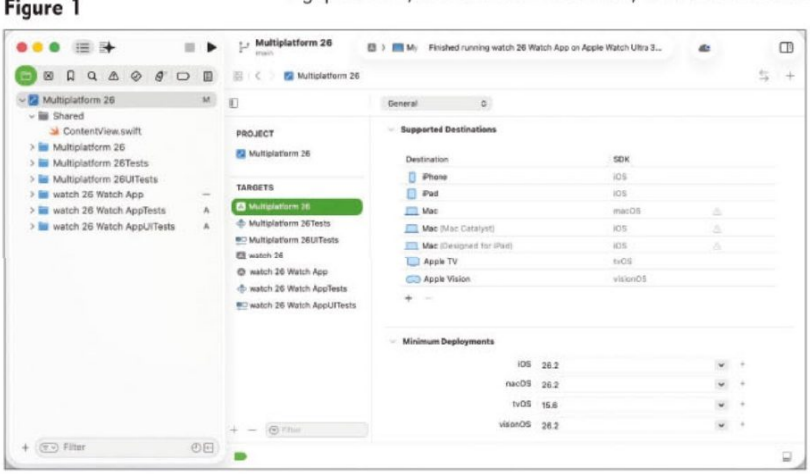
La cible Multiplatform dans Xcode

La cible Multiplatform introduite dans Xcode vise à simplifier le développement d'applications capables de fonctionner sur plusieurs plateformes Apple tout en conservant une structure de projet unique. Dans ce modèle, une application est décrite comme un produit unique auquel sont associées plusieurs destinations possibles : iOS, iPadOS, macOS, tvOS ou visionOS (**figure 1**). L'objectif est de réduire la duplication de code et de configuration lorsque les plateformes partagent des comportements similaires.

Cette approche fonctionne particulièrement bien avec SwiftUI. Le cycle de vie des applications SwiftUI repose sur une structure déclarative commune, ce qui permet souvent de partager une partie importante de l'interface. Une même hiérarchie de vues peut être compilée pour plusieurs plateformes, tandis que certaines différences d'interaction sont gérées par le système ou introduites explicitement dans le code.

La cible Multiplatform mutualise plusieurs éléments du projet : la structure de l'application, certaines ressources, et une partie des paramètres de build. Dans de nombreux cas, une seule cible suffit à produire des versions iOS, macOS ou tvOS d'une application. Le développeur peut ensuite introduire des

Figure 1



spécialisations lorsque c'est nécessaire à l'aide de conditions de compilation ou de vues spécifiques.
Cependant, cette mutualisation ne signifie pas que toutes les plateformes se comportent de manière identique. Les conventions d'interface, les capacités système et les attentes ergonomiques restent différentes. La cible Multiplatform doit donc être vue comme un outil de convergence technique plutôt que comme un mécanisme d'uniformisation des expériences utilisateur.

SwiftUI face aux différences d'interaction

Si SwiftUI constitue aujourd'hui un langage d'interface commun à toutes les plateformes Apple, les modes d'interaction restent profondément différents. Chaque appareil possède son propre paradigme : interaction tactile sur iPhone et iPad (**figure 2**), clavier et souris sur macOS (**figure 3**), navigation par focus sur tvOS, interaction spatiale sur visionOS et interactions très brèves sur Apple Watch (**figure 4**). Une application réellement multi-plate-forme doit donc s'adapter à ces contextes tout en conservant une architecture cohérente.
L'intérêt de SwiftUI est de permettre de décrire l'intention de l'interface plutôt que sa mise en œuvre précise. Le développeur exprime une hiérarchie de vues et des comportements déclaratifs, tandis que le système adapte certains détails à la plateforme cible. Par exemple, une liste d'éléments peut être manipulée par tap sur iOS, parcourue au clavier sur macOS ou naviguer via le moteur de focus sur tvOS.
Cette abstraction ne supprime pas les différences d'expérience utilisateur. Dans de nombreux cas, il reste nécessaire d'introduire des adaptations spécifiques. SwiftUI permet alors d'utiliser des conditions de compilation ou des variantes de vues pour spécialiser le comportement selon la plateforme. Les interactions tactiles, la gestion du focus ou la navigation spatiale peuvent ainsi être exprimées de manière explicite lorsque cela améliore l'expérience utilisateur.
Le rôle de SwiftUI n'est donc pas d'unifier les interfaces entre les plateformes, mais de fournir un langage commun permettant de partager une grande partie de la structure de l'application tout en laissant chaque appareil exprimer ses propres conventions d'interaction.
Un exemple minimal illustre cette convergence.

```
import SwiftUI

@main
struct DemoApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
    }
  }
}

struct ContentView: View {
  var body: some View {
    VStack(spacing: 12) {
      Text("Hello Apple Platforms")
        .font(.title)
      Text(platformDescription)
    }
    .padding()
  }

  var platformDescription: String {
    #if os(iOS)
    return "Running on iOS or iPadOS"
    #elseif os(macOS)
    return "Running on macOS"
    #elseif os(watchOS)
    return "Running on watchOS"
    #elseif os(tvOS)
    return "Running on tvOS"
    #elseif os(visionOS)
    return "Running on visionOS"
    #else
    return "Unknown platform"
    #endif
  }
}
```

Figure 2

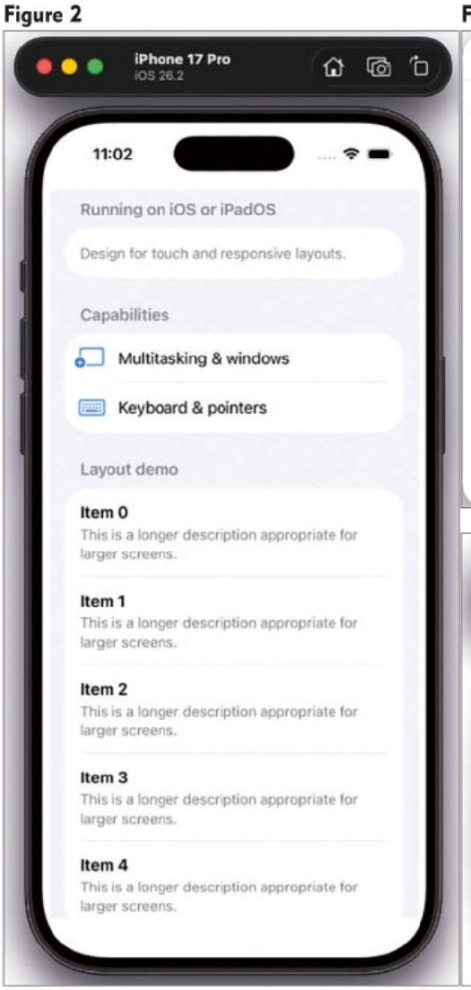


Figure 3

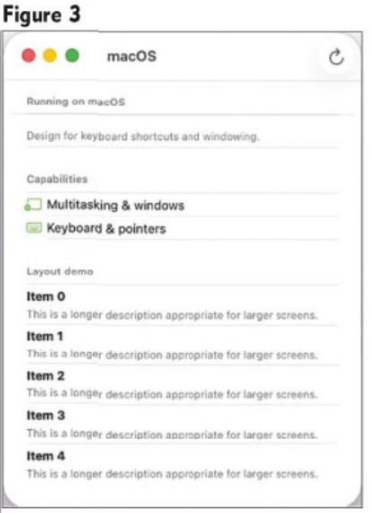




Figure 4

La singularité de watchOS

watchOS occupe une place particulière dans l'écosystème Apple. La plateforme partage pourtant plusieurs technologies avec les autres systèmes : Swift, SwiftUI et une grande partie des frameworks système. Malgré cela, watchOS n'apparaît pas comme une destination possible dans une cible Multiplatform de Xcode. Cette situation s'explique par la

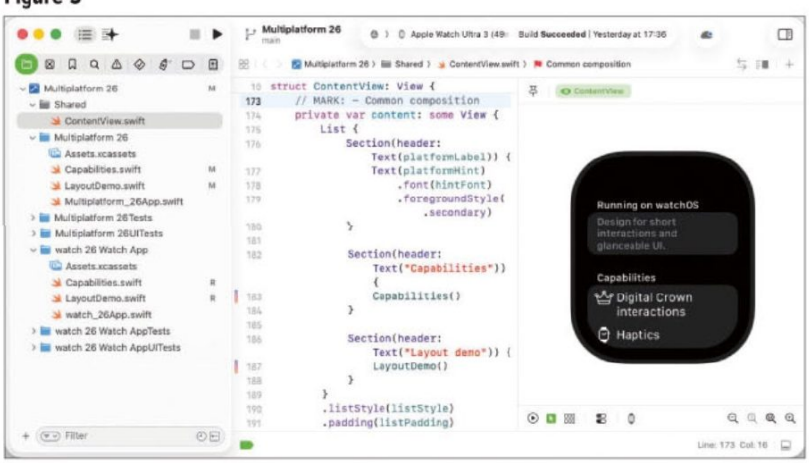
nature même des applications Apple Watch.

Contrairement à iOS ou macOS, watchOS est conçu pour des interactions très courtes et très contextualisées. L'utilisateur consulte une information rapide, lance une action simple ou interagit avec un capteur. Les contraintes de ressources, d'autonomie et de latence sont également plus fortes. Ces caractéristiques influencent fortement la conception des interfaces et le cycle de vie des applications.

Une seconde raison tient à l'architecture historique des projets watchOS. Les premières applications Apple Watch étaient associées à une application iPhone via une extension. Même si watchOS supporte désormais des applications autonomes, cette organisation continue d'influencer la structure des projets et le fonctionnement des outils de build dans Xcode.

Dans la pratique, la stratégie recommandée consiste à partager le code plutôt que la cible. La logique métier, les modèles de données et certains composants SwiftUI peuvent être placés dans un module commun, généralement sous forme de Swift Package ou dans un groupe partagé. (C'est l'exemple de ContentView dans le groupe Shared de la **figure 5**). L 'application watchOS consomme ensuite ce module partagé tout en conservant une cible dédiée et des réglages adaptés à la montre. Cette approche permet de conserver une architecture cohérente sans masquer les différences réelles entre les plateformes.

Figure 5



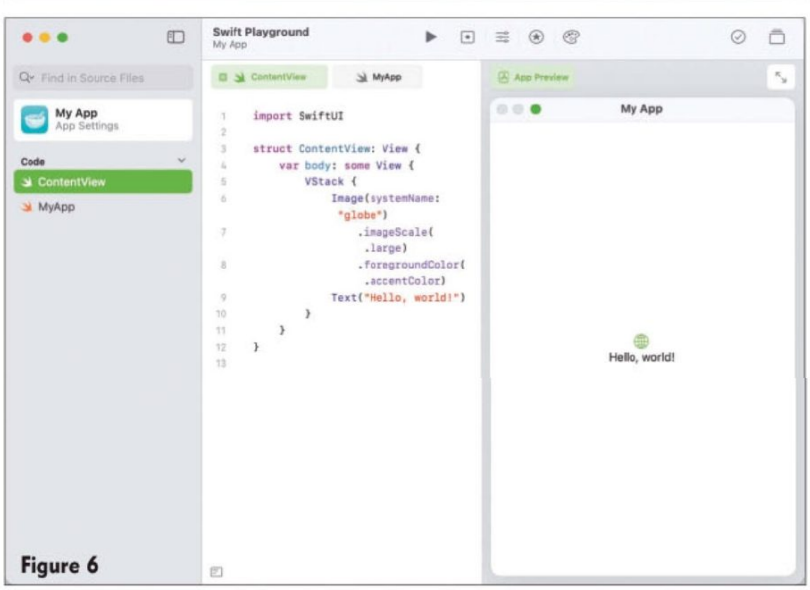


Figure 6

macOS : plusieurs chemins depuis iOS et iPadOS

La convergence entre les plateformes Apple apparaît particulièrement visible lorsqu'une application conçue pour iPhone ou iPad doit fonctionner sur macOS. Contrairement à ce que l'on pourrait imaginer, il n'existe pas une seule solution, mais plusieurs stratégies possibles, chacune correspondant à un niveau différent d'intégration dans l'environnement Mac.

La première possibilité consiste à autoriser l'exécution directe d'une application iOS ou iPadOS sur macOS. Depuis l'arrivée des Mac Apple silicon, le système peut exécuter le même binaire que sur iPhone ou iPad. Cette solution permet d'obtenir rapidement une présence sur Mac avec très peu d'adaptations. En revanche, l'expérience utilisateur reste souvent proche de celle d'un appareil tactile et peut paraître moins naturelle avec un clavier, une souris ou un trackpad.

Une seconde approche est proposée par Mac Catalyst. Cette technologie permet de recompiler une application iPad afin de produire une véritable application macOS. Le code UIKit existant peut être largement réutilisé, tandis que l'interface peut être adaptée pour intégrer des menus, des raccourcis clavier ou une gestion plus riche des fenêtres. Catalyst constitue ainsi une étape intermédiaire entre une application iPad et une application Mac pleinement native.

Enfin, il est également possible de créer une application macOS spécifique tout en partageant la logique métier avec les autres plateformes. Dans ce modèle, la couche métier est placée dans un module commun — souvent un Swift Package — tandis que l'application macOS possède sa propre cible et ses propres conventions d'interface. SwiftUI facilite cette approche en permettant de partager certaines vues tout en spécialisant celles qui doivent respecter les usages du Mac.

Ces trois modes pour la destination Mac sont visibles sur la **figure 1**. L'avertissement sur ces lignes indique que, en dehors de cette démonstration, il faut préciser un mode unique pour la destination Mac.

Swift Playgrounds dans l'écosystème de développement Apple

Swift Playgrounds est souvent perçu comme un outil de mise au point rapide ou un outil pédagogique destiné à apprendre le langage Swift. Pourtant, son évolution récente en fait également une pièce intéressante de l'architecture de développement de l'écosystème Apple (**figure 6**). L'application s'appuie en réalité sur les mêmes briques fondamentales que les projets créés avec Xcode : Swift, SwiftUI et Swift Package Manager. Les projets générés dans Playgrounds reposent sur une structure de package Swift qui peut être comprise et manipulée par les autres outils de l'écosystème.

Sur iPad, Swift Playgrounds permet aujourd'hui de concevoir une application SwiftUI complète, de l'exécuter localement et même de la publier sur l'App Store. L'environnement propose un éditeur de code, une gestion de projet simplifiée et un système de prévisualisation permettant de tester rapidement les interfaces. L'iPad devient ainsi une machine de développement légère, particulièrement adaptée aux prototypes, aux projets pédagogiques ou à certaines applications relativement simples.

La version macOS de Swift Playgrounds joue un rôle complémentaire. Elle permet d'explorer des packages Swift, de tester rapidement des idées et de manipuler du code SwiftUI dans un environnement plus léger que Xcode. Même si elle ne remplace pas un IDE complet pour les projets complexes, elle illustre la cohérence de l'écosystème : les mêmes formats de projet et les mêmes technologies peuvent circuler entre iPad, Mac et Xcode.

Swift Playgrounds révèle ainsi une idée importante : dans l'écosystème Apple, la diversité des outils ne remet pas en cause l'unité de l'architecture logicielle. Les interfaces de développement peuvent varier, mais elles reposent sur les mêmes fondations techniques.

L'impact des agents d'intelligence artificielle dans l'environnement de développement

L'arrivée d'agents d'intelligence artificielle capables de comprendre et de produire du code modifie progressivement la manière dont les développeurs interagissent avec les outils de programmation. Dans l'écosystème Apple, cette évolution s'inscrit naturellement dans les environnements existants comme Xcode, qui deviennent progressivement des espaces de collaboration entre le développeur et des assistants capables d'analyser un projet logiciel.

