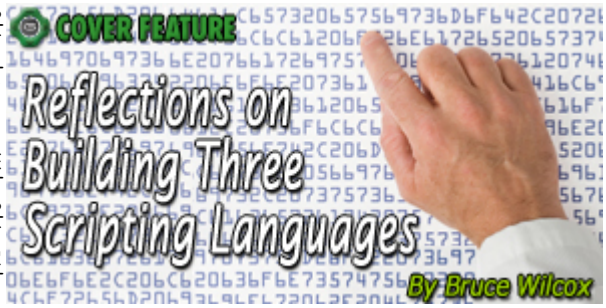
※ 본 아티클은 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

Gamasutra.com

세 가지 스크립트 언어 제작에 대해 돌이켜보기

Bruce Wilcox
2007년 4월 12일

http://www.gamasutra.com/features/20070412/wilcox_01.shtml



게임 업계에서 나의 역할은 "원래" AI 전문가였다. 하지만 지난 2000 년 이후로 3 곳의 회사에서 3 가지의 스크립트 언어를 만들었다. 본 아티클은 그런 나의 경험을 바탕으로 작성됐다.

2000 년에 나는 3DO 에서 전통적 게임 스크립트 언어인 ICE 를 작성했다. ICE 의 주요 특징으로는 게임 프로그래밍 지원 내장(이벤트, 타이머, 스크립트 멀티 태스킹, 저장 & 복구), 혼합 컴파일(intermix compile), 인터프리트된 스크립트의 동적 로딩이 있었다.

2005 년에는 Radical Entertainment 에서 HIPE 를 작성했다. HIPE 는 차세대 게임의 인공지능에 특화된 실시간 계층구조 태스크 네트워크 관리 언어였다.

또 2006 년에는 LimeLife 에서 FLIRT 를 작성했다. 휴대전화의 화면크기도 서로 다르고 운영체제에 문제가 있는 경우도 있으며 어떤 휴대전화는 J2ME 를 사용하는 반면에 어떤 것들은 BREW 를 사용하는 바람에 응용 프로그램을 휴대전화로 포팅하는 작업은 가히 악몽에 가까웠다. 하지만 FLIRT 를 사용하면 응용 프로그램을 단 한 번만 작성하고 추가적인 포팅 작업은 거의 없이 모든 플랫폼에서 실행할 수 있었다.

제작할 것인가, 구매할 것인가, API를 사용할 것인가?

기존에 사용하던 프로그래밍 언어의 공통 라이브러리 루틴에 접근할 수 있는 API 를 제작할 것인가, 이미 시장에 출시된 소프트웨어를 구입(아니면 공개 소프트웨어를 사용)할 것인가, 그것도 아니라면 자신만의 언어를 개발할 것인가에 대한 질문은 항상 나오기 마련이다.

API 개발은 당신이 프로그래머(코딩에 대한 두려움이 없는)이고 라이브러리 루틴이 코드에서 상대적으로 독립된 부분일 경우에 적합하다. 하지만 프로그래머가 아닌 스크립터이거나 완전히 새로운 런타임 시스템 모델이 필요할 경우에는 적합하지 않다.

스크립트 언어를 사용할 경우

1. 스크립트를 사용하면 더 적은 코드를 사용해 비전문가라 하더라도 게임의 구조를 표현할 수 있다. 빠른 개발(Rapid Development)은 보통 기존 언어와 함께 사용할 수 있는 스크립트 언어를 사용하는 주된 이유이며 스크립트 언어는 보통 더 적게 명령을 입력할 수 있도록 단순화된 구문(Syntax)을 가지고 있다.
2. 스크립트를 사용하면 실제 코드보다 스크립터의 코드가 훨씬 더 안전하게 작동할 수 있도록 가상 머신(Virtual Machine)을 만들 수 있다. 가상 머신을 사용하면 엔진의 복잡성이 줄어들며 이미 디버깅이 끝난 코드를 여러 프로젝트에서 공유할 수 있게 된다. 또 가상 머신을 이용할 때는 보통 메모리 관리가 자동화되므로 스크립터가 그런 골치 아픈 부분을 신경 쓰지 않아도 된다.
3. 스크립트는 흔히 동적 로딩(hot-loaded)이 가능하므로 컴파일, 로드, 디버그 주기를 빠르게 할 수 있다.

자신만의 언어를 제작하지 않을 경우

기존 언어들은 생산성을 빠르게 끌어올릴 수 있고, 관리 필요성이 좀 더 적으며 잘 정리된 문서를 가지고 있다. 게다가 흔히 아주 중요한 라이브러리들이 포함되어 있는 경우가 많다.