Ces agents peuvent intervenir à plusieurs niveaux du cycle de développement. Ils peuvent générer des fragments de code, proposer des implémentations pour une API donnée, produire des tests unitaires ou analyser une base de code afin d'identifier des duplications ou des opportunités de refactorisation. L'objectif n'est pas de remplacer le développeur, mais d'accélérer certaines tâches répétitives et d'explorer plus rapidement différentes solutions techniques.

Dans un projet multi-plate-forme, ces outils deviennent particulièrement intéressants. Un agent peut par exemple suggérer l'adaptation d'une vue SwiftUI pour une plateforme spécifique, générer une implémentation alternative pour macOS ou proposer des spécialisations conditionnelles pour tvOS ou visionOS. Le développeur conserve la responsabilité de l'architecture globale, mais bénéficie d'un assistant capable d'explorer rapidement l'espace des implémentations possibles.

Cette évolution renforce encore l'importance d'une architecture claire et modulaire. Plus le code est structuré autour de modules cohérents — logique métier, services, interface utilisateur — plus les outils d'analyse et de génération peuvent intervenir efficacement. Les agents d'IA ne remplacent donc pas les principes d'ingénierie logicielle ; ils rendent au contraire ces principes encore plus précieux.

Les outils de génération de code spécialisés modifie profondément la manière dont les projets logiciels sont amorcés. Des plateformes comme Rork (**figure 7**) illustrent cette évolution : au lieu de commencer par une feuille blanche, le développeur peut générer en quelques minutes la structure initiale d'une application. Modèles de données, premières vues SwiftUI, navigation de base et organisation du projet peuvent être produits automatiquement à partir d'une description textuelle.

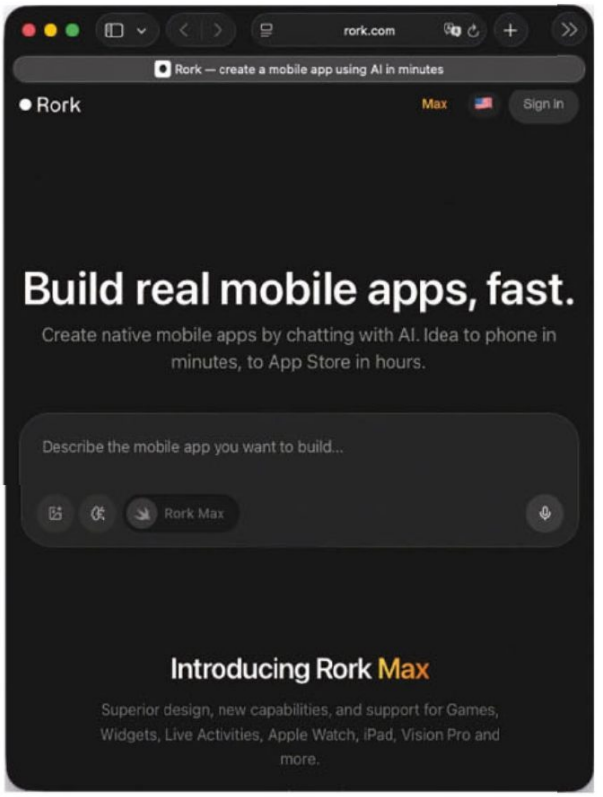


Figure 7

Unité architecturale et diversité des expériences

Le développement multi-plate-forme dans l'écosystème Apple repose sur un équilibre particulier. Les appareils et les contextes d'usage sont réellement différents : interaction tactile sur iPhone et iPad, clavier et souris sur Mac, navigation par focus sur Apple TV, interactions rapides sur Apple Watch ou interfaces spatiales sur visionOS. Concevoir une application pour ces environnements implique donc d'accepter une diversité d'expériences utilisateur.

Dans le même temps, l'écosystème repose sur des fondations techniques remarquablement cohérentes. Swift constitue le langage commun, SwiftUI propose un modèle déclaratif d'interface partagée et Swift Package Manager permet d'organiser le code en modules réutilisables. Les mécanismes de distribution, de signature et de publication des applications suivent également des principes similaires d'une plateforme à l'autre. La cible Multiplatform, les technologies comme Mac Catalyst ou la possibilité d'exécuter certaines applications iOS sur macOS illustrent différentes formes de convergence. Ces mécanismes ne cherchent pas à effacer les différences entre les appareils, mais à faciliter le partage des fondations logicielles.

Pour les développeurs, la stratégie la plus robuste consiste généralement à stabiliser le cœur applicatif dans des modules partagés, puis à construire autour de ce noyau des applications adaptées à chaque plateforme. Cette approche permet de conserver une architecture cohérente tout en respectant les conventions propres à chaque environnement.

Pour finir, l'écosystème Apple ne repose pas sur l'idée d'une interface identique partout. Il repose plutôt sur une architecture logicielle commune capable de produire des expériences différentes selon les appareils. C'est cette combinaison d'unité technique et de diversité d'interaction qui rend possible le développement multi-plate-forme sur les plateformes Apple.

Retrouvez plus de contenu, et des liens sur ce sujet, sur le site https://murmurations.dev/fr/



Florent Morin

CTO indépendant. J'ai accompagné plusieurs projets dans l'univers Apple. Aujourd'hui, je suis CTO de ClearSurgery à plein temps. Je suis aujourd'hui spécialiste de la plateforme Apple Vision en particulier.

TÉMOIGNAGE DE FLORENT MORIN :

oui, j'adore Xcode

Quand es-tu arrivé dans l'univers Apple ?

Étant passionné des technologies open source, j'ai commencé ma carrière en tant que développeur système Linux. Puis j'ai découvert en 2008 le SDK iPhone : une véritable opportunité pour les développeurs indépendants, en particulier grâce à l'App Store.

J'ai - pendant des années - conçu de nombreuses apps iOS et iPadOS. J'ai également contribué à des projets sur watchOS et tvOS. Puis, assez tardivement et en particulier grâce à l'uniformisation des plateformes, je suis arrivé sur macOS. Enfin, en 2023, j'ai choisi de me spécialiser sur visionOS.

On présente parfois l'écosystème développeur comme rigide et peu ouvert (par exemple : obligation d'utiliser Xcode), quel est ton avis, ton experience avec ?

Avis peut-être assez peu populaire : j'adore Xcode. En particulier depuis l'intégration native des agents tels que Claude et Codex. L'écosystème développeur peut sembler assez peu ouvert. Cela étant, le noyau des systèmes et une majorité du système d'exploitation sont open source.

Le langage Swift est open source. Core ML est open-Source. MLX est open source. Le moteur de iCloud est également open source. WebKit est open source. Finalement, Apple ne fait pas moins d'open source que les autres. C'est juste une question de communication.

Avec l'expérience et les projets que tu as codé, comment vois-tu l'évolution de la plateforme ?

Le fait que les plateformes s'uniformisent est une véritable opportunité pour les développeurs. Aujourd'hui, porter un projet d'une plateforme à l'autre est en général une formalité. Un énorme atout du langage Swift est également sa portabilité : aujourd'hui, nous pouvons aller au-delà des systèmes Apple.

Les serveurs web en Swift sont extrêmement performants, vraiment en haut du panier. Et le Swift en embarqué (i.e. Raspberry Pi) offre également d'excellentes performances. Cette capacité de traiter plusieurs plateformes avec un seul langage est évidemment un atout pour le développeur.

Mais c'est également une opportunité exceptionnelle de simplifier le travail des IA, une approche définitivement tournée vers le futur.

Swift est LE langage par défaut sur les plateformes Apple. Quel est ton regard sur ce langage et les évolutions, parfois importantes depuis les versions 5.x et 6.x ?

Clairement, aujourd'hui, Swift est devenu un langage extrêmement puissant. Chacune des évolutions du langage a été certes douloureuse, mais totalement justifiée. Avec Swift Concurrency dans Swift 6, les performances sont exceptionnelles. Donc finalement, on ne peut pas vraiment sans plaindre.

Aujourd'hui, le langage en lui-même semble s'être stabilisé et son écosystème s'étoffe progressivement, avec un nombre croissant de paquets open source. On peut probablement dire qu'il est arrivé à maturité.

Xcode reste l'outil par défaut pour les développements natifs. Pour toi, est-ce un bon environnement même s'il apparaît monolithique et peu modulaire ?

Franchement, Xcode est capable de compiler du code vieux de 30 ans tout en intégrant les dernières technologies d'IA. Comme pour chaque outil, cela implique des compromis. Le fait que Xcode avance à son propre rythme me semble totalement acceptable. On peut regretter sa rigidité : j'aurais apprécié de pouvoir compiler des projets en Swift embarqué depuis Xcode. Mais franchement, pour 90 % des tâches, c'est un excellent outil.

Qu'est-ce qui est le plus difficile à maîtriser sur le modèle de développement ? Le langage, les nombreux SDK, la toolchain, les processus liés aux stores ?

À la fois, tout est difficile et tout est simple. Si on veut créer un projet assez simple, c'est assez facile de créer une application en Swift et la publier sur l'App Store. Cependant, et en toute logique, dès que l'on souhaite un résultat plus avancé, les choses se complexifient.

L'avantage est tout de même la change d'avoir un certain niveau de granularité au travers des outils proposés par Xcode et l'écosystème de développement Apple.

1 an de programmez!
Le magazine des dév CTO - Tech Lead
ABONNEMENT PDF : 39 €
Abonnez-vous directement sur www.programmez.com

À Swift ouvert

J'ai écrit ma première ligne de "code" en 1999. J'avais 9 ans et un PC sous Windows 98 (personne n'est parfait), et une curiosité dévorante pour ces machines qui semblaient pouvoir tout faire à condition qu'on leur parle correctement. Vingt-sept ans plus tard, je n'ai jamais arrêté.



Clément Sauvage

Ingénieur de formation, Entrepreneur, Fondateur de BNC Consulting SAS (Bits'n Coffee) — Agence développement iOS, macOS et web basé à Lille.

Liens / contact

https://clementsauvage.me

Voir le site de Clément

Mon entrée dans l'univers Apple date de 2006, un an avant l'iPhone, avec une toute petite app mac, mais surtout une app "mobile" pour mon Palm Zire 72s.

Puis 2007 est arrivé, et avec lui l'iPhone. Puis 2008, et l'App Store. Et là, tout a basculé.

Depuis, j'ai créé plus de quarante applications pour des clients aussi divers que LVMH, Iliad ou Airbus, et des dizaines de startups connues (ou moins connues).

Au-delà du code, j'ai toujours eu besoin de la communauté. J'ai dirigé les *CocoaHeads* Lille pendant six ans, qui vont renaître de leurs cendres. J'organise depuis 2022 le "WWDC Frenchies Dinner" — ce dîner annuel qui réunit les développeurs francophones à Cupertino pendant la conférence Apple —, et j'anime des espaces Slack et WhatsApp qui rassemblent plusieurs centaines de développeurs français. J'ai formé plus de 150 développeurs Swift, donné des conférences à SwiftConf, AltConf, iOS Conf SG. Ce métier ne se vit pas seul.

Aujourd'hui, je partage mon temps entre le consulting pour des startups, grands comptes, le développement de mes propres applications — dont Onyria, et Punchy —, et cette passion intacte pour un écosystème qui, malgré ses défauts, reste le plus stimulant que je connaisse.

Quand es-tu arrivé dans l'univers Apple ?

J'ai découvert le monde du Mac en 2004, lors de mon stage de 3eme à l'Atelier Lyrique de Tourcoing (je voulais alors être chanteur d'Opéra). Mon premier Mac, c'était en 2006. Je cherchais autre chose que le monde Windows et son chaos de DLL manquantes et d'écrans bleus...

Mac OS X m'a séduit immédiatement : l'élégance de l'interface, la puissance d'Unix sous le capot, et cette philosophie qui semblait dire "on a pensé à tout pour que tu puisses te concentrer sur ce qui compte".

J'ai commencé par Objective-C, ce langage étrange avec ses crochets et sa syntaxe venue de NeXT (Steve Jobs <3). Au début, c'était déroutant. Puis j'ai compris : Objective-C n'était pas juste un langage, c'était une façon de penser le code, héritée de Smalltalk, où les objets s'envoient des messages plutôt que d'appeler des méthodes. Une nuance subtile qui change tout.

Et puis 2007 est arrivé. La keynote de janvier. Steve on stage, ce "one more thing" qui n'en était pas un puisque c'était LE sujet : l'iPhone : *An iPod, a Phone and a Breakthrough Internet Communicator*,... et pas de SDK public. Apple voulait qu'on fasse des "web apps".

Un an plus tard, Apple "cédait" et ouvrait l'App Store. J'étais dans la première vague. Pas parce que j'étais visionnaire, mais parce que j'étais là, prêt, avec déjà un an d'Objective-C dans les mains. Parfois, le timing fait tout.

Depuis, je n'ai jamais quitté cet écosystème. Pas par loyauté aveugle — je suis profondément critique de certains choix d'Apple —, mais parce qu'aucune autre plateforme ne m'a offert cette combinaison de cohérence, de qualité, et de connexion directe avec des centaines de millions d'utilisateurs.

On présente parfois l'écosystème développeur comme rigide et peu ouvert. Quel est ton avis ?

Cette critique revient sans cesse, et je la comprends.

Pour développer nativement sur iOS ou macOS :

- T'as besoin d'un Mac.
- Tu dois utiliser Xcode, l'IDE d'Apple, que tu l'aimes ou non.
- Tu dois passer par l'App Store, te soumettre à la review, respecter des guidelines qui évoluent chaque année.

Tu es, d'une certaine façon, captif.

Mais cette captivité a une contrepartie que les critiques oublient souvent : la cohérence.

Quand je développe une app iOS, je sais exactement sur quels appareils elle tournera. Cinq ou six tailles d'écran, des processeurs que je connais par cœur, des versions d'iOS dont je peux tracer l'adoption au pourcentage près. Compare ça à Android, où tu dois gérer des milliers de configurations différentes, des constructeurs qui modifient le système à leur sauce, des utilisateurs bloqués sur des versions vieilles de cinq ans parce que leur fabricant a cessé les mises à jour.

Le "jardin clos" d'Apple, c'est aussi un jardin où les outils sont pensés ensemble. Simulator pour les tests, Instruments pour le profiling, Xcode Cloud pour l'intégration continue, TestFlight pour la distribution bêta. Tout communique, pas toujours parfaitement certes, mais ensemble.

Ce qui me frustre un peu ?

L'App Store Review. Après dix-huit ans, ce processus reste une boîte noire. Tu soumets ton app, tu attends, tu croises les doigts. Parfois tu es accepté en deux heures. Parfois tu es rejeté au bout de trois jours pour une raison que tu ne comprends pas, tu corriges, tu resoumets exactement la même chose, et tu passes. Il y a un côté loterie qui, après 400M téléchargements, me sidère encore parfois.

Mais au final, est-ce que je changerais de plateforme? Non. Parce que cette rigueur produit aussi de la qualité. Les apps iOS sont généralement meilleures que leurs équivalents Android. L'expérience utilisateur est plus soignée. Et ça, c'est en partie grâce à ce cadre contraignant qui pousse les développeurs vers le haut.

Comment vois-tu l'évolution de la plateforme ?

En dix-huit ans, j'ai traversé trois ères distinctes, chacune avec sa philosophie, ses outils, et ses promesses.

L'âge de pierre (2008-2014)

Les premières années de l'App Store ressemblaient au Far West. Pas de règles établies, pas de patterns clairs, et cette excitation (de jeunesse je dirais) de défricher un territoire vierge. On inventait nos architectures, on se trompait souvent et on recommençait.

C'était l'époque où tu pouvais lancer une app lampe de poche (un jeu de morpion dans mon cas) et te retrouver en tête des charts. Où une bonne idée, bien exécutée, suffisait à percer. Les barrières à l'entrée étaient faibles, la concurrence encore gérable, et Apple nous laissait une liberté immense.