스크립트 언어를 작성하는 것은 상당한 부담이 따르는 일이다. 그러므로 관리자 입장에서는 당연히 피하려고 할 것이다. 스크립트 언어를 작성하려면 번역기(Translator), 런타임 시스템(Runtime System), 문서(Documentation), 회귀 테스트 코드(Regression Test Code), 빌드 스크립트(Build Script) 등을 모두 작성해야만 한다. 언어를 사용할 사람들의 교육도 해야 한다. 게다가 작업을 편리하게 하기 위해 도구(Tool)들도 만들어야 한다. 그것도 꽤 많이 만들어야 한다. 덤으로 일반적인 코드보다 디버그 스크립트 코드를 작성하는데 시간이 더 오래 걸린다면 빠른 코딩이라는 이점은 사라지게 되어 버린다.

다음은 내가 작성한 3 가지 시스템의 크기 비교(코드 줄 단위)이다.

	Grammar	Translator	Runtime System	Documentation
ICE	1,102	12,385 C	9,731 C	7373
HIPE	1,310	12,608 C	11,798 C	8422
FLIRT-J2ME	1,162	3,815 C	15,175 Java	8435

FLIRT 의 런타임 시스템이 가장 큰 이유는 스크립트와 화면 레이아웃용 기능이 모두 포함되어있기 때문이다. 하지만 번역기는 가장 작은데, 이것은 FLIRT 가 처음부터 번역기가 없는 어셈블리(Assembly) 언어에 가까운 형태로 제작되었으므로 번역기가 매우 단순하기 때문이다.(이 번역기는 휴대전화 에뮬레이터에서 실행될 때 런타임에서 수많은 인증을 처리한다.) FLIRT 를 BREW 에서 실행하려면 다른 런타임 시스템이 필요하다.(표에는 표시되지 않았음.)

위의 시스템들에 대한 문서가 비슷한 크기를 가지고 있지는 않은데, 그 이유는 스크립트 자체로도 이해가 가능하거나 단지 내가 어느 시점에서 지칠 대로 지쳐버렸기 때문이다.

HIPE 는 하청을 받아서 제작한 프로젝트였으므로 제작에 5 개월 가까이 소요됐다. ICE 와 FLIRT 는 둘 다 내가 회사에 근무하면서 제작한 것이고 다른 업무를 하면서 진행했으므로 제작에 얼마나 걸렸는지는 정확하게 알 수 없지만 아마도 FLIRT 의 J2ME 버전을 제작하는데 4~5 개월 정도 걸렸던 것으로 생각한다.

언어 1: ICE

2000 년에 3DO(현재는 사라졌음)는 스크립트 언어의 요소 중에 하나인 멀티 플랫폼 공통 라이브러리 개발을 시작했다. 3DO 는 각 프로젝트마다 서로 다른 언어를 디자인하고 제작한 후에 레벨 디자이너를 교육하는 일에 지쳐있었다.

우리 회사의 프로젝트(PS2 용 *그린 로그[Green Rogue]*)가 시작됐을 때도 당연히 3DO 의 스크립트 언어를 포함한 공통 라이브러리 코드를 사용하라는 새로운 3DO 의 정책을 지키기 위해 디자인 문서(Design Doc.)를 작성했다. 그리고 스크립트 언어의 문서를 읽었다. 아니, 읽으려고 했었다.

그 새로운 스크립트 언어는 실제로는 C 언어용 매크로 전처리기에 불과했다. 난 그 문서를 어떻게든 이해해보려고 했지만 어떤 레벨 디자이너도 그 문서를 읽고 이해할 수 있는 사람은 없으리라는 것을 알고 있었다. 게임용 스크립트 언어라면 이벤트 처리 메커니즘과 타이머를 제공하는 것이 당연했지만 그런 것은 하나도 없었다. 이 말은 곧, 각 프로젝트마다 자신들만의 이벤트 처리 메커니즘과 타이머 기능을 하는 매크로를 새로 만들어야 한다는 뜻이었다. 그리고 실제로도 각 프로젝트는 최소한의 공통 코어(Common Core)를 제외하고는 완전히 다른 스크립트 언어를 사용했다.

마지막으로 스크립트 언어 전체가 C 의 전처리기였다. 그래서 스크립트를 PS2 용 로드 파일(Load File)로 컴파일 해야만 했다. 혹시라도 스크립트를 변경하려면 다시 컴파일 한 후에 새로운 ELF 파일로 다시 다운로드하는 수밖에 없었다.

그때만 하더라도 PS2 DevStation 의 이더넷 연결이 “미치도록” 느렸다. 그래서 디자이너가 스크립트를 변경하는 것은 끔찍할 수밖에 없었다. 그러니 서로의 요구사항들이 충돌할 수밖에 없었는데, 팀원의 일부는 최대한 빨리 스크립트가 컴파일 되기를 바랐다. 그리고 나머지 대다수는 게임 제작을 빠르게 하기 위해 게임 플레이 도중에 인터프리트된 스크립트를 동적으로 불러올 수 있기를 바랐다.

제작할 것인가, 구매할 것인가, API를 사용할 것인가?

우리는 프로그래머가 아닌 레벨 디자이너들을 위한 스크립트 언어가 필요했으므로 API 는 제외시켰다. 많은 사람들이 기존에 이미 제작이 되어있는 스크립트 언어를 사용하자고 제안했지만 그것들을 게임과 통합하는 데는 좀 문제가 있었다. 기존의 모든 스크립트 언어가 일반적인 목적의 프로그래밍 언어로 제작되었기 때문이다. 그러므로 스크립터가 게임에 문제가 될 수도 있는 코드를 작성할 수도 있다는 뜻이었다.

우리가 알고 있던 복잡한 스크립트들로는 절대로 스크립트의 모든 실행 경로를 테스트할 수 없었다. 그러므로 우리는 최대한 게임에 문제가 생기지 않도록 스크립터가 할 수 있는 일을 통제할 수 있어야 했다. 스크립트가 “아무런 문제도 일으키지 않는다는” 것을 보장하기 위해서 말이다.

기존 스크립트 언어들은 또 각각의 고유한 약점들을 가지고 있었다. Lua 를 예로 들면 가비지 컬렉터(Garbage Collector)가 내장되어 있다. 하지만 이 기능은 실시간 게임 도중에 실행하기에는 좀 부적합하다. Python 은 엄격하게 문법을 지켜서 인터프리트되는데(Strictly Interpreted) 이것은 몇몇 스크립트에서는 너무 느려서 부적합하다. 그리고 이 둘 중 어떤 언어도 컴파일 되고 인터프리트된 스크립트를 동적으로 불러올 수 있어야 한다는 우리의 주된 요구를 만족시키지 못했다. 이 언어들은 지속적으로 발전되어왔으므로 지금이라면야 이 둘 중에 하나를 사용하겠지만 그때 당시만 하더라도 두 언어 모두 부적합했다.

그래서 나는 두 페이지에 걸쳐 라이브러리 스크립트 언어의 문제점을 상세하게 기술한 문서를 작성했고 또 다른 스크립트 언어 제작에 대한 승인을 받아냈다. 스크립트 언어 제작 요청을 하기 전에 반달 정도를 소비해 ICE 의 기본적인 컨셉을 증명할 수 있는 프로토타입 시스템을 제작한 것이 승인을 받는데 상당한 도움이 됐다. 하지만 나의 목표는 단순히 우리 프로젝트만을 위한 스크립트 언어를 작성하는 것 이상의 것이었다.

스크립트 언어를 제작하는 것과 같은 노력은 다른 프로젝트에서도 사용될 수 있어야만 보상받을 수 있기 때문이다. 그래서 나는 3DO 가 원래 의도했던 것을 하기 위해 기존의 라이브러리 언어를 나의 스크립트 언어로 대체하고자 했다.

많은 사람들이 자신들이 근무하는 회사에서 사용할 목적으로 스크립트 언어를 제작했지만 하나, 심지어 한 개의 프로젝트에서도 사용되지 않고 사라져갔다.

그러므로 나의 언어가 실제로 보편적으로 사용되려면 다음과 같은 요구사항을 만족시켜야만 했다. a) 더 훌륭한 언어를 제작한다. b) 사용법을 배울 수 있도록 문서화한다. c) 새로운 기능을 필요로 하는 프로젝트에서도 사용할 수 있도록 지원을 해줘야 한다. d) 언어를 변경하고 싶어하는 사람들이 언어에 대해 알 수 있도록 점심 시간이나 다른 방법을 활용해 홍보한다.

위와 같은 사항을 지키지 않는다면 단순히 또 다른 프로젝트에서 사용되는 또 다른 스크립트 언어로 전락할 뿐이었다. 그리고 사람들이 좋아할만한 훌륭한 이름을 붙이는 것도 매우 중요했다.(홍보관련 문제) 그리고 난 이름으로 ICE 를 택했다.

ICE 는 간략화된 C 스타일의 구문과 런타임에서 멀티 태스킹, 이벤트, 타이머, 인터프리트된 코드를 지원했다. 모든 코드들은 스크립터의 어처구니없는 오류들에서 “보호(Protected)”받을 수 있었다. 이것은 공통 라이브러리 코드에 접근할 수 있는 간단한 API 보다 더 진보한 것이었다. 그리고 스크립트 언어에 필요한 기능과 프로그래밍 언어에서 호출할 수 있는 다양한 공용 라이브러리 루틴이 얼마나 가치 있는 것인가를 보여주는 고전적인 예제라고 할 수 있다.