Chaque WWDC apportait des briques qu'on empilait : Core Data, Core Animation, Core Bluetooth, Core Location... On avait le sentiment de construire quelque chose de neuf, ensemble.

L'âge du fer (2014-2020)

En juin 2014, Apple a fait un pari audacieux : annoncer Swift, un nouveau langage de programmation, et demander à toute sa communauté de développeurs d'abandonner progressivement Objective-C, le langage qu'ils utilisaient depuis **trente ans.**

Les premières versions de Swift étaient chaotiques. Chaque mise à jour cassait la compatibilité. Tu passais des jours à migrer du code qui fonctionnait parfaitement, juste pour qu'il compile avec la nouvelle version. C'était parfois... ragecnt.

Mais Apple avait raison sur le fond. Objective-C, malgré ses qualités, traînait un héritage lourd. Swift apportait la sécurité du typage, les optionnels qui éradiquent les *null pointer exceptions*, une syntaxe moderne, et une expressivité qui rendait le code plus lisible.

À partir de Swift 5, la stabilité ABI est arrivée. Enfin, tu pouvais distribuer des frameworks sans te soucier de la version exacte du compilateur. Le langage était mûr.

Gist Github

Figure 1

```objc
- (void)loadUserProfileWithCompletion:(void (^)(User *user, NSError *error))completion {
    NSURL *url = [NSURL URLWithString:@"https://api.example.com/user/profile"];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    [[[NSURLSession sharedSession] dataTaskWithRequest:request
        completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {

        if (error) {
            dispatch_async(dispatch_get_main_queue(), ^{
                completion(nil, error);
            });
            return;
        }

        NSError *jsonError;
        NSDictionary *json = [NSJSONSerialization JSONObjectWithData:data
                                                             options:0
                                                               error:&jsonError];

        User *user = [[User alloc] initWithDictionary:json];

        // Maintenant, récupérer les posts de cet utilisateur...
        NSURL *postsURL = [NSURL URLWithString:
            [NSString stringWithFormat:@"https://api.example.com/user/%@/posts", user.userId]];
        NSURLRequest *postsRequest = [NSURLRequest requestWithURL:postsURL];

        [[[NSURLSession sharedSession] dataTaskWithRequest:postsRequest
            completionHandler:^(NSData *postsData, NSURLResponse *postsResponse, NSError *postsError)
{
            if (postsError) {
                dispatch_async(dispatch_get_main_queue(), ^{
                    completion(nil, postsError);
                });
                return;
            }

            NSArray *postsJSON = [NSJSONSerialization JSONObjectWithData:postsData
                                                                 options:0
                                                                   error:nil];
            user.posts = [Post postsFromArray:postsJSON];

            // Retour sur le main thread pour mettre à jour l'UI
            dispatch_async(dispatch_get_main_queue(), ^{
                completion(user, nil);
            });

        }] resume];  // N'oubliez pas le resume sinon rien ne se passe !

    }] resume];
}
```

L'âge du cuivre ou la convergence (2020-aujourd'hui)

Aujourd'hui, Apple unifie tout. Un seul langage dominant (Swift), un framework UI déclaratif (SwiftUI), une architecture matérielle (Apple Silicon). La promesse : écrire une fois, déployer partout — iPhone, iPad, Mac, Apple Watch, Apple TV, Vision Pro *(comme Java et la JVM quoi, hein, quoi, la porte? OK je m'en vais)*.

SwiftUI, annoncé en 2019, incarne cette vision. Fini le code impératif où tu manipules des vues à la main. Tu déclares ce que tu veux, et le framework se débrouille. C'est élégant, puissant, et parfois terriblement frustrant quand ça ne marche pas comme prévu. Car SwiftUI n'est pas encore au niveau d'UIKit pour tout mais on y arrive après 8 ans. Les cas limites, les personnalisations poussées, les performances sur des éléments complexes — il reste du chemin. Mais la direction est claire. Dans quelques années, UIKit sera le legacy qu'on maintient, mais qu'on n'écrit plus.

Swift : ton regard sur le langage et ses évolutions ?

Swift est, à mes yeux, l'un des meilleurs langages de programmation jamais créés (OK je suis biaisé). Ce n'est pas du chauvinisme de développeur Apple — j'ai pratiqué suffisamment d'autres langages pour comparer. Swift combine la sécurité de Rust, l'expressivité de Python, et une modernité qui fait paraître Java antédiluvien. Les optionnels, par exemple. En Swift, une variable peut contenir une valeur ou être *nil* (vide), et le compilateur t'oblige à gérer explicitement les deux cas. Fini les crashs à 3h du matin parce qu'un pointeur null s'est glissé quelque part. Cette seule feature justifierait presque l'existence du langage.

Les origines : d'Objective-C à la nécessité d'un nouveau langage

Pour comprendre Swift, il faut d'abord comprendre ce qu'il remplace. Objective-C est né en 1983, créé par Brad Cox et Tom Love chez Stepstone. Brad, fasciné par Smalltalk, mais contraint de travailler en environnement Unix/C, a développé un "Object-Oriented Pre-Compiler" (OOPC) qui traduisait une syntaxe inspirée de Smalltalk en C pur. L'idée de Cox était visionnaire : créer des composants logiciels réutilisables qu'on pourrait acheter sur étagère, comme des circuits intégrés électroniques.

En 1988, Steve Jobs, fraîchement évincé d'Apple et à la tête de NeXT, licencie Objective-C pour son système d'exploitation NeXTSTEP. C'est Steve Naroff, ingénieur chez Stepstone puis NeXT, qui transforme le langage brut de Cox en quelque chose de réellement utilisable pour le développement d'applications graphiques. NeXT finit par racheter tous les droits sur le langage.

Quand Apple rachète NeXT en 1996, Objective-C devient le langage de Mac OS X (2001), puis de l'iPhone (2007). Pendant près de 30 ans, ce langage des années 80, avec sa syntaxe à crochets héritée de Smalltalk et ses header files venus du C, a été le seul moyen de développer nativement pour Apple. Exemple pour récupérer un profil user et ses posts. **Figure 1**

La genèse secrète de Swift (2010-2014)

En juillet 2010, **Chris Lattner** — déjà créateur de LLVM et Clang — commence à travailler sur Swift pendant ses nuits et week-ends. Lattner, embauché par Apple en 2005 pour remplacer GCC par LLVM (malgré le scepticisme interne), avait une vision claire : Apple avait besoin d'un langage moderne.

Pendant un an et demi, Lattner développe Swift en secret tout en dirigeant une équipe de plus de 40 personnes le jour. Quand il pré-

sente une première version au leadership d'Apple, la réaction est tiède : *"Why would we want a new language?"* Lattner persévère, documente, argumente.

Swift emprunte à de nombreux langages : Objective-C pour l'interopérabilité, Rust pour la sécurité mémoire, Haskell pour les fonctionnalités fonctionnelles, Python pour l'expressivité, C# pour certains patterns. Lattner lui-même reconnaît que Swift a pris des idées de "far too many [languages] to list."

Le 2 juin 2014, à la WWDC, Swift est annoncé comme "Objective-C without the C." L'application WWDC (Maintenant "Developer") elle-même, distribuée ce jour-là, est la première app publique écrite en Swift. Apple publie simultanément un manuel de 500 pages, gratuit sur l'iBooks Store. **Figure 2**

Les années chaotiques (2014-2018)

Les premières versions de Swift sont instables. Swift 1.0 sorten septembre 2014 avec Xcode 6. Swift 1.1 arrive un mois plus tard. Swift 1.2 en avril 2015. Chaque version apporte des changements syntaxiques qui cassent le code existant.

Swift 2.0, annoncé à la WWDC 2015, introduit la gestion d'erreurs avec try/catch et le protocole-oriented programming. Surtout, le 3 décembre 2015, Apple rend Swift **open source** sous licence Apache 2.0 — une décision rare à Cupertino.

Swift 3.0 (septembre 2016) est le plus traumatisant. Apple refonde complètement la syntaxe : nommage des paramètres, conventions de casse, APIs redessinées. La migration d'un projet Swift 2 vers Swift 3 peut prendre des jours (on avait pas d'IA à l'époque). Mais c'est aussi le début de la stabilisation : l'équipe Swift annonce que la stabilité du code source devient une priorité.

En janvier 2017, Chris Lattner quitte Apple pour Tesla, puis Google Brain. Ted Kremenek prend la direction du projet Swift.

Swift 5 : la maturité (mars 2019)

Swift 5, sorti le 25 mars 2019 avec Xcode 10.2, marque un tournant historique : **la stabilité ABI** (Application Binary Interface).

Qu'est-ce que ça signifie concrètement? Avant Swift 5, chaque application devait embarquer le runtime Swift dans son bundle — plusieurs mégaoctets supplémentaires. Pire : une app compilée avec Swift 4.1 ne pouvait pas utiliser une bibliothèque compilée avec Swift 4.2. L'ABI stability garantit qu'une app compilée avec Swift 5.0 fonctionnera avec le runtime Swift intégré à iOS 12.2+, et sera compatible avec les futures versions.

Ted Kremenek, à l'époque responsable des langages chez Apple, déclare : *"Swift 5 is a major milestone in the evolution of the language. Thanks to ABI stability, the Swift runtime is now incuded in current and future versions of Apple's platform operating systems."*

Les conséquences pratiques :

- Apps plus légères (pas de runtime embarqué)
- Distribution de frameworks binaires possible
- Fin des migrations obligatoires à chaque version de Swif

Swift 5.5 : la révolution concurrence (WWDC 202)

Swift 5.5, annoncé à la Dubdub 2021, est probablement la plus grande évolution du langage depuis sa création. Apple introduit **async/await** et les **actors** — une refonte complète de la gestion de la concurrence.

Chris Lattner avait publié un *"Concurrency Manifesto"* dès 2017, posant les bases théoriques de ce qui allait devenir Swift Concurrency. Quatre ans plus tard, c'est livré.



Figure 2 :
Chris Lattner et moi (2016)

```swift
// Swift 5.5+ (2021) : la même fonctionnalité, lisible comme du code séquentiel

func loadUserProfile() async throws -> User {
    let profileURL = URL(string: "https://api.example.com/user/profile")!
    let (profileData, _) = try await URLSession.shared.data(from: profileURL)
    var user = try JSONDecoder().decode(User.self, from: profileData)

    let postsURL = URL(string: "https://api.example.com/user/\
(user.id)/posts")!
    let (postsData, _) = try await URLSession.shared.data(from: postsURL)
    user.posts = try JSONDecoder().decode([Post].self, from: postsData)

    return user
}

// Appel :
let user = try await loadUserProfile()
```

Figure 3 :
La simplicité de Swift vs Objective -C.

Avant async/await, la concurrence en Swift reposait sur Grand Central Dispatch (GCD), une API C d'Apple. Le code asynchrone nécessitait des closures imbriquées — le fameux "callback hell" : **Figure 3**

Swift 6 : la rigueur au prix de la douleur (WWDC 2024)

Swift 6, annoncé à la WWDC 2024 et officiellement disponible le 17 septembre 2024 avec Xcode 16, active par défaut le **strict concurrency checking** — les vérifications strictes de concurrence.

Holly Borla, membre du Swift Core Team, l'annonce ainsi : *"Swift 6 now includes a new, opt-in language mode that extends Swift's safety guarantees to prevent data races in concurrent code by diagnosing potential data races in your code as compiler errors."*

L'objectif est noble : éliminer les data races — ces bugs vicieux où deux threads accèdent simultanément aux mêmes données.

Avant Swift 6, ces erreurs ne se manifestaient qu'à l'exécution, souvent de façon non reproductible. Désormais, le compilateur les détecte avant même que l'app ne soit lancée.

Le prix à payer : une migration brutale. Les développeurs doivent annoter leur code avec **@Sendable**, **@MainActor**, comprendre les concepts d'*isolation domains*, gérer les cas où des types existants ne sont pas *thread-safe*. J'ai personnellement passé des jours à migrer 2 projets existants.

Swift 6.2, annoncé en septembre 2025, tente d'adoucir la pilule avec l'*"Approachable Concurrency"* — une option qui isole par défaut tout le code sur le MainActor, simplifiant drastiquement les cas d'usage courants.

Ce que j'en retiens après 19 ans d'environnement Apple (donc plus de la moitié de ma vie)

Swift est, objectivement, un langage remarquable. Il combine :

- La **sécurité mémoire** héritée de langages modernes comme Rust
- L'**expressivité** de langages dynamiques comme Python

- La **performance** d'un langage compilé
- Une **interopérabilité** totale avec l'écosystème Objective-C existant

Mais son évolution illustre aussi les tensions inhérentes à tout langage vivant. Apple a fait des choix courageux — casser la compatibilité avec Swift 3, imposer la rigueur de la concurrence avec Swift 6 — au prix de la productivité immédiate des développeurs.



Iterview de Chris Lattner

Chris Lattner, dans une interview récente, résume bien la philosophie : les systèmes qui durent des décennies — LLVM, Swift — sont ceux dont l'architecture est pensée pour évoluer. *"The reason that those systems were fundamental, scalable, successful, and didn't crumble under their own weight is because the architecture of those systems actually worked well."* Dix ans après son annonce, Swift est devenu ce que Lattner espérait : un langage qui permet à des millions de développeurs de créer des applications pour des milliards d'utilisateurs. Et ce n'est que le début.

Xcode : bon environnement ou monolithe pesant ?

Xcode, c'est le couteau suisse obligatoire du développeur Apple. Il fait tout : édition de code, compilation, debug, profiling, gestion des assets, signature des apps, upload sur l'App Store. C'est impressionnant. C'est aussi son problème.

Ce que j'apprécie :

L'intégration verticale. Tout fonctionne ensemble parce que tout vient du même endroit. Les simulateurs sont excellents, le debugger est puissant, Instruments pour le profiling est un outil de classe mondiale que peu d'autres plateformes peuvent égaler.

Les Previews SwiftUI. Quand ils fonctionnent — et c'est le cas de plus en plus souvent —, c'est magique. Tu modifies ton code, tu vois le résultat instantanément. Le cycle édition-compilation-exécution fond en quelques secondes.

La stabilité globale. Malgré ses bugs légendaires — qui n'a jamais pesté contre un "Indexing" qui tourne à l'infini ? — Xcode reste un IDE robuste pour le développement quotidien.

Ce qui me pèse :

La lourdeur sur les gros projets.

Ouvrir un workspace complexe peut prendre plusieurs minutes. L'autocomplétion rame. Le processus indexe en permanence. Sur une machine Apple Silicon, c'est gérable. Sur un Mac plus ancien, c'est calvaire (à l'heure de l'écriture j'ai un M1 Pro).

Chaque version majeure de Xcode apporte son lot de régressions. Des fonctionnalités qui marchaient parfaitement cassent soudain. **Tu apprends à ne jamais mettre à jour Xcode en plein projet critique**

Quand AppCode, l'IDE de JetBrains pour iOS, a été abandonné en 2023, j'ai eu un pincement. Ce n'était pas parfait, mais c'était une alternative.

Qu'est-ce qui est le plus difficile à maîtriser ?

Si je devais résumer dix-neuf ans de développement Apple en une phrase, ce serait : "Le langage, c'est 20% du job. Tout le reste, c'est l'écosystème."

Le langage Swift

En trois jours de formation intensive, je peux rendre quelqu'un opérationnel en Swift. Les bases sont accessibles, la syntaxe intuitive, les concepts progressifs. Un développeur venant de Java ou Python s'y retrouve vite.