결과적으로 시스템은 (속도와 동적 로딩 중 어떤 것을 원하는 가에 따라) C 소스코드 또는 인터프리트가 가능한 바이트코드를 생성하는 C 번역기로 구성되었다.

ICE 는 인자(Arguments)를 사용하는 이벤트를 지원했고 이 이벤트들은 모두 콜백(Callback) 대신에 비동기 호출(Asynchronous Call)을 사용했다. 시스템은 두 개의 인자를 사용해 패턴 매치(Pattern Match)가 가능한 이벤트를 “받는(Listening)” 것도 지원했으며 나는 이 기능을 콜백과 극도로 느슨한 결합 메커니즘으로 활용하는 것을 선호했다. 일례로 게임 엔진의 애니메이션 시스템이 어떤 캐릭터의 애니메이션 재생을 완료하면 캐릭터와 애니메이션의 이름을 가지고 있는 재생 완료 이벤트를 전체적으로 발송(Broadcast)한다. 그러면

애니메이션을 시작한 코드를 포함한 “어떤” 코드에서든지 해당 이벤트에 반응해 특정 작업을 수행할 수 있었다.

기존의 코드를 전혀 변경하지 않고, 해당 정보를 가지고 다른 작업을 수행하는 독립적인 루틴을 작성할 수도 있었다. 그러므로 특수효과 코드는 차후에 독립적으로 추가할 수 있었다. 이것은 클래스 객체의 가상 함수(Virtual Functions)를 사용하는 것보다 좋은 방법인데, 가상 함수를 사용하면 작성자가 의도한 내용을 알기 위해 차후에 어떤 함수를 호출할 것인가를 계획해야 한다. 그러므로 가상 함수를 사용할 때 차후에 뭔가 다른 것을 추가하려면 기존의 코드로 돌아가 수정을 가해야 하기 때문이다.

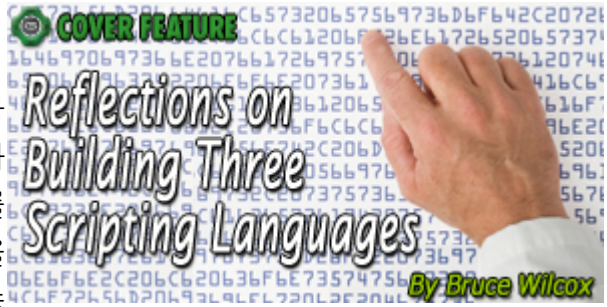
ICE 는 게임 엔진 함수 전체를 사용할 수 있어야 하므로 엔진이 이름, 코드 주소, 인자 설명, 항목, 루틴의 문서를 등록 인터페이스(Registration Interface)를 사용해 ICE 에 자신의 사본을 사용할 수 있도록 등록할 수 있었다. 또한 ICE 에는 스크립터가 ICE 의 게임 엔진 인터페이스에 새로운 기능이 추가되더라도 “항상” 최신의 문서를 사용할 수 있도록 등록된 모든 루틴과 엔진 변수(Engine Variables)를 출력하는 명령을 가지고 있었다.

ICE 는 객체(Objects), 문자열(Strings), 이벤트(Events), 정수(Integers), 실수(Floats), 열거형(enumerations), 배열(Arrays), 레코드(Records)라는 상대적인 자료형을 지원한다. 그리고 위치와 벡터를 직접 지원하기 위해 XY 와 XYZ 라는 내장 레코드 형식을 지원한다. 자료형 검사는 컴파일 때 이루어진다. 객체는 스크립트 상에서 이름으로 사용할 수 있으며 런타임에서 객체에 이름이 할당되는 즉시(게임의 후반부에서 지정될 수도 있음) 객체가 “등록(Registered)”되며 해당 이름을 사용하는 스크립트에서 객체를 사용할 수 있게 된다. 그러므로 스크립트에서는 “Sarge” 또는 “PickupTruck1”과 같이 손쉽게 객체를 참조할 수 있으며 이름과 연결된 객체를 찾는 문제는 신경 쓰지 않아도 된다.

ICE 는 간략화된 C 헤더 파일(Header files)을 직접 읽을 수 있었다. 이로 인해 이전 프로젝트에서 흔히 일어나던 잠재적 오류의 근원인 상수 이름(Constant Names)와 값(Value)을 스크립트 언어에 손으로 옮겨 적는 대신 간단히 헤더 파일을 엔진에서 공유할 수 있게 되었다.

ICE 는 곧 3DO 내부의 스크립트 언어로 지정되었고 실제로 사용되는 것을 본적이 없는 사람을 교육하는데 한 명 정도의 사람이 필요했다.

언어 2: HIPE



Radical 과 나와의 관계는 내가 그쪽에서 면접을 보면서 시작됐다. Radical 에서 일자리를 제안했지만 나는 그 제안을 거절했다. 하지만 그들과는 계속 연락을 하며 지냈고 거의 1 년이 지난 후에 그쪽에서 내게 AI 를 더 정교하게 만들 수 있는 방법에 대한 백서(White Paper)를 작성해달라고 요청했다. 그들의 관심사는 차세대 콘솔의 능력을 최대한 활용할 수 있도록 혁명적인 AI 를 만드는 것에 있었다.

일반적인 게임 엔진은 AI 를 조종할 때 명시적인 스크립트(Explicit Scripting)이나 유한 상태 기계(Finite State Machine, FSM) 중 하나를 사용하거나 두 가지 모두를 사용하는 것이 일반적이다.

스크립트의 단점은 모든 통제권 전환(Control Transition)에 대해 명시적으로 작성을 해줘야 한다는 것이다. 즉, AI 가 X 라는 행동을 취하게 하는 조건이 있고 Y 라는 행동을 취하게 하는 조건이 있다면 모든 경우의 수에 따라 if 와 then 구문을 한 곳에 모두 작성해야 하며 이렇게 되면 길고 끔찍한 코드가 작성되게 된다. 이런 것은 해야 할 행동이 단 한 가지 행동밖에 없다고 가정할 때만 사용하는 것이 좋다. 하지만 상황이 그렇게 단순한 경우는 거의 없다. 간혹 무작정 작성하고 이론적으로 검증한 후에 좋지 않다면 버리고 다른 해결책을 찾아봐야 할 수도 있다. 이런 경우에 스크립트는 적합하지 않다.

FSM 은 특정 행동을 수행하는 사람이 이해 가능한 순차적인 코드 중에 다음에 어떤 코드 조각을 수행할 것인지 명시적으로 검사하는 스크립트를 제작하면서 점차적으로 발전되어 왔다. 이런 코드들로 구성된 FSM 메커니즘을 조직화하며 발전시켜나가면 게임들 간에 서로 공유할 수도 있다. 하지만 여전히 FSM 을 사용하면 이론적인 코드의 경로들을 살펴보기 힘들고 다음에 어떤 코드 조각을 수행할 것인지를 명시적으로 코딩 해줘야 한다. 그리고 FSM 이 점점 복잡해지면 제어 경로가 이리저리 엉켜서 수정, 디버깅하기가 매우 끔찍해진다.

당신이 커맨드 앤 컨커(Command and Conquer)와 같이 서로 협동해 자원을 채취하고 건물, 유닛을 생산하고 수송선을 보호하는 등의 정말로 멋진 AI 를 만들고 싶다면 플래너(Planner)를 작성해야 한다. 이런 AI 는 현재 어떤 것을 가지고 있고 어떤 것을 할 수 있느냐에 기반해 어떤 행동을 취할 것인가를 결정한다.

그래서 나는 다음과 같은 문장으로 시작하는 문서를 작성했다. 본 백서(White Paper)에서는 게임 AI 에 계획, 목표, 조종, 감정, 인격, 기억을 적용하는 것에 대해 다루고 있다. 처음 3 가지 요소의 공통점은 모두 다음에 어떤 행동을 취할 것인가를 결정할 때 필요한 요소라는 것이다. 그리고 각각이 행동 결정에 어느 정도의 영향을 끼친다. 그리고 다른 3 가지 요소는 현재 감각적 유입(Sensory Input)과 실제 능력을 나타낸다.

5 년이나 7 년 후에는 학습형 계획 시스템(Academic Planning Systems)이 굉장히 발전될 것이다. 2 년에 한 번씩 열리는 계획 시합(Planning Competitions)은 1998 년에 5 가지 프로그램이 경쟁하는 것으로 시작되었다. 그 결과로 플래너는 더 크고, 더 복잡한 문제들을 다룰 수 있게 되었다. 본래 이 시합에서는 순수한 “조각 기반(strips-based)” 계획을 다루고 있었다. 그때까지만 해도 그들은 여전히 모든 것이 참(true) 아니면 거짓(false)이라는 사실로만 이뤄진 월드에서 작업했다. 2002 년이 되자 플래너는 시간 기반(time-based) 영역과 숫자 기반(numeric-based) 영역(수치로 표현 가능한 변수의 제약을 받는 영역)까지 다루게 되었다. 그러다 2004 년이 되자 그에 더해서 확률적(probabilistic) 영역까지 포함하는 19 가지의 시스템이 서로 경쟁하게 되었다. 그 시합의 결과 페이지에는 다음과 같은 문구가 적혀있다. “런타임 성능의 관찰 결과는 더 흥미롭고 다양한 결과들이 도출되었다. 당연히 우리는 우리가 완전히 실행이 불가능할 것이라고 생각했던 영역에서 몇몇 플래너들이 보여준 성능은 정말이지 충격적이라고 밖에 할 수가 없다!!”