Les SDK

Et puis tu découvres les SDK. Des dizaines de frameworks, chacun avec sa philosophie, ses patterns, ses pièges. UIKit et ses delegates, Core Data et son contexte managé, SwiftUI et son cycle de vie déclaratif. Tu peux passer des années à maîtriser tout ça, et Apple en ajoute chaque WWDC. (Swift Data...).

La toolchain

Les certificats, les profils de provisioning, la signature de code. Après dix-huit ans d'App Store, ce reste le cauchemar numéro un des développeurs. J'ai formé des gens brillants, capables d'écrire des algorithmes complexes, qui se retrouvaient bloqués pendant des heures sur un problème de signing. Apple a tenté de simplifier avec Xcode Cloud et la signature automatique, mais dès que tu sors des cas standards, tu replonges dans l'obscurité. Et la un coup de chapeau à l'ami Félix Krause, qui nous a pondu Fastlane, qui m'aide ÉNORMÉMENT.

L'App Store

Et enfin, la review. Tu soumets ton app, tu attends, tu espères. Les guidelines font des centaines de pages et changent régulièrement. Ce qui passait l'an dernier peut être rejeté aujourd'hui.

Les réponses de la review team sont souvent vagues — *"Guideline 4.2 - Minimum Functionality"* —, et tu dois deviner ce qu'il faut corriger.

Ce que je dis aux développeurs que je forme : préparez-vous à être frustrés. Pas par le code — le code, c'est la partie fun. Par tout ce qui l'entoure.

Ton sentiment sur Liquid Glass (iOS 26) ?

Apple n'avait pas touché au design système depuis iOS 7 en 2013. Douze ans de stabilité, avec des ajustements mineurs, mais aucune refonte profonde. Liquid Glass, annoncé en 2025 pour iOS 26, est donc un évènement.

Ce qui m'enthousiasme :

L'audace. Apple aurait pu continuer à itérer sur le design existant, jouer la sécurité. Ils ont choisi de tout repenser : translucidité généralisée, effets de profondeur, surfaces qui réagissent au contexte. C'est beau, c'est vivant, c'est différent.

L'unification. Pour la première fois, le même langage visuel traverse toutes les plateformes — de l'Apple Watch à visionOS (OK pas tout an fait, mais presque quand même). Quand tu développes une app multi-plate-forme, cette cohérence est précieuse.

SwiftUI natif. Si ton app est en SwiftUI, tu bénéficies automatiquement du nouveau design. Pas de migration, pas de code à réécrire. C'est la récompense pour ceux qui ont adopté le framework tôt, malgré ses bugs de jeunesse.

Ce qui m'inquiète :

La lisibilité. Le verre translucide, c'est magnifique sur les démos Apple avec leurs fonds parfaitement choisis. Dans la vraie vie, sur une photo chargée ou un fond clair, les textes deviennent illisibles.

La migration UIKit. Si ton app est en UIKit, comme la majorité des apps professionnelles, tu as du travail. Les contrôles système s'adaptent automatiquement, mais les composants custom, les designs sur mesure, les interfaces complexes — tout ça demande de l'attention.

Ce qui te manque au quotidien ?

Une documentation unifiée. Actuellement, pour comprendre une API Apple, tu consultes la documentation officielle (parfois datée), les vidéos WWDC (excellentes), les forums Apple Developer (souvent sans réponse), Stack Overflow (qualité variable), et des blogs de développeurs (qualité excellente, mais dispersés) et maintenant les IA, mais leur entraînement est parfois limitant.

Des outils de migration automatique. Chaque année Apple déprécie des API. "Cette méthode sera supprimée dans iOS 27." Très bien. Mais où est l'outil qui migre automatiquement mon code vers la nouvelle API ? Un peu comme des projets web qui proposent des `codmob`.

De la transparence sur l'App Store Review. J'ai soumis des dizaines (et des dizaines) d'apps en dix-huit ans. Je ne comprends toujours pas les décisions de la review team. Un rejet devrait venir avec une explication claire, des exemples précis. Aujourd'hui, c'est un mail générique, un lien vers une guideline de 50 pages. D'un autre côté avec 7 millions d'apps soumises par an faire un diagnostic précis est impossible. Sur ces mises à jour, je me sens un peu privilégié, car mon parcours m'a permis d'avoir certains précieux contacts qui peuvent souvent débloquer la situation, mais ici je parle au nom de tous.

ET DEMAIN ?

Cinquante ans après sa fondation, Apple n'est plus le challenger du garage de Los Altos (blague à part, le propriétaire de Chrst Drive n'est pas agréable du tout, on y a été avec les francophones de la WWDC). C'est l'une des entreprises la plus valorisée au monde, avec une responsabilité immense envers les millions de développeurs qui ont bâti leur carrière sur ses plateformes.



Swift comme langage universel. Swift sur serveur avec Vapor c'est pas mal, Swift compilé en WebAssembly pour le web, Swift comme alternative à Rust pour la programmation système. Apple a créé un angage magnifique; il mérite de dépasser les frontières de l'écosystème Apple.

L'IA au service du développeur. Codex & Claude montrent la voie, Apple suit. Je pense que c'est à eux de montrer la voie, mais comme assistant réel pas simplement en mode prompt → génération de code, mais en debug assisté, suggestions d'architecture... Les outils tiers comme Cursor prouvent que c'est possible. Apple a les moyens de faire mieux.

La maturité complète de SwiftUI. Dans trois ans, pour ses 10 ans, SwiftUI doit pouvoir tout faire. Plus de cas où tu dois "descendre" en UIKit pour une fonctionnalité manquante. Une parité totale, et UIKit qui devient progressivement le legacy qu'on maintient sans y toucher.

visionOS et le spatial computing. Le Vision Pro est cher, mais c'est le futur. Les développeurs qui s'y investissent maintenant — comme ceux qui ont sauté sur l'iPhone en 2008 — auront une longueur d'avance quand le marché décollera.

Ce que je n'espère pas :

Des révisions dans le DMA. Soyons clairs : je suis profondément opposé à la façon dont le Digital Markets Act a été conçu et imposé. C'est une régulation écrite par des bureaucrates qui n'ont jamais publié une app, sous la pression de géants comme Epic et Google qui défendent leurs propres intérêts, pas ceux des développeurs plus ou moins indépendants.

Le modèle App Store me protège. La commission de 15% pour les petits développeurs — car oui, ce n'est plus 30% pour ceux qui font moins d'un million de revenus —, c'est un deal honnête. En échange, Apple gère le paiement mondial, la fraude, la distribution, les serveurs, la facturation TVA dans 175 pays. Essaie de faire ça seul. Calcule ce que ça coûte vraiment.

Le DMA produit de la fragmentation. Des stores alternatifs de qualité douteuse. Des problèmes de sécurité — car Apple ne pourra plus garantir l'intégrité des apps installées hors App Store. Et au final, ce sont les utilisateurs et les petits développeurs qui paieront. Epic a son store, Google aura le sien, et nous, les indépendants, on devra supporter trois canaux de distribution avec les ressources d'un solo ou d'une petite équipe.

J'ai plusieurs fois participé aux discussions avec la Commission Européenne sur ces sujets. Le fossé entre leur compréhension théorique du marché et la réalité quotidienne d'un développeur indépendant startup est vertigineux. Ils pensent défendre les petits contre les gros. En pratique, ils servent les intérêts des très gros — les Epic, les Spotify, les Meta — au détriment de ceux qu'ils prétendent protéger.

En conclusion

Cinquante ans d'Apple, c'est une histoire de paris audacieux. Le Macintosh contre IBM. L'iPhone contre Nokia. Swift contre Objective-C. Chaque génération de développeurs a dû s'adapter, apprendre, parfois tout recommencer.

Ce qui me fascine, c'est que cette histoire continue. En 2026, être développeur Apple, c'est naviguer entre SwiftUI et UIKit, entre le confort de l'App Store et les menaces réglementaires, entre l'enthousiasme pour visionOS et le pragmatisme des projets qui paient les factures.

Ce métier m'a tout donné : une carrière, une communauté, des amis, des rencontres à travers le monde entier. Si je devais recommencer à coder aujourd'hui, sur quelle plateforme choisirais-je de me lancer ? La même. Sans hésitation.

UN COUP DE CHAPEAU AUX DEVELOPER RELATIONS

On ne peut pas parler de l'écosystème Apple sans mentionner les Worldwide Developer Relations (WWDR), ceux qui font le lien entre Cupertino et nous. En France, Michel (Sutter) a pavé la voie à une époque où le Mac était une niche et l'iPhone n'existait pas encore. Mac (Alnakari) et SiQing (Lin) ont pris le relais, toujours présents et à l'écoute. Désormais, c'est Martin (Vialle) qui porte le flambeau — après 10 ans aux manettes de l'App Store, il connaît le terrain. À vous quatre : merci de défendre les développeurs français à Cupertino. Merci ! Sans vous, l'écosystème France ne serait certainement pas ce qu'il est.

Swift : 12 ans d'évolution, du mobile au cloud



Sébastien Stormacq

Sébastien code depuis l'époque du Commodore 64 et n'a jamais vraiment arrêté. Aujourd'hui Developer Advocate chez Amazon Web Services, il accompagne les développeurs dans la construction de systèmes cloud performants, distribués et automatisés. Il s'intéresse particulièrement aux outils pour développeurs, aux architectures modernes et à tout ce qui permet d'aller plus vite sans sacrifier la qualité.

Conférencier régulier aux AWS Summit, blogueur sur l'AWS News Blog et animateur du podcast AWS en français, il aime rendre accessibles des sujets techniques complexes — toujours avec des exemples concrets et du cod prêt à compiler.

En 2014, Apple lançait le langage de programmation Swift pour remplacer Objective-C. Dix ans plus tard, le langage a dépassé les frontières d'iOS pour conquérir les serveurs Linux, le cloud, les systèmes embarqués, et même les appareils Android.

L'histoire commence en 2010. À l'époque, Chris Lattner, le créateur des compilateurs LLVM et Clang, imagine un nouveau langage de programmation pour se substituer à Objective-C, lui-même inventé dans les années 80 et hérité avec l'acquisition de NeXT et l'utilisation de NeXTSTEP comme fondation de Mac OS X.

Chris et son équipe conçoivent un langage performant, moderne, expressif et surtout sûr ! Plus question de gérer la mémoire à la main (malloc et free), plus de risque de recevoir une NullPointerException. Il s'inspire des qualités de langages établis ou naissants à l'époque, tels que Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, et d'autres.

Par exemple :

```
// En C
char *buffer = malloc(1024);
// ... utiliser buffer
free(buffer);
// En Swift
var buffer = Data(count: 1024)
// Pas de free(), ARC s'en charge dès que la variable buffer sort du scope
```

Je passe sur les tribulations et jeux politiques internes et en 2014, Apple présente Swift 1.0 à la conférence WWDC et invite les développeurs à l'utiliser pour coder leurs applications pour les OS maison, de l'iPhone au Mac en passant par les iPads.

La force d'Apple est de fournir des chemins de migration faciles (souvenez-vous des migrations des processeurs Motorola vers PowerPC vers Intel vers Apple Silicon). Swift s'intègre facilement dans l'existant Objective-C. L'idée au début était de pouvoir progressivement adopter le nouveau langage dans les applications existantes ou créer des nouvelles apps en utilisant les APIs offertes par les OS.

Il aura fallu quelques années à Apple pour offrir l'immense majorité des fonctions systèmes en Swift. En 2019, Apple présente SwiftUI, un nouveau framework déclaratif et réactif pour concevoir des interfaces graphiques pour tous leurs OS et progressivement remplacer les frameworks historiques tels que UIKit et AppKit. Il en est de même pour Combine, Swift Data et Core Data ou Reality Kit et Scene Kit.

Pourquoi les développeurs aiment Swift

Swift est rapide, expressif et sûr. Le langage possède de nombreux aspects qui rendent l'écriture de code Swift simple et amusante. J'en liste quelques-uns, dans l'ordre où j'y pense :

- Tout d'abord il est construit selon un principe de découverte progressive : construire des choses simples se fait simplement (vous vous souvenez de "public static void main(String[] args)" qui effrayait tous les développeurs débutants dans un autre langage ?). Nous n'avons pas besoin de connaître tous les détails pour commencer à coder et les constructions plus complexes ne sont nécessaires que quand on commence à faire des applications plus complexes.
- Les applications Swift sont compilées en instructions natives. Il n'est pas nécessaire d'utiliser une machine virtuelle ou un interpréteur pour les faire tourner. Les binaires Swift sont quasiment aussi performants que les binaires C ou C++.
- Le runtime Swift (les librairies standards liées à votre code) s'occupe de la gestion de la mémoire pour vous. Vous ne devez pas vous soucier d'allouer ou de désallouer de la mémoire. Swift utilise un système de compteur de références (Automatic Reference Counting - ARC) pour tenir compte des pointeurs vers les structures en mémoire : +1 quand vous copiez une référence et -1 quand elle sort du scope. La mémoire est automatiquement libérée quand il n'y a plus de référence vers un bloc de mémoire (quand le compteur est à zéro). Il n'y a pas de pause pour le ramasse-miettes (garbage collector). Les temps d'exécution sont prévisibles, sans pause. Par exemple :

```
class User { ... }

var u1: User = User()  // compteur = 1
var u2 = u1            // compteur = 2

u1 = nil               // compteur = 1
u2 = nil               // compteur = 0 → User est automatiquement libéré
```

- Le langage est fortement typé et les types sont vérifiés par le compilateur. Comme en Java, Rust ou TypeScript, il n'est pas possible d'assigner par erreur un String à un Int ou tout type de conversion contre nature. Je sais que certains développeurs Python ou JavaScript n'aiment pas cette rigueur imposée par le compilateur. La règle est : au plus tôt on détecte une erreur, au moins cher elle coûte à corriger. Les erreurs détectées par le compilateur sont les moins chères à corriger.
- En parlant de type, Swift a été un des premiers langages à

populariser les types optionnels. En Swift, si une variable peut soit avoir une valeur, soit être nulle, vous devez l'indiquer avec un ?. Par exemple let name: String? = nil. Quand j'utilise des variables de ce type, le compilateur m'oblige à tester le cas où la variable serait nulle. Cette vérification au moment de la compilation permet d'éviter les terribles erreurs NullPointerException connues dans les autres langages. Par exemple :

```
// fetchName() est une fonction qui peut retourner une String ou nil
func fetchName() -> String? { ... }

let name: String? = fetchName()

// Le compilateur oblige à traiter le cas nil
if let safeName = name {
  print("Bonjour \(safeName)")
} else {
  print("Nom inconnu (nil)")
}
```

- Swift offre des primitives natives pour le support de la concurrence. Outre les désormais standards async et await pour coder des appels asynchrones non bloquants, le compilateur s'assure de la bonne gestion des états mutables et partagés, comme en Rust. En clair, il détecte les risques d'erreurs liées aux modifications simultanées de données (les fameuses "data race"). Swift utilise une approche différente des autres langages. Bien qu'il y ait le concept de "owner" et "borrower", il s'agit d'un mécanisme avancé qu'il n'est nécessaire de comprendre que dans des cas plus avancés (souvenez-vous du principe de découverte progressive). Swift propose le concept de domaines d'isolation. Deux tâches qui travaillent en parallèle appartiennent à deux domaines d'isolation différents et ne peuvent pas partager des structures de données modifiables. Si je définis une structure de données qu'on peut partager entre tâches (soit parce qu'elle est non modifiable, soit parce qu'elle utilise des mécanismes comme des locks pour protéger les valeurs modifiables), je la marque comme étant Sendable. Le compilateur va vérifier que seules des structures de données Sendable sont échangées entre tâches (entre domaines d'isolation pour être précis). Ça demande un peu de discipline au début, surtout quand on travaille avec des callbacks réseau, c'est tellement facile de capturer une variable dans une closure sans qu'on s'en rende compte. Mais une fois adopté, c'est le compilateur qui garantit que le code est correct. Si ça passe à la compilation, ça passe à l'exécution aussi (sauf quand on fait des trucs débiles comme désactiver les vérifications du compilateur :-)) Par exemple :

```
// les class sont modifiables par défault et passées par référence
class MutableUser {
  var name: String = ""
}

func process(user: MutableUser) async {
  Task {
    print(user.name)
    // ✗ erreur en Swift 6 strict concurrency car user est mutable et pourrait être modifié.
    // en parallèle pendantq ue Task s'exécute.
    // L'erreur exacte du compilateur est:
    // "Passing closure as a 'sending' parameter risks causing data races between code in the current task
    // and concurrent execution of the closure"
    // "Closure captures 'user' which is accessible to code in the current task"
  }
}
```

La solution est d'utiliser des structures de données non modifiables :

```
// Les struct ne sont pas modifiables par défault et passées par copie
// Sendable indique au compilateur que cette structure peut être envoyée au travers
// de domaines d'isolation différents. Dans ce cas simple, le compilateur infère `Sendable` automatiquement
// je l'ai ajouté à des fins pédagogiques.
struct User: Sendable {
  let name: String
}

func process(user: MutableUser) async {
  Task {
    print(user.name) // ✔ Ceci est correct car User n'est pas modifiable
  }
}
```

Swift 6 : un changement majeur

Apple présente Swift 6 à la conférence Worldwide Developer Conference (WWDC) 2024 et le lance en septembre de la même année. Le grand changement de cette nouvelle version est le renforcement des contrôles du compilateur liés à l'exécution concurrente de code et l'introduction du concept de concurrence structurée.