나는 이런 계획 언어가 AI 를 위한 차세대 고급 언어가 될 것이라고 주장했고 Radical 은 내 의견을 받아들였다.

제작할 것인가, 구매할 것인가, API를 사용할 것인가?

실시간 플래너의 기능은 API 에 적합하지 않으며 높은 결합도가 필요하다.

플래너는 본래 비디오 게임 업계의 요구를 만족시키기 위해 작성된 것이 아니다. 플래너는 학습 목적으로 만들어진 것이며 LISP 로 작성되었고 난해한 문법을 가지고 있는 경우가 많다. 플래너는 모든 행동을 통제할 수 있고 결과를 완벽하게 예측할 수 있는 독립된 환경에서 수행된다. 그리고 플래너는 메모리 사용량 문제에 대해 신경 쓰지 않으며 자신이 필요로 하는 시간을 모두 사용하고 게임 엔진에 자연스럽게 통합될 준비도 되어 있지 않다. 그러므로 구입하는 것은 논할 가치도 없다.

그래서 Radical 은 나에게 하나 디자인해달라고 부탁했다. 그 후에는 나에게 그것을 제작해달라고 했다. 바로 HIPE 를 말이다. HIPE 라는 이름 역시 Radical 에서 마음에 들어 할만한 이름을 선택한 것이다.

HIPE 는 비디오 게임을 위해 주문 제작된 정렬 계층구조 태스크 네트워크(Hierarchal Task Network, HTN) 플래너였다. 완전히 주문 제작된 HTN 플래너인 관계로 백트래킹(Backtracking)에 효율적이며 기존의 월드 상태(World State)를 활용할 수 있었다. 게다가 비디오 게임용으로 제작되었으므로 HIPE 는 플래너에서 계획이 완료되기 전에도 원하기만 한다면 즉시 실제 월드 행동을 취하도록 할 수 있었다.

HIPE 는 내가 제작한 첫 번째 플래너는 아니었다. 나는 이전에도 3DO 에서 우주 정복 형식의 게임을 제작할 때 한 번 제작한 적이 있었다. 그 플래너는 특정 클래스의 함선을 가지고 싶다면 새 함선을 건조하거나, 교환하거나, 훔칠 수 있었고 새 함선 건조를 위한 다양한 자원을 채취, 무역, 수거, 도둑질 등의 방법으로 얻을 수 있었다. 그리고 AI 는 어떤 클래스의 함선이 절대적으로 필요한 상황이라면 적의 함선을 훔치거나, 수거하거나, 새로운 함선을 건조하기도 했다. 하지만 그 플래너는 미래를 생각하고 제작한 것이 아니었을 뿐더러 턴 기반(turn-based) 게임용으로 디자인한 것이었다. HIPE 는 그것보다 더 유연한 구조를 가지고 있다.

완성된 계획 명세서(Plan Specification)는 *모델(Model)*, *논리 구조(Logic)*, *문제(Problem)*라는 3 가지 부분으로 구성되어 있었다.

*모델*에는 월드가 어떤 엔티티(Entities)로 구성되고, 어떤 행동을 취할 수 있으며, 특정 행동이 어떤 효과를 나타내는 지와 같은 월드에 대한 정의(Definition)가 포함된다. 그러므로 모델은 어떤 행동을 *할 수* 있는지에 대한 공통적인 기반을 나타낸다고 할 수 있다. 모델에서는 행동을 어떤 상황에서 수행하는지, 어떻게 해야 하는지에 대해서는 기술하지 않는다. 모델은 (모든 정보가 플래너의 내부 월드 모델에 정의된) 완전히 자급 자족이 가능한 월드일 수도 있고 (몇몇 정보가 외부 월드에 대한 질의의 결과값이 되는) 외부 월드로 통하는 인터페이스일 수도 있다.

모델은 다음과 같은 것들로 구성된다.

1. 사용자 정의 형식(객체의 항목)
2. 월드에 존재하는 객체들
3. 숫자 상수
4. 게임 엔진이나 계획에 의해 비동기적으로 발생할 수 있는 이벤트들
5. 다른 조건으로 계획을 다시 정렬할 때 사용될 수 있는 우선순위
6. 객체들이 어떻게 표현되는가(객체 속성과 객체 간의 관계)
7. 게임 엔진이나 시뮬레이터에서 사용할 수 있는 외부 함수들
8. 월드에서 지원되며 사용 가능한 행동들
9. 계획에 따라 이벤트에 독립적으로 반응할 수 있는 인터럽트 함수들

*논리 구조*는 모델을 활용해 현재 상황에서 적합한 행동을 판단하는 방법을 정의한다. 논리 구조는 전체가 *계획과 행동(Plans and Acts)*으로 구성된다. (일반적으로 일정 시간 동안 다양한 행동을 취하도록 하는 효과를 나타내기 위해 특정 조건이 우세할 때 어떤 행동을 취하는 가에 대한 설명)

행동(Acts)은 원자 단위 행동(Atomic Actions)을 나타내며 계획(Plans)은 계획과 행동이 합쳐진 것이다.

*문제*는 게임에서 달성해야 하는 특정한 목표, 또는 게임 속 현재 상태와 미래의 상태를 나타낸다.

Radical 에서 처음에 제시한 데모 시나리오는 “동물원 사육사(Zookeeper)” 문제였다. 이 문제에서는 몇 개의 출구와 두 개의

동물 우리가 있고 몇 마리의 닭과 여우가 무작위로 배치되어 있을 때 최대한 많은 닭과 여우가 동물원 안에 남아있도록 하는 문제이다. 동물들은 천천히 자신들에게 가장 가까운 출구로 이동하지만(우리에 갇혀있지 않다면) 닭은 가까운 거리에 여우가 있다면 여우에게서 멀어지는 쪽으로 이동하며 여우가 닭을 일정한 거리에 떨어져있다면 여우는 최단 거리로 닭에게 이동해 닭을 잡아먹는다. 사육사는 동물의 옆으로 이동해 동물을 잡거나 (우리 안으로) 떨어뜨릴 수 있으며 (동물이 말뚝의 밧줄을 갉아먹어서 빠져 나오기 전까지 움직이지 못하도록) 말뚝에 묶어버릴 수도 있다.

그러므로 사육사는 게임 엔진을 통제하고 동물을 보호해 문제를 해결해야만 한다. 그리고 약간의 재미를 위해 HUMAN(인간)은 언제든지 어떤 동물 주변으로 순간 이동할 수 있다고 한다면 사육사의 행동에 갑작스러운 변화가 생겨버리지만 여전히 이 새로운 설정에서 최대한 많은 동물을 보호해야만 한다.

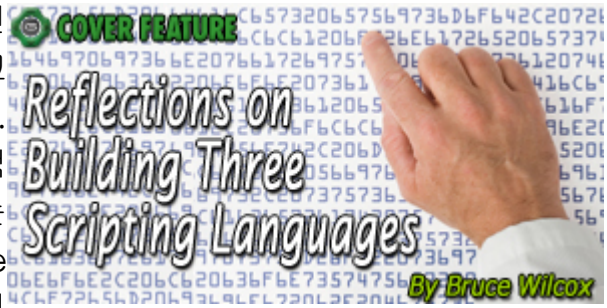
다음은 사육사 문제를 위한 HIPE 스크립트의 대략적인 예제이다.

형식 정의(Type Declarations)는 게임 속의 엔티티의 구성을 정의한다. 엔티티는 구체적인 객체이거나 추상적인 컨셉이 될 수 있다. 형식의 일부로 객체나 다른 형식의 이름을 할당할 수 있다. 그러므로 다음의 코드에서 <chicken> 형식에는 두 마리의 이름이 할당된 닭이 포함된다. <beings> 형식에는 <human> 형식(사육사)과 <animal> 형식(chicken1, chicken2, fox1, fox2)이 포함된다.

TYPE <chicken> : chicken1 chicken2
TYPE <fox>: fox1 fox2
TYPE <animal>: <fox> <chicken>
TYPE <human> : zookeeper
TYPE <beings> : <human> <animal>
TYPE <cage> : cage1 cage2
TYPE <exit> : exit1 exit2
TYPE <holdable> : <animal> nothing
TYPE <state> : free immobile escaped

대부분의 형식은 “대번에 알 수 있는 형식(Obvious)”이다. <holdable> 형식에는 모든 동물과 사육사가 빈손이라는 것을 나타내는 “아무 것도

아닌(nothing)" 객체가 포함된다. <state> 형식은 동물의 상태를 나타내는 이름이 할당된 추상적인 컨셉 형식이다. HIPE 는 또한 런타임에서 요구에 맞춰서 생성할 수 있도록 일정 범위의 객체를 예약해놓는 동적 형식을 선언할 수도 있다. 그러므로 게임에서 현재 2 마리 정도의 닭만 존재하더라도 100 마리의 닭을 위한 공간을 미리 예약해놓을 수도 있다.