L'idée est simple mais puissante : une tâche concurrente est liée à un scope, exactement comme une variable locale. Souvenez-vous de la programmation structurée qui a remplacé les goto par des blocs (if, for, fonctions) où la durée de vie des ressources est prévisible. La concurrence structurée applique le même principe aux tâches asynchrones. Quand je lance des tâches en parallèle avec async let ou un TaskGroup, elles sont rattachées au scope qui les a créées. Le scope attend que toutes les tâches enfants se terminent avant de se fermer. L'annulation se propage automatiquement du parent vers les enfants. Les erreurs remontent naturellement, comme un throw dans du code synchrone. En pratique, si vous ouvrez un fichier ou une connexion réseau dans un scope qui lance des tâches concurrentes, vous avez la garantie que le nettoyage s'exécutera après que toutes les tâches soient terminées. Si vous avez déjà debuggé un callback oublié ou une tâche orpheline qui continue de tourner après que l'écran a disparu, vous apprécierez.

Concrètement, Swift offre deux constructions structurées principales. async letpermet de lancer une tâche enfant et de récupérer son résultat plus tard, comme on déclarerait une

variable : async let image = downloadImage(url) suivi de let result = try await image.
Par exemple :

```
func loadProfile() async throws -> Profile {
  async let user = fetchUser()
  async let avatar = downloadAvatar()

  // try await attend que toutes les tâches définies avec async let soient terminées ou annulées
  return try await Profile(
    user: user,
    avatar: avatar
  )
}
```

Les deux appels tournent en parallèle. La fonction ne retourne qu'une fois les deux terminés.
Le compilateur garantit que la tâche est attendue (ou annulée) avant la sortie du scope.

```
func parentTask() async throws {
  async let data = fetchLargeFile()

  throw SomeError() // annule automatiquement data
}
```

Pour un nombre dynamique de tâches, withTaskGroup crée un groupe dans lequel on ajoute des tâches à la volée et on collecte les résultats au fur et à mesure.

```
func downloadAll(_ urls: [URL]) async throws {
  try await withThrowingTaskGroup(of: Data.self) { group in
    // lance une sous-tâche pour chaque url à télécharger
    for url in urls {
      group.addTask {
        try await download(url)
      }
    }

    // boucle et attend les résultats un à un
    // il y aura autant de lignes imprimées que de URLs à télécharger
    for try await data in group {
      print("Une URL a retourné \(data.count) bytes")
    }
  }
}
```

Le scope ne se termine pas tant que toutes les tâches n'ont pas fini.
À l'opposé, Task { } et Task { } et Task.detached { } sont les échappatoires non structurées : elles créent des tâches qui vivent indépendamment du scope, un peu comme un Thread en Java ou un goroutine en Go. Elles restent utiles (par exemple pour lancer une opération depuis un contexte synchrone), mais on perd les garanties de durée de vie et de propagation d'annulation. Préférez toujours les constructions structurées et ne sortez l'artillerie non structurée que quand c'est vraiment nécessaire.

Dans l'exemple ci-dessous, cette tâche n'est liée à aucun scope. Elle peut continuer à tourner même si l'écran disparaît.

```
// A éviter à tout prix
Task.detached {
  await doWork()
}
```

Swift Open Source

En décembre 2015, Apple fait un geste que peu auraient anticipé : le code source de Swift est publié sur GitHub sous *licence* Apache 2.0. *Ce n'est pas* juste un coup de communication. Apple publie le compilateur, la librairie standard, le gestionnaire de paquets (Swift Package Manager), le debugger, et même le site swift.org.
Ce geste change tout. Des développeurs, hors de l'écosystème Apple, commencent à s'intéresser au langage, à le porter sur Linux, puis sur Windows et Android. Des entreprises comme IBM ou Google investissent dans l'écosystème. Des frameworks web apparaissent : Vapor, Kitura (maintenant abandonné) et, plus récemment, Hummingbird. La communauté explose. Aujourd'hui, le dépôt principal du compilateur compte plus de 1000 contributeurs et des centaines de paquets open source sont disponibles via le Swift Package Index. Sans cette ouverture, Swift serait resté un langage captif de l'écosystème Apple, aussi bon soit-il techniquement. C'est l'open source qui lui a donné ses ailes au-delà de l'iPhone.
L'évolution du langage elle-même se fait en open source, via le processus Swift Evolution. N'importe qui peut proposer un changement au langage en soumettant une proposition (un document numéroté, par exemple SE-0302 pour Sendable). La proposition est discutée publiquement sur les forums Swift, revue par la communauté et par le core team, puis acceptée, modifiée ou rejetée. Tout est transparent : les motivations, les alternatives considérées, les arguments pour et contre. Ce n'est pas une démocratie (le core team a le dernier mot), mais c'est un processus ouvert où votre voix compte si vos arguments sont solides. En pratique, de nombreuses fonctionnalités majeures de Swift 5 et 6 sont nées de propositions de développeurs extérieurs à Apple. Si vous voulez comprendre pourquoi une fonctionnalité du langage existe et quels compromis ont été faits, allez lire la proposition correspondante sur swift.org/swift-evolution, c'est souvent plus instructif que la documentation officielle.

Au-delà d'Apple

C'est côté serveur que Swift a d'abord fait ses preuves hors de l'écosystème Apple. Dès 2016, des frameworks web comme Vapor et Kitura (IBM) apparaissent pour écrire des backends en Swift sur Linux. En 2017, Apple publie SwiftNIO, une couche réseau asynchrone inspirée de Netty (le monde Java), qui devient la fondation de tout l'écosystème serveur. La même année, Apple crée le Swift Server Workgroup (SSWG), un groupe de travail avec des membres d'Apple, d'AWS, de Vapor et de la communauté. Aujourd'hui, deux frameworks web dominent : Vapor, le plus complet, et Hummingbird, plus léger et modulaire. Les avan-

tages de Swift sur un serveur sont les mêmes que sur un iPhone : des binaires natifs performants, pas de garbage collector (donc pas de pauses imprévisibles), une empreinte mémoire faible et un démarrage quasi instantané. Ce dernier point est particulièrement intéressant pour l'embarqué et les fonctions serverless où chaque milliseconde de cold start compte.
Avec Hummingbird, Une application web minimale qui répond à HTTP GET /hello est aussi simple que ça:

```
import Hummingbird

// 1. définir les routes
let router = Router().get { req, context in
  return "Hello, Swift!"
}

// 2. démarrer le server
let app = Application(router: router)
try await app.runService()
```

Et ce ne sont pas que des promesses théoriques. Cultured Code, l'éditeur de Things (une app de gestion de tâches qui a gagné deux Apple Design Awards), a migré son backend Things Cloud de Python 2 sur Google App Engine vers Swft avec Vapor. Après plus d'un an en production, les coûts de calcul ont été divisés par trois et les temps de réponse ont chuté. Le tout tourne sur quatre instances Kubernetes qui encaissent des pics à 500 requêtes par seconde. Leur stack? Vapor, SwiftNIO, MySQLKit, RediStack, Soto pour l'accès à Amazon S3, APNswift pour les notifications push. Du Swift de bout en bout.
Apple aussi migre ses services. En 2024, l'équipe derrière le service Password Monitoring (celui qui gère vos mots de passe et vous alerte quand un de vos mots de passe apparaît dans une fuite de données) a migré son backend de Java vers Swift. Les chiffres : 40% de gain de performance, une empreinte mémoire divisée par 50 (de dizaines de gigaoctets à quelques centaines de mégaoctets), une latence sous la milliseconde pour 99,9% des requêtes, 85% de lignes de code en moins. Le tout sur Linux, dans Kubernetes, avec Vapor, pour des milliards de requêtes par jour. Et ce n'est pas un cas isolé : lors de la keynote d'AWS re:Invent 2025, Payam Shodjai, VP Cloud Infrastructure chez Apple, a révélé que de nombreux services Apple tournent dans le cloud et sont écrits en Swift.
Depuis avril 2024, Apple pousse Swift dans une direction inattendue : les systèmes embarqués. Embedded Swift est un mode de compilation expérimental pour programmer des microcontrôleurs : du Swift sur Raspberry Pi Pico, un STM32 ou un ESP32 en RISC-V. Pour tenir dans quelques kilo-octets de mémoire, Embedded Swift désactive certaines fonctionnalités (la réflexion, les types existentiels, la stabilité ABI) et génère des binaires autonomes adaptés au firmware. On perd un peu de la richesse du langage, mais on garde la syntaxe, le typage fort et la sécurité mémoire. Pour ceux qui ont déjà écrit du C pour un microcontrôleur à 3 heures du matin en cherchant un buffer overflow, c'est une bouffée d'air frais. Voici un mini exemple de code Swift embarqué

```
@main
struct Blinky {
  static func main() {
    let led = GPIO(pin: 13)
    while true {
      led.toggle()
      sleep(1)
    }
  }
}
```

Côté desktop, Swift est officiellement supporté sur Windows depuis 2020. C'est The Browser Company qui a défriché le terrain en utilisant Swift pour construire Arc, leur navigateur web, sur Windows. Grâce à l'interopérabilité C++ introduite dans Swift 5.9 et au support du Component Object Model (COM), ils ont pu écrire leur logique applicative en Swift tout en accédant aux APIs Windows. Ce travail de pionnier a montré que Swift sur Windows n'était pas qu'une curiosité. En janvier 2026, Apple a formalisé cet élan en créant le Windows Workgroup pour maintenir et améliorer le support de la plateforme.
Et puis il y a Android. Des apps Swift tournent en production sur Android depuis des années : Spark, flowkey, MediQuo… des développeurs courageux qui ont fait le travail de cross-compilation à la main, avec des millions de téléchargements. Fin 2025, le projet devient officiel : le Swift SDK for Android est annoncé, avec des nightly builds à partir de Swift 6.3. Swift compile en code natif via le NDK Android, et l'interopérabilité avec les APIs Java/Kotlin passe par JNI grâce aux outils jextract et wrap-java. L'idée n'est pas de remplacer Kotlin pour l'UI, mais de partager le cœur de la logique applicative entre iOS et Android. Un seul code Swift, deux plateformes.

Les outils

Pendant longtemps, développer en Swift signifiait utiliser Xcode sur Mac, point final. C'est encore le cas pour les plateformes Apple, et Xcode reste un excellent IDE. Mais pour le développement serveur, embarqué ou cross-platform, l'écosystème d'outils s'est étoffé et il n'est plus nécessaire de posséder un Mac pour écrire du Swift.
Commençons par l'installation. Jusqu'à récemment, installer Swift sur Linux ou Windows demandait de télécharger manuellement une archive depuis swift.org, l'extraire au bon endroit, configurer le PATH… bref, le genre de manipulation qui fait fuir les débutants et agace les habitués. En mars 2025, l'équipe Swift a publié Swiftly, un gestionnaire de versions inspiré de rustup (l'équivalent chez Rust). Une commande pour installer Swift, une autre pour changer de version, le support des snapshots nightly, et un fichier .swift-version à la racine du projet pour que toute l'équipe utilise la même version. Swiftly fonctionne sur macOS et les principales distributions Linux. Il est lui-même écrit en Swift, ce qui est plutôt élégant pour un outil d'amorçage.

```
~ swiftly install 6.2.3
...

~ swiftly use 6.2
```

```
The global default toolchain has been set to `Swift 6.2.3` (was 6.2.3)

~ swiftly list
Installed release toolchains
--------------------------
Swift 6.2.3 (in use) (default)

Installed snapshot toolchains
---------------------------

Available system toolchains
-------------------------
xcode
```

Côté éditeur, Visual Studio Code est devenu le compagnon naturel de Swift hors de Xcode. L'extension Swift pour VS Code, désormais officiellement maintenue par l'équipe Swift chez Apple, offre la complétion de code, la navigation, les diagnostics en temps réel, le débogage via LLDB, le Test Explorer et la gestion des packages Swift Package Manager. Tout ça repose sur SourceKit-LSP, le serveur de langage de Swift. Et SourceKit-LSP n'est pas lié à un éditeur particulier : Neovim, Emacs, n'importe quel éditeur compatible LSP en bénéficie.

L'extension est aussi publiée sur Open VSX, le registre d'extensions open source géré par la Eclipse Foundation. Pourquoi c'est important ? Parce que les éditeurs comme Kiro, Cursor, Windsurf, Gitpod ou tout environnement basé sur Eclipse Theia utilisent Open VSX et non le marketplace de Microsoft. En publiant sur les deux registres, l'équipe Swift s'assure que le support fonctionne quel que soit votre éditeur.

Et les agents de code ? me demandez-vous. Kiro, Cursor, Windsurf, GitHub Copilot, Claude Code : ces outils qui utilisent des modèles de langage pour écrire, analyser et refactorer du code sont en train de transformer la manière dont on développe. La bonne nouvelle, c'est qu'ils fonctionnent avec Swift. Les agents basés sur VS Code bénéficient directement de l'extension Swift et de SourceKit-LSP. Les agents en ligne de commande comme Claude Code interagissent avec le compilateur, le gestionnaire de paquets et les tests via le terminal. On peut leur donner du contexte via des fichiers de règles (un CLAUDE.md ou un .cursorrules à la racine du projet) pour qu'ils respectent les conventions de l'équipe. Ce n'est pas encore parfait (les modèles connaissent mieux Python ou TypeScript que Swift, il faut parfois les guider), mais un langage fortement typé avec un compilateur strict et un LSP qui fournit des diagnostics précis, c'est un terrain plutôt favorable pour les agents. Le compilateur rattrape leurs erreurs, exactement comme il rattrape les nôtres.

La communauté

Un langage de programmation, aussi élégant soit-il, ne vaut rien sans une communauté qui le porte. Sur ce plan, Swift a de la chance.

Les échanges se font principalement sur les Swift Forums (forums.swift.org), des questions de débutants jusqu'aux débats pointus sur l'évolution du langage. Il y a aussi un Slack dédié à Swift open source, où l'ambiance est décontractée et l'entraide est la norme. Ce qui frappe quand on débarque, c'est la bienveillance. Les questions naïves reçoivent des réponses détaillées et respectueuses, les désaccords techniques restent courtois. Ce n'est pas un accident : le projet porte des valeurs d'inclusion et de diversité qui se traduisent concrètement par des groupes de soutien sur les forums (Women in Swift, Black in Swift, Pride in Swift) et un groupe de travail Diversity in Swift qui veille à ce que ces valeurs ne restent pas de belles paroles.

Côté événements, la communauté s'est taillé une place au FOSDEM, la plus grande conférence open source indépendante au monde, à Bruxelles. En 2024, Swift obtient pour la première fois un track complet : 12 présentations qui couvrent le serveur, l'embarqué, les IDE, le déploiement Linux, et même un visualiseur de musique à base de ferrofluide piloté en Embedded Swift (oui, vous avez bien lu). La salle était pleine à craquer. Doug Gregor, un des architectes du langage, est intervenu dans le track principal sur la sécurité mémoire. En 2025, la communauté va plus loin avec un pre-FOSDEM Community Event le vendredi après-midi, avec des talks et un moment de convivialité. Pour un langage que certains percevaient comme captif d'Apple, être présent au FOSDEM est un signal fort.

Et si vous ne pouvez pas vous déplacer à Bruxelles, pas de problème. Depuis plus de deux ans, des meetups en ligne rassemblent régulièrement la communauté pour des présentations techniques. Les vidéos sont toutes disponibles sur la chaîne YouTube Swift Community Meetups.

Le futur

Swift a 12 ans et n'a jamais avancé aussi vite. L'interopérabilité s'étend dans toutes les directions : C++ depuis Swift 5.9, Java en cours de développement actif, C depuis le début et en amélioration constante. Embedded Swift mûrit à chaque release. Le SDK Android est en nightly preview, le Windows Workgroup vient d'être créé, et WebAssembly permet de faire tourner du Swift dans un navigateur. Chaque version de Swift 6.x continue d'affiner la concurrence structurée et de rendre la sécurité mémoire plus accessible. La trajectoire est claire : Swift veut être un langage universel, aussi à l'aise sur un microcontrôleur à 3 euros que sur un cluster Kubernetes qui traite des milliards de requêtes. Et quand on voit le chemin parcouru depuis le premier "Hello, World" à la WWDC 2014, force est de constater qu'il y est déjà.

PROGRAMMEZ! N°274
3 avril 2026

LES PROCHAINS NUMÉROS À PARAITRE

PROGRAMMEZ! N°275
29 mai 2026

Xcode : un héritage de 30 ans… (en musique)

Avant de parler de ce qu'il faut installer aujourd'hui pour développer pour les plateformes Apple, il est essentiel de comprendre doù viennent ces outils. Contrairement à Android où l'écosystème a été créé de toutes pièces par Google entre 2005 et 2007, la toolchain Apple plonge ses racines dans les années 80 (*What a feeling* ou *Enola Gay* passaient en boucle à la radio et moi… je n'étais pas né).



Clément Sauvage

Ingénieur de formation, Entrepreneur, Fondateur de (Bits'n Coffee) — Agence développement iOS, macOS et web basé à Lille

Liens / contact

https://clementsauvage.me



Le site de Clément



Cet article en musique

À la sortie de NeXTStep 3, NeXT y inclut Project Builder, un environnement de développement intégré, couplé à Interface Builder, *un éditeur visuel pour créer les interfaces graphiques en glissant-déposant des composants, puis de les connecter au code par de simples liaisons visuelles.* En 1992, ça tient de la magie. Les fichiers générés portent l'extension .nib — pour "NeXT Interface Builder" — une extension qui existe toujours aujourd'hui, trente-cinq ans plus tard (mais plus utilisée au profit des .xib (Xcode Interface Builder) et des storyboards. Comme quoi, certaines choses traversent les décennies mieux que les *one-hit wonders*.

Quand Apple rachète NeXT en 1996 et que Steve Jobs revient aux commandes, ces outils deviennent la base de Mac OS X. Project Builder est rebaptisé **Xcode** en octobre 2003, avec la sortie de Mac OS X 10.3 Panther. Beyoncé sortait *"Crazy in Love"*, et nous, développeurs, on tombait amoureux d'un IDE (...ou pas).

Xcode 8-26 : 10 ans, la maturité (2016-2026)

Chaque version majeure de Xcode accompagne une version majeure de Swift, comme un vieux vinyle qui traverse les années :

- **Xcode 8** (2016) : *"Cheap Thrills"* (Sia) effectivement, migrer de Swift 2 à 3 n'était pas cher payé en heures de travail... c'était GRATUIT, mais ça prenait trois jours.
- **Xcode 9** (2017): *"Despacito"* dominait les charts, et on avançait despacito aussi dans nos migrations.
- **Xcode 10** (2018) → Swift 4.2, puis 5.0 avec Xcode 10.2
- **Xcode 11** (2019): Swift 5.1 et **SwiftUI** — Billie Eilish murmurait *"Bad Guy"*, et SwiftUI était effectivement le bad guy qui allait chambouler nos habitudes UIKit (bon, pas dans la version 1, hein, faut pas abuser).
- **Xcode 12** (2020) → Support Apple Silicon (M1) — *"Binding Lights"* de The Weeknd, comme la vitesse de compilation qui nous aveuglait.
- **Xcode 13-15**: Swift 5.5 à 5.9, async/await — On était *"Running Up That Hill"* (Kate Bush - merci Stranger Things).
- **Xcode 16** (2024): Swift 6.0, il faudra beaucoup *"Espresso"* de Sabrina Carpenter pour migrer vers Swift 6.
- **Xcode 26** (2025) : Support de l'IA, puis de l'agent coding avec Xcode 26.3 (février 2026). *"Harder, Better, Faster, Stronger"* de *Daft Punk* — parce que c'est exactement ce que l'IA promet de nous rendre. Et si on est honnête, un peu de *"Help!"* (The Beatles) quand l'agent génère du code qu'on ne comprend pas.

LE SETUP DU DÉVELOPPEUR APPLE EN 2026

Prérequis matériel : le Mac obligatoire

"Hotel California" des Eagles : *"You can check out any time you like, but you can never leave."* Ça résume parfaitement l'écosystème Apple.

C'est la première contrainte, et elle est non négociable : **pour développer nativement pour iOS, il faut un Mac.**

Xcode ne fonctionne que sur macOS. Pas de version Windows, pas de version Linux. C'est le prix du "jardin clos" Apple. Comme chantaient les Clash : *"Should I stay or should I go?"* Si tu veux développer pour le monde d'Apple, la réponse est simple : tu restes.

Configuration minimale Xcode 26 :

- macOS Tahoe 26 ou ultérieur
- 16 Go de RAM minimum (32 Go recommandés pour l'IA locale — *"More, more, more"* comme disait Andrea True Connection)
- 60+ Go d'espace disque (Xcode + simulateurs + modèles IA)

Installation de Xcode

"Let's get it started" — Black Eyed Peas

Option 1 : Mac App Store (recommandé pour les débutants)

1. Ouvrir le Mac App Store
2. Rechercher "Xcode"
3. Cliquer "Obtenir" — téléchargement de ~15 Go (*"The Long and Winding Road", The Beatles)*
4. Au premier lancement, Xcode installe les composants système additionnels et télécharge les modèles IA

Option 2 : Téléchargement direct (développeurs avancés)

1. Se connecter sur developer.apple.com/download
2. Télécharger le fichier Xcode
3. Permet d'avoir plusieurs versions en parallèle — comme un DJ avec ses platines. (Attention au versonning de vos projets)

Installation des Command Line Tools :

```
xcode-select —install
```

Cette commande installe les outils en ligne de commande (git, make, compilateurs) sans l'IDE complet — utile pour les serveurs CI/CD. C'est le *Minimalist Mix* de Xcode.

L'APPLE DEVELOPER PROGRAM

Gratuit vs Payant : que peut-on faire ?

"Money money money, must be funny, in a rich man's world" — ABBA savait de quoi ils parlaient.

Sans inscription (Apple ID gratuit) :

- Télécharger Xcode
- Accéder à la documentation
- Tester sur simulateurs
- Installer sur son propre appareil (limité à 7 jours, 3 appareils) — *"Seven Days"* (Craig David), mais en moins romantique
- Utiliser l'assistant IA de Xcode 26 (fonctionnalités limitées)

Avec l'Apple Developer Program (99$/an) :

- Publier sur l'App Store — *"Livin' on a Prayer"* (Bon Jovi) que les utilisateurs téléchargent
- TestFlight (distribution beta)
- Jusqu'à 100 appareils de test enregistrés par type (100 iPhones, 100 iPads...)
- Certificats et provisioning profiles de production
- Accès aux betas iOS/macOS — être le premier sur les nouveautés, comme les fans qui campent pour un concert ou 10h devant un Apple Store pour être le premier (*"...j'y ai joué aussi je sais de quoi j'cause" - Morgane de toi* - Renaud)
- Agent coding Xcode 26.3 complet — *"The Future's So Bright, I Gotta Wear Shades"* (Timbuk 3)

Apple Developer Enterprise Program (299$/an) :

- Distribution interne sans App Store
- Réservé aux entreprises de +100 employés
- *"Working 9 to 5"* de Dolly Parton, version corporate

Le cauchemar du signing : certificats et provisioning profiles

"Welcome to the jungle" — *Guns N' Roses*. C'est LE point de friction historique du développement iOS. Même après 18 ans d'App Store, la gestion des certificats reste obscure pour beaucoup. Si le signing était une chanson, ce serait *"Comfortably Numb"* de Pink Floyd — parce qu'après des heures à débugger un problème de certificat, vous ne ressentez plus rien.

Les composants du signing :

- **Certificat de développement** : Prouve votre identité de développeur — *"Therefore I Am"* comme dirait Billie Eilish (ou votre "ça" comme dirait Freud, selon les générations)
- **Certificat de distribution** : Nécessaire pour publier sur l'App Store
- **App ID** : Identifiant unique de votre application (ex: co.bitsnandcoffee.onyria)
- **Provisioning Profile** : Lie le certificat, l'App ID et les appareils autorisés — le DJ qui mixe tout ensemble

Comment ça fonctionne :

Certificat + App ID + Devices = Provisioning Profile

Provisioning Profile + Code = App signée

C'est comme une recette de cuisine, sauf que si vous vous trompez d'ingrédient, au lieu d'un pudding à l'arsenic, vous avez un message d'erreur cryptique et trois heures de debug. *"Killing Me Softly"*, *Fugees*.

Automatic Signing (recommandé) :

Depuis Xcode 8, le "Automatic Signing" gère tout ça automatiquement. Dans les paramètres du projet, cochez "Automatically manage signing" et Xcode crée les certificats et profiles nécessaires. C'est le *"Easy"* de Commodores — enfin, quand ça marche.

Quand ça se complique :

- Équipes multiples sur le même projet — *"Too Many Cooks"*
- CI/CD avec des machines headless — *"Invisible" de U2*
- Distribution Enterprise — *"Corporate World"*
- Certificats expirés ou révoqués — *"Another One Bites the Dust"*, *Queen*

Mon conseil après 19 ans : utilisez le signing automatique tant que possible. Ne passez en manuel que si vous avez une raison spécifique et que vous comprenez exactement ce que vous faites. Sinon, vous allez passer de *"Happy" à "Mad World"* en 3 minutes 45.

Felix, le sauveur : Fastlane

"Heroes" — (David Bowie). En 2014, pendant que le monde découvrait Swift, un développeur autrichien de 20 ans nommé Felix Krause en avait marre. Marre de passer des heures sur le signing. Marre des screenshots App Store à faire manuellement pour chaque taille d'écran. Marre de l'upload via Application Loader qui plantait une fois sur trois.

Alors il a créé **Fastlane**. Et il a changé la vie de centaines de milliers de développeurs iOS.

Ce que Fastlane automatise :

- **match** : Synchronise les certificats et profiles entre toute l'équipe via un repo Git privé. Plus jamais de "ça marche sur ma machine, mais pas sur la vôtre". *"We're All in This Together"*, High School Musical.
- **gym** : Compile et archive l'app en une commande. Fini les 47 clics dans Xcode. *"One"*, U2.
- **pilot** : Upload sur TestFlight automatiquement. *"Fly Away"*, Lenny Kravitz.
- **deliver** : Soumet sur l'App Store avec métadonnées et screenshots. *"Special Delivery"*.
- **snapshot** : Génère les screenshots pour toutes les tailles d'écran et toutes les langues. *"Picture This"*, *Blondie*.
- **scan** : Lance les tests.

Un exemple de Fastfile :

```
lane :beta do
  match(type: "appstore")
  gym(scheme: "MonApp")
  pilot(skip_waiting_for_build_processing: true)
  slack(message: "Nouvelle beta disponible ! 🚀 ")
end
```

Une commande. fastlane beta. Et tout se fait.

L'histoire de Felix :

Felix a développé Fastlane en open source, gratuitement. Le projet a explosé. Twitter (qui venait de racheter Crashlytics) l'a *acquihired*, puis en 2017, Google rachète Fastlane et embauche Felix. Il quitte Google en 2019 pour créer d'autres projets open source, notamment autour de la vie privée.

À 32 ans à peine, Felix a probablement économisé des millions d'heures de travail cumulées à la communauté iOS mondiale. Si le développement iOS avait un *Hall of Fame*, il serait juste derrière Chris Lattner.

"Thank You for Being a Friend" — Andrew Gold (oui, le générique des Golden Girls, mais c'est sincère).

Mon conseil :

Apprenez Fastlane. Même si Xcode Cloud existe maintenant, même si l'Automatic Signing fonctionne mieux qu'avant, Fastlane reste l'outil le plus fiable pour automatiser tout le pipeline de release. C'est le *"Stairway to Heaven"* de la toolchain iOS : long à maîtriser, mais une fois que vous connaissez, vous ne pouvez plus vous en passer.

Et si un jour vous croisez Felix dans une conférence, offrez-lui une bière. Il l'a méritée.
"Cheers (Drink to That)" — (Rihanna), moi c'est déjà fait !

TESTFLIGHT : LA DISTRIBUTION BETA

Un peu d'histoire

TestFlight n'a pas toujours appartenu à Apple. Fondé le 23 décembre 2010 par Benjamin Satterfield et Trystan Kosmyrka — pendant que Katy Perry était en mode *"Firework"* — c'était une startup indépendante qui simplifiait la distribution de builds beta. À l'époque, pour distribuer une beta, il fallait collecter les UDID des testeurs (un identifiant unique de 40 caractères hexadécimaux), les ajouter manuellement au portail développeur, générer un nouveau provisioning profile, et leur envoyer des fichiers .ipa par email. *"Ain't No Sunshine"* de Bill Withers résumait bien l'expérience.
Apple rachète Burstly (la société mère de TestFlight) en février 2014 et arrête immédiatement le support Android (Biiim). Le service est relancé nativement avec iOS 8 en septembre 2014. *"Shake It Off"* de Taylor Swift (Bah oui, ne pas mettre de Taylor Swift dans un article sur Apple, ça aurait fait bizarre) sortait la même année — et on pouvait enfin secouer nos vieux workflows ad-hoc.

TestFlight aujourd'hui

"I Gotta Feeling" — Black Eyed Peas. *Tonight's gonna be a good night...* pour les beta testers.
Capacités :

- Jusqu'à 25 testeurs internes (membres de l'équipe App Store Connect)
- Jusqu'à 10 000 testeurs externes via email ou lien public — assez pour remplir l'Accor Arena
- Builds valides 90 jours — "90 *Days*", la version développeur de *"500 Miles"*
- Crash reports automatiques
- Feedback intégré avec captures d'écran annotées

Workflow typique :

1. Archiver l'app dans Xcode — *"Pack Up"*
2. Upload vers App Store Connect — *"Upload"* (même si ce n'est pas une chanson, ça devrait)
3. Soumettre à la Beta App Review (24-48h) — *"Patience"*, (Guns N' Roses)
4. Inviter les testeurs — *"Come Together"*, (The *Beatles)*
5. Collecter les retours et crashs — *"Fix You"*, (Coldplay)

LA TOOLCHAIN COMPLÈTE EN 2026

Outils Apple intégrés

"We Are Family" — Sister Sledge
Xcode 26 reste le patriarche incontesté de la famille. L'IDE principal où tout commence et tout finit — compilation, debug, design d'interface, et désormais assistance IA. Si Xcode était un album, ce serait *"The Wall"* de Pink Floyd : monumental, parfois oppressant, impossible à ignorer. *"Man and Machine"* (Robbie Williams) résume bien ce qu'il est devenu avec l'intégration de l'agent coding.

Simulator permet de tester sur des dizaines d'appareils virtuels sans vider votre compte en banque chez Apple (si ce n'est pas déjà fait). iPhone 16 Pro Max, iPad mini, Apple Watch Ultra — tout ça sur votre écran. *"Virtual Insanity"* (Jamiroquai), évidemment. Le seul bémol : pas de push notifications (enfin, pas facilement), pas de caméra, pas d'App Store. C'est une simulation, pas la réalité.

Instruments est l'outil que vous ouvrez quand votre app rame et que vous ne savez pas pourquoi. Profiling mémoire, CPU, réseau, Energy Log pour la batterie. C'est le cardiologue de votre application — il trouve les fuites et les goulots d'étranglement. *"Under Pressure"* de Queen et Bowie s'impose. Quand Instruments vous montre un memory leak de 500 Mo, vous comprenez la chanson différemment.

TestFlight gère votre distribution beta comme un roadie gère une tournée. Jusqu'à 10 000 testeurs externes, crash reports automatiques, feedback intégré. *"Fly"* de Sugar Ray pour l'envol de vos builds vers vos testeurs.

App Store Connect est le backstage de votre carrière sur l'App Store. Analytics, ventes, reviews utilisateurs, gestion des métadonnées. C'est là que vous voyez si votre app cartonne ou si elle fait un *"Fade to Grey"* (Visage) dans les charts.

Xcode Cloud est la solution CI/CD d'Apple, lancée en 2021 et enfin mature. Build dans le cloud, tests automatisés, déploiement sur TestFlight sans lever le petit doigt. *"Head in the Clouds"*— littéralement. Le prix pique un peu, mais c'est intégré nativement, et ça juste marche.

L'agent coding de Xcode 26.3 : révolution ou gadget ?
"Fate of Ophelia" — *Taylor Swift* (allez une dernière, je suis un Swifty, mais dans plus le sens Swift que Taylor) *"All that time I sat alone in my tower, you were just honing your powers"* — Taylor aurait pu écrire ça pour nous, développeurs Apple, qui avons passé des années à galérer avec le signing, les certificats, les builds manuels. Pendant ce temps, Xcode "honait ses pouvoirs" dans les labos de Cupertino.

Depuis février 2026, Xcode 26.3 intègre un **agent coding**, capable de :

- Générer du code SwiftUI à partir de descriptions en langage naturel — vous décrivez, il code
- Proposer des corrections de bugs avec explications détaillées — pas juste le fix, le *pourquoi*
- Écrire des tests unitaires automatiquement — la corvée que personne n'aime faire
- Refactorer du code legacy Objective-C vers Swift moderne — le Saint Graal pour les vieux (...projets, pas développeurs)

Mon avis ? *"Keep it one hundred"*, comme dit Taylor.
C'est impressionnant sur les tâches répétitives. L'agent vous fait gagner un temps fou sur le boilerplate SwiftUI, les patterns classiques, les extensions triviales.
Mais il faut relire chaque ligne.
Parce que parfois, l'agent hallucine. Il invente des méthodes SwiftUI qui n'existent pas. Il crée des APIs sorties de son imagination. Et si vous copiez sans vérifier, vous risquez le sort d'Ophelia

originale — celle qui *"lived in fantasy" jusqu'à ce que "the venom stole her sanity"*. La différence entre un junior et un senior en 2026 ? Le senior relit le code de l'IA. Le junior fait confiance.
"And if you'd never come for me, I might've drowned in the melancholy". Honnêtement ? Avant l'agent, certaines tâches me déprimaient.

- Écrire un Codable (qui vous convertit un JSON du serveur en structure), 200 lignes de tests unitaires pour une feature triviale.
- Générer des previews SwiftUI pour 15 configurations différentes.

Maintenant, *"I swore my loyalty to me, myself and I, right before you lit my sky up"* — l'agent a rallumé une flamme.
Attention quand même : "You light the match to watch it blow".
L'agent est puissant. Mal utilisé, il peut foutre le feu à votre projet — du code généré que personne ne comprend, des architectures bancales, des dépendances inutiles. *"Quite the pyro"*, effectivement.
Mon conseil ? *"Pledge allegiance to your hands, your team, your vibes"*. L'agent est un outil, pas un remplaçant. Il amplifie ce que vous savez déjà. Si vous maîtrisez Swift et SwiftUI, il vous rend 3x plus rapide. Si vous débutez, il risque de vous noyer dans du code que vous ne comprenez pas.
"It's 'bout to be the sleepless night you've been dreaming of" — et pour une fois, c'est une nuit blanche productive, pas une nuit blanche de debug.
Swift Package Manager (SPM) est désormais le standard. Intégré à Xcode depuis la version 11, il a enfin atteint la maturité. Plus besoin de Podfile, plus de `pod install` qui prend 10 minutes. Vous ajoutez une dépendance, Xcode la résout. *"Native New Yorker"*, version Cupertino — c'est local, c'est intégré, c'est chez soi.

Outils tiers recommandés

Gestion de dépendances :
CocoaPods est en train de mourir. Pas "en déclin", pas "moins populaire" — *mourir*. Après 15 ans de bons et loyaux services, l'annonce est tombée : **le 2 décembre 2026, CocoaPods trunk deviendra définitivement read-only**. Plus de nouveaux pods. Plus de mises à jour. Game over. *"The Show Must Go On"* de Queen, sauf que là, le show s'arrête.
C'est la fin d'une époque. CocoaPods a été pendant des années LA solution pour gérer les dépendances iOS. Avant SPM, avant Carthage, il y avait CocoaPods. Des millions de projets l'utilisent encore. Des centaines de milliers de pods dans le registry. Et dans quelques mois, tout ça sera figé dans l'ambre, comme un fossile du développement iOS des années 2010.
Si votre projet utilise encore CocoaPods — et statistiquement, il y a de bonnes chances —, c'est le moment de migrer vers SPM. Pas demain. Pas "quand on aura le temps". Maintenant. *"Time Is Running Out"* de Muse n'a jamais été aussi pertinent.
Carthage est en soins palliatifs. *"Fade to Grey" de Visage*. Il a eu son heure de gloire pour ceux qui voulaient éviter CocoaPods sans attendre SPM, mais aujourd'hui, il n'a plus vraiment de raison d'être. Rest in peace (bientôt).

CI/CD :
Fastlane reste indispensable, même en 2026. Felix Krause a créé un monstre — dans le bon sens. Automatisation du signing, des screenshots, de l'upload TestFlight et App Store. *"Fast Car"* de Tracy Chapman : ça roule, ça roule vite, et ça vous emmène où vous voulez.

GitHub Actions est le choix évident pour les projets open source. CI gratuite, intégration native avec GitHub, marketplace d'actions prêtes à l'emploi. *"Free Fallin"* de Tom Petty — gratuit (enfin raisonnablement) et libérateur.

Xcode Cloud monte en puissance. L'intégration native avec Xcode est son argument massue. Vous configurez votre CI sans quitter votre IDE. *"In the Air Tonight"* de Phil Collins — vous sentez que quelque chose se prépare, et quand le drop arrive (le build réussi), c'est satisfaisant.

Debug & Monitoring :
Charles *"Old School"* ou **Proxyman** *"New Rules"* inspectent tout votre trafic réseau. Chaque requête, chaque réponse, chaque header. *"Every Breath You Take"* de The Police — ils surveillent tout ce que votre app envoie et reçoit. Indispensable pour débugger les APIs. Charles est le vétéran, Proxyman est le petit nouveau natif macOS avec une UI moderne.
Sentry va plus loin avec le monitoring d'erreurs non fatales, les breadcrumbs, les replays de session. *"Somebody's Watching Me"* de (Rockwell) — parce que Sentry voit tout ce qui se passe dans votre app, même ce que vous préféreriez ignorer.
Amplitude est devenu incontournable pour l'analytique produit. Comprendre comment vos utilisateurs naviguent dans votre app, où ils abandonnent, ce qui les fait revenir. Ce n'est pas du debug technique, c'est du debug comportemental. *"Every Move You Make"* — toujours The Police, décidément ils auraient dû bosser dans la tech. Vous voulez savoir si votre nouvelle feature est utilisée ? Amplitude vous le dit. Vous voulez comprendre pourquoi votre taux de rétention J7 s'effondre ? Amplitude vous le montre. *"The Numbers"* de Radiohead — parce qu'au final, tout est dans les chiffres.

Assistants IA (hors Xcode) :
Claude (Anthropie) pour l'architecture, les discussions techniques, la rédaction de documentation, et les débats philosophiques sur le DMA à 3h du matin, et le code bien sur.

Codex d'OpenAI, même came, génération de code à partir de prompts en langage naturel, refactoring intelligent, explication de code legacy incompréhensible. *"Rosetta Stone"* (Young Thug), parce que c'est exactement ça : un traducteur entre ce que vous voulez et ce que la machine comprend.

Cursor est l'IDE IA-first qui fait fureur. Malheureusement, il ne gère pas le signing iOS ni les simulateurs. Vous codez dedans, vous buildez dans Xcode. *"So Close"* de Hall & Oates — si proche d'être parfait, mais pas encore. Le jour où Cursor gère la toolchain Apple complète, Xcode aura du souci à se faire. En attendant, c'est le mariage forcé : Cursor pour l'intelligence, Xcode pour l'exécution.

Sources :

MartianCraft, "Xcode Through the Years" ;
Apple Developer "Choosing a Membership" ;
Apple Developer Program "What's Included"

Apple, trop en avance sur son temps ?

DU NEWTON (1993) À L'IPAD (2010), UNE REVANCHE… ET UNE LEÇON POUR DÉVELOPPEURS



Franck Perthuis
Expert en développement Swift (iOS, tvOS, macOS et visionOS)

Pour les 50 ans d'Apple, on retient volontiers les "hits" : Macintosh, iPod, iPhone… Mais l'histoire de la marque est aussi jalonnée de paris trop avant-gardistes, en particulier dans les années 90. Parmi eux, un symbole : le Newton MessagePad, lancé en 1993. Le Newton rêvait déjà d'un futur "post-PC" dans la poche : notes, agenda, e-mail, synchronisation, reconnaissance d'écriture, applications installables. Pourtant, le produit n'a jamais trouvé son marché et Apple met fin au développement de Newton OS en 1998. À l'inverse, quand Apple lance l'iPad en 2010, le concept de l'ordinateur personnel tactile devient instantanément mainstream et l'iPad s'impose comme le nouveau standard.

Comparer Newton et iPad, ce n'est pas seulement opposer un échec à un succès. C'est surtout observer comment le bon produit, au bon moment, s'appuie sur un écosystème logiciel et des pratiques de développement qui changent tout.

Newton : une vision mobile avant l'heure

Le Newton MessagePad apparaît à une période où le grand public n'a pas encore "appris" la mobilité. Les usages dominants restent le PC de bureau, le téléphone est encore largement "bête", et Internet n'est pas une évidence quotidienne. Le Newton tente néanmoins de proposer une expérience radicalement nouvelle : capturer de l'information à la volée, au stylet, puis la retrouver, la classer, la synchroniser. L'interface est pensée pour la prise de notes, l'agenda, le carnet d'adresses et des apps dédiées.

Sur le plan technique, Apple mise sur un système d'exploitation spécialisé (Newton OS) et sur une reconnaissance d'écriture qui doit devenir le cœur de l'expérience. Or, les limitations matérielles de l'époque pèsent lourd : ressources rares, CPU et mémoire contraints, stockage restreint, autonomie et performances fragiles. Résultat : un produit visionnaire, mais perçu comme lent, cher, et trop imparfait sur sa promesse la plus spectaculaire. La culture pop s'en mêle : une technologie "qui fait rire" est très difficile à rattraper.

Autre frein majeur à l'adoption : le Newton arrivait avec un tarif très élevé pour l'époque, ce qui le plaçait face à une attente de "magie" immédiate — difficile à tenir quand la technologie est encore en phase de maturation.

Et pourtant, l'intuition était la bonne. Les principes qui feront le succès de l'iPhone puis de l'iPad sont déjà là : l'objet personnel, toujours avec soi, centré sur les données de l'utilisateur, avec une interface tactile (au stylet) et un catalogue d'applications.

Programmer sur Newton : NewtonScript et Newton Toolkit

Pour les développeurs, le Newton est fascinant parce que son modèle logiciel n'était pas un simple "mini-Mac". Apple a conçu une plateforme mobile avec son propre langage, ses



propres outils, et une philosophie adaptée aux contraintes de l'embarqué.

Au départ, le langage Dylan (pour DYnamic LANguage) avait été envisagé comme langage principal du Newton. Inspiré de Lisp et développé par Apple au début des années 90, il combinait typage dynamique, fonctions génériques et multi-dispatch avec la possibilité de compilation vers du code natif performant.

Finalement jugé trop immature et trop coûteux en ressources, il fut remplacé par NewtonScript, conçu spécifiquement pour les contraintes mémoire extrêmes du MessagePad.

Les avantages de NewtonScript : un langage orienté objet… mais basé sur des prototypes, influencé notamment par Self. Au lieu de classes et d'instances au sens classique, on manipule des objets qui héritent d'autres objets par délégation. Cette approche a un intérêt majeur sur une machine contrainte : on peut partager beaucoup de comportements et de valeurs par "différence" plutôt que de dupliquer. Dans l'esprit, c'est une optimisation par conception, pas une optimisation "à la fin".

Un détail rarement cité, mais très "systems" : NewtonScript s'appuyait sur une approche dite de "differential inheritance" (héritage différentiel). Plutôt que de dupliquer des structures complètes, un objet ne stocke que ses différences par rapport à son prototype ; le reste est résolu par délégation. Sur une machine aux ressources limitées, c'était une façon élégante — et franchement révolutionnaire pour l'époque — d'économiser de la mémoire tout en gardant un modèle objet très flexible.

Le développement se faisait via le Newton Toolkit (NTK), un environnement intégré tournant sur Mac (et à l'époque, dans des conditions minimales qui paraissent aujourd'hui surréalistes). Typiquement, on conçoit l'interface, on écrit le code, on génère un paquet applicatif, puis on transfère le tout sur le MessagePad via un câble série (souvent un simple montage type "null modem"). Ensuite seulement, on testait sur l'appareil.

Ce workflow impose une discipline très différente de celle à laquelle les développeurs iOS sont habitués :

- Boucles de feedback plus lentes : chaque installation sur device est "coûteuse" en temps.
- Débogage moins confortable : on doit raisonner davantage, tracer, reproduire, plutôt que s'appuyer sur des outils de profiling et d'inspection en continu.
- Optimisation omniprésente : mémoire et performance ne sont pas un sujet de "polish", mais une contrainte initiale.

Pour illustrer la différence de paradigme, voici un exemple minimaliste (pseudo NewtonScript, volontairement simplifié) :

```
bouton := { _proto : buttonProto, viewBounds: {left:10, top :10, right:110,
bottom:40},
      text: "OK",
      DoCommand : func() begin
        Notify("Hello Newton!");
      end };
```

Le code ressemble à la description d'un objet : on part d'un prototype (buttonProto), on surcharge quelques "slots" (propriétés), puis on attache du comportement. C'est une manière de programmer très orientée "composition d'objets UI", pensée pour des interfaces manipulées au stylet.

Autre différence cruciale : la distribution. Dans les années 90, pas de magasin d'applications centralisé. Les logiciels circulent via CD, disquettes, BBS ou téléchargements, parfois via des communautés très spécialisées. La découvrabilité est faible, la mise à jour difficile, et la monétisation incertaine. Pour le développeur, cela signifie peu d'effet réseau : même une excellente application peut rester invisible.

iPad (2010) : la même ambition, portée par un écosystème mature

Quand Apple présente l'iPad en avril 2010, la situation est presque inversée. Le monde a "appris" le tactile avec l'iPhone (2007), les smartphones ont habitué le public à l'ordinateur de poche, les réseaux Wi Fi sont devenus courants, et l'App Store (2008) a déjà prouvé qu'un marché massif d'applications est possible.

L'iPad arrive aussi avec un SDK pensé pour industrialiser le développement : outils modernes, simulateur, API cohérentes, et surtout un canal unique de distribution. Apple communique dès le lancement sur un SDK iPad avec simulateur pour tester et déboguer sur Mac, et la possibilité de créer des applications universelles (iPhone + iPad) en mutualisant le code.

Le succès de l'iPad ne tient pas uniquement à un meilleur matériel. Il tient au contexte :

- Un marché prêt : le tactile n'est plus exotique.
- Un usage évident : navigation web confortable, vidéo, lecture, jeux, productivité légère.
- Un écosystème : applications, accessoires, contenus, et une plateforme de paiement intégrée.

Développer pour iPad : SDK, simulateur, App Store et industrialisation

Du point de vue du développeur, l'iPad prolonge l'iPhone, mais change la grammaire des interfaces. On passe d'un écran "téléphone" à un écran de consultation et de création, où les patterns d'UI (split view, popovers, multi colonnes) deviennent pertinents. Côté technique, on développe avec Xcode, les frameworks Cocoa Touch, UIKit, et à l'époque principalement en Objective C (Swift n'arrivera que plus tard).

Le workflow est lui aussi un saut générationnel :

- Simulator intégré : on itère vite, on automatise les tests, on reproduit.
- Signature de code, provisioning, sandbox : des contraintes, mais aussi une sécurité et une cohérence de déploiement.
- Instruments et outils de profiling : on mesure la mémoire, le CPU, l'énergie, les fuites, les allocations.
- Distribution centralisée : l'App Store apporte visibilité, mises à jour, paiement, et un modèle économique clair.

En pratique, cela change la nature même des applications possibles. Sur Newton, on vise souvent des outils très ciblés, minimalistes, optimisés pour une tâche (notes, formulaires, utilitaires). Sur iPad, on peut viser des applications riches : édition, création, jeux complexes, lecture interactive, ou apps métier à grande échelle.

Un exemple de contraste : la gestion de l'interface. Là où NewtonScript décrivait l'UI comme un ensemble d'objets prototypaux, UIKit propose une hiérarchie de vues, des contrôleurs, des cycles de vie et des conventions strictes. Le code iOS moderne structure aussi davantage les responsabilités (MVC, MVVM, etc.), et s'appuie sur des outils de refactor et de build automatisés.

Pourquoi le Newton a échoué… et l'iPad s'est imposé comme référence

Le Newton n'a pas échoué parce qu'il était "mauvais" : il a échoué parce qu'il demandait au monde d'être prêt avant lui. Il fallait des usages mobiles, des réseaux, des prix accessibles, et un écosystème d'applications plus large. Il fallait aussi que la promesse phare (la reconnaissance d'écriture) soit suffisamment fiable pour devenir naturelle.

L'iPad, lui, capitalise sur trois acquis :

1. Un public éduqué au tactile via l'iPhone.

2. Un modèle de distribution et de monétisation déjà installé via l'App Store.
3. Une base technique (iPhone OS / iOS) éprouvée, et des outils de développement industrialisés.

On pourrait résumer ainsi : le Newton était un produit "plateforme" sans marché, alors que l'iPad est un marché porté par une plateforme. Le Newton invente l'idée ; l'iPad la rend évidente.

Comparatif concret : ce que le développeur gagne (et perds) entre 1993 et 2010

Si l'on se place "dans la peau" d'un développeur, la différence la plus frappante n'est pas le langage : c'est la vitesse d'itération et la capacité à livrer.

Sur Newton, le chemin entre une idée et un utilisateur est long : fabriquer ou trouver la distribution, documenter l'installation, gérer les versions, et surtout composer avec un parc installé réduit. La contrainte de marché devient une contrainte technique : on a moins de feedback, donc on corrige moins vite.

Sur iPad, la chaîne est intégrée : on développe, on teste (simulateur + appareil), on pousse une mise à jour, et on obtient immédiatement des métriques (crash reports, retours, notes, téléchargements). Cela encourage un style de développement agile avant l'heure : livrer petit, mesurer, améliorer.

Autre différence : l'interface utilisateur. Newton est pensé pour le stylet et l'écriture. Le développeur doit composer avec une "saisie naturelle" encore instable. Sur iPad, le multi touch devient une certitude : scroll inertiel, zoom, gestes. Les API et les guidelines encouragent la fluidité, et le matériel suit. En clair : la plateforme iPad réduit le risque UX pour le développeur, parce qu'une grande partie de l'expérience (gestes, animations, latence acceptable) est garantie par le système.

Enfin, la sécurité et la compatibilité. iOS impose une sandbox, une signature et des règles strictes. C'est parfois frustrant, mais cela apporte aussi une prévisibilité : une app "bien construite" a de bonnes chances de tourner de manière stable sur un large parc. À l'époque Newton, l'hétérogénéité et la maturité des outils rendent ce type de promesse plus difficile.

Fiction : si le Newton sortait avec l'App Store

Imaginons le Newton avec un magasin d'applications, une synchronisation simple, et des mises à jour instantanées. Son principal handicap (une promesse imparfaite au lancement) aurait pu être atténué : on corrige, on améliore la reconnaissance, on pousse des versions, on suit les crashs, on instrumente. Autrement dit : on traite la plateforme comme un produit vivant.

C'est exactement ce que l'iPad a permis à Apple et aux développeurs tiers : apprendre en production, à grande échelle, avec une boucle de retour continue. C'est aussi pourquoi l'iPad a pu "s'imposer" face à la concurrence : le matériel compte, mais la vitesse d'évolution de l'écosystème compte encore plus.

Zoom code : NewtonScript vs UIKit, deux générations

Comparer NewtonScript et UIKit, c'est comparer deux manières de "déclarer" une interface. Sur iPad (2010), un bouton en UIKit ressemble davantage à ceci (extrait Objective C, simplifié) :

```
UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];
[button setTitle:@"OK" forState:UIControlStateNormal];
[button addTarget:self action :@selector(tapOK) forControlEvents:UIControl
EventTouchUpInside];
```

On voit apparaître des conventions industrielles : typage, *cible/action*, gestion du cycle de vie, outils de compilation, warnings, analyse statique. Le Newton, lui, visait la flexibilité et la compacité. L'iPad vise la robustesse et la standardisation. Deux philosophies qui se comprennent : l'une naît dans l'incertitude d'un marché à inventer, l'autre dans la nécessité de scaler à des millions d'appareils.

Conclusion : être en avance, c'est aussi être seul

Le Newton a posé des jalons : informatique personnelle mobile, assistant numérique, saisie naturelle, applications. Mais il manquait d'alliés : un réseau, un store, un marché mûr, et des outils suffisamment confortables pour industrialiser. L'iPad, lui, arrive quand l'industrie a déjà accepté l'idée d'un écran tactile personnel, et quand Apple dispose d'un socle logiciel et d'un modèle économique prêts.

Et il y a un clin d'œil moderne : avec l'iPad Pro et l'Apple Pencil, Apple réhabilite l'idée du "stylet", non plus comme palliatif à un écran peu précis, mais comme outil de création. Notes manuscrites, annotation, dessin : ce que le Newton promettait autour de l'écriture trouve enfin, des années plus tard, un matériel et un écosystème capables de le rendre naturel et fiable.

Pour les développeurs, la leçon est double. D'un côté, le Newton rappelle qu'un bon modèle de programmation (NewtonScript et sa logique prototype) ne suffit pas si l'écosystème ne suit pas. De l'autre, l'iPad montre qu'une plateforme réussie, c'est une "expérience complète" : API, outils, distribution, feedback, et règles du jeu stables. Et souvent, l'avenir se joue autant dans l'outillage que dans l'idée.

Références

Communiqué Apple : arrêt du développement Newton OS (27 février 1998).

Communiqué Apple : lancement iPad (27 janvier 2010) et SDK/simulateur.

Ressources Newton : Newton Toolkit (NTK) et tutoriels NewtonScript (MessagePad.org).

Article historique : Developing for the MessagePad (TidBITS, 13 septembre 1993).

Documentation / analyses : NewtonScript et approche prototype-based (NewtonScript.org).

LES PROCHAINS NUMÉROS À PARAITRE

PROGRAMMEZ! N°274	PROGRAMMEZ! N°275
3 avril 2026	29 mai 2026



Patrick-Le-Geek

Geek et explorateur des technologies, disciple de JOBS, j'aime butiner de technologie en technologie !

L'intelligence visuelle, le futur d'Apple ?

À l'aube de ses cinquante ans, quelles vont être les futures orientations technologiques d'Apple ? Nous l'attendions sur le côté IA, appuyé par le hardware, mais c'est plus probablement du côté interface homme-machine que les choses vont changer. Projetons-nous dans un futur peut-être pas si lointain.

Encore une fois, c'est le hardware qui va piloter ce changement et l'IA qui en sera le cerveau. N'oublions pas qu'Apple a la maîtrise totale du hardware. Il a aussi maintenant à sa disposition une arme redoutable, un LLM de premier plan, Gemini.

En s'appuyant sur la réutilisation de technologies, le côté "frugal" de la firme de Cupertino et grâce aux avancées de l'IA, Apple a probablement trouvé le chemin vers le succès.

On peut imaginer que dans ce domaine, Apple détient un savoir unique, riche de son expérience Face ID et de l'Apple Vision Pro, et peut-être même du projet Titan de voiture d'Apple.

Et si l'Apple Vision Pro était le MVP Apple ?

L'Apple Vision Pro, avant sa sortie, avait des ambitions bien plus importantes qu'aujourd'hui. Entre autres, toute une partie "cognitive" qui finalement n'a pas été développée, sur laquelle un certain nombre d'ingénieurs Apple ont travaillé, donc "S. Crispin" pour permettre d'analyser le comportement d'une personne en face du Vision Pro (Persona, qui permet aux autres de voir une représentation dynamique et naturelle de vous) ainsi qu'une technologie de "Biofeedback" sur laquelle je vais revenir.

Mais, je pense qu'Apple a l'ambition d'aller beaucoup plus loin et d'enrichir l'interface utilisateur homme-machine, l'interaction homme-machine de ses futurs produits.

A la souris, la roue crantée, le doigt et le Pencil, il est probable que nous assistions à l'arrivée d'une nouvelle interface multidimensionnelle.

Imaginez un dispositif à l'image de Face ID, un maillage virtuel 3D composé de milliers de points autour de vos mains, de l'ensemble de votre visage et une projection dans l'environnement proche.

La technologie derrière Face ID deviendra un élément essentiel de cette architecture.

La caméra jouera un rôle stratégique, bien au-delà de la simple vision d'images, ou de la prise de photos et de vidéos. Ce sera "l'œil" de cette technologie.

Aujourd'hui, vous entendez beaucoup parler des caméras algorithmiques (vidéosurveillance algorithmique, à la "Minority Report") et des possibilités fantastiques qu'elles ont. Mais cette technologie va aller bien au-delà de ce qu'il est possible de faire avec la reconnaissance d'images et de vidéos.



L'usage de ces nouvelles technologies pourrait être en rapport avec les dernières acquisitions de la firme, Apple pourrait aussi introduire un concept de communication non verbale. Un élément de plus dans l'arsenal user interface d'Apple.

La firme de Cupertino pourrait intercepter toutes les micro-expressions du visage, mais il y encore plus intéressant ! Si l'on reprend les travaux menés par S. Crispin*, "Neurotechnology Prototyping Researcher in the Technology Development Group" au sein d'Apple lors du développement de l'Apple Vision Pro, on a là une vision complète de ce que pourrait être le futur d'Apple.

De plus, Imaginez une interface dans le monde réel (une interface de communication) qui vous permette de communiquer avec l'environnement, les objets qui vous entourent.

Une interface avec votre visage, vos mains et vos expressions

Le "device Vision d'Apple" (attention, ce n'est pas un Apple Vision Pro bis !) et être en mesure de déclencher des actions automatiquement, piloter et interagir avec des objets qui vous entourent, mais aussi l'environnement.

Une nouvelle interface cognitive au service des utilisateurs. "Le futur est visuel", les usages sont très larges et ils rayonnent dans tous les domaines dans lesquels Apple évolue : la Santé, le bien-être, le Sport, le "Home Automation" (Maison), la Robotique, etc.

Imaginez cette dimension sensorielle, visuelle et acoustique dans un nouveau device.

Apple a compris que l'iPhone ne peut pas être l'avenir. Il doit laisser sa place, ou tout au moins partager le futur avec un nouveau device à la pointe de la technologie cognitive.

Il est probable que cette technologie et déjà dans les cartons d'Apple. Il est aussi probable que cette technologie soit déjà dans un état très avancé dans les laboratoires de la firme à la pomme.

Apple détient des technologies de Machine Learning (ML) très avancées dans le domaine de la vision.

Mais allons encore plus loin. On peut tout à fait imaginer qu'après cette technologie de maillage virtuel 3D, on puisse mettre une surcouche virtuelle pour enrichir cette faculté de vision. A l'image d'un device comme le Vision Pro.

Au lieu que l'utilisateur soit derrière les verres d'un device, ce sera l'intelligence artificielle et le hardware qui seront aux commandes de cette surcouche virtuelle.

Et enfin, pour conclure, quelle forme pourrait prendre ce device ? Un dispositif ou plusieurs, je ne sais pas exactement sous quelle forme, peut-être la reprise de devices existants dans la gamme Apple avec ses évolutions technologiques, ou bien une toute nouvelle catégorie de device.

Apple a-t-elle toujours les ingénieurs ou a-t-elle besoin de reconstituer des équipes pour cela ? Peut-être est-ce la raison des dernières acquisitions de la firme.

Apple en a-t-elle les moyens ? La réponse est oui.

Apple a-t-il les capacités opérationnelles de pouvoir le faire ? La technologie est-elle suffisamment avancée pour partir en production ? Là est encore toute la question. Il faudra attendre encore certainement un certain temps avant d'avoir la réponse.

Mais la perspective offerte et les possibilités sont fascinantes.

*la lecture de sa publication X (Twitter) @sterlingcrispin est absolument passionnante. Il a aussi participé à un podcast (replay sur YouTube.)

Les partenaires 2026 de

programmez!

Le magazine des dév - CTO & Tech Lead



Vous voulez soutenir activement Programmez! ?
Devenir partenaires de nos dossiers en ligne et de nos événements ?

Contactez-nous dès maintenant :

ftonic@programmez.com

Boostez votre carrière avec

PYTHON

Livres et e-formations pour réussir vos projets Data Science et IA













eni

www.eni-elearning.com

Découvrez nos e-formations