일단 형식에 이름을 할당했다면 속성의 이름을 할당하고 객체와의 관계를 기술해야 한다. 속성은 숫자로 구성된 "선택 인자(Selector Arguments)" 또는 단 하나의 "값 인자(Value Argument)"를 최대 5 개까지 가질 수 있는 투플(tuples)이다. 그러므로 속성은 상호 배타적(Mutually Exclusive)이라고 할 수 있다. 예를 들어 차량 형식의 객체는 특정한 시간에 단 하나의 특정 숫자를 가지는 연료량이라는 속성을 가질 수 있다. 관계(Relationships)도 속성과 마찬가지지만 상호 배타적이지는 않다. 인간을 예로 들면, 인간은 사랑이라는 관계를 가질 수 있으며 동시에 여러 명의 사람이나 사물을 사랑할 수 있다.

속성과 관계는 월드 데이터베이스에 저장되는 명제 데이터(Propositional Data)이다. 사용자는 현재 월드와 월드가 시간이 지나면서 어떻게 변해가는지를 묘사하기 위해 이 데이터베이스에 저장된 사실을 확인(Assert 또는 Retract)할 수 있다. 데이터베이스에 저장된 사실 중에 몇몇은 게임 엔진 자체에 저장된 것과 동일하므로 월드가 정상적인 상태인지 확인할 수 있다.

속성은 이름과 인자 목록(Argument List)에 의해 정의된다. 그리고 인자 목록에서 중요한 것은 형식과 인자의 순서뿐이다. 할당된 이름은 오직 사용자를 위한 문서의 역할만을 수행한다.

ATTRIBUTE holding(<holdable> what)
ATTRIBUTE status(<animal> who <state> state)

사육사의 손은 holding 이라는 속성으로 표현된다.(사육사는 한 번에 하나의 동물을 잡거나 빈손으로 있을 수 있다.) 각 동물은 한 번에 하나의 상태만 가질 수 있다.

월드에서 사용할 수 있는 기본적인 행동은 ACT 라고 불리며 다음과 같이 사전에 정의할 수 있다. 하지만 결국에는 행동들에 대한 코드를 작성하거나 게임 엔진 루틴에 연결을 해야만 한다. <OBJECT>는 다른 모든 형식보다 우선시되는 사전 정의된 슈퍼타입(Supertype)이다.

ACT intercept(<OBJECT> who <OBJECT> what)
ACT pickup(<animal>what)
ACT putincage(<cage> where)
ACT Intercept(<OBJECT> where)
ACT stake(<animal> who)

일단 행동을 작성했다면 계획 조각(Plan Pieces)을 작성할 수 있다. 계획이 순서대로 실행되려면 이름, IF 절, ACTS(행동)이나 PLANS(계획) 중 하나 또는 모두를 포함하는 THEN 절을 포함해야 한다.

다음은 어떻게 하면 동물들을 구할 수 있을 지를 자동적으로 판단해서 동물을 구하는 방법에 대한 간략한 예제이다. 계획은 오직 동물에게 다가가 잡은 후에 우리로 가서 우리에 집어넣는 것밖에는 없다. 동물을 말뚝에 묶어놓거나 동물을 주변 땅에 떨어뜨려놓는 것은 고려하지 않았다.

```
PLAN savcanimals()
IF  FINDLOW <animal>(?what)
     {
          Status(what,free) // is what free?
          ObjectDistance(zookeeper,what,?dist)
     }
THEN      saveananimal(what)
          saveanimals()

PLAN SaveAnAnimal(<animal >what)
IF
THEN      Intercept(what)
          HelpAnAnimal (what)

PLAN HelpAnAnimal(<animal> what)
IF        Status(what,free)          // is this attribute true?
          PickCage(what, ?where)
THEN pickup(what)              // pick him up
          Intercept(where)           // goto cage
          Putincage(where)           // drop him into cage
```

계획은 내장 데이터베이스 시스템에 질의를 하는 것을 포함해 반환값(Return Values)을 가지는 함수들을 호출할 수 있다. “?변수이름”을 사용하면 해당 변수에 들어있는 값을 찾고 있다는 것을 나타낸다. FINDLOW 와 같은 함수들은 질의에 알맞은 값을 가지고 있을 지도 모르는 집합들을 반복해서 검색하며 각괄호로 묶여있는 모든 테스트 결과와 AND 연산을 수행한다. 그러므로 SaveAnimals 는 각 동물을 검색해서 동물이 free 인지 아닌지를 데이터베이스에 저장된 Status 속성과 비교해 그에 해당하는 첫 번째 결과를 반환한다. 해당하는 결과가 없다면 다음 인스턴스(Instance)로 넘어간다. 동물이 free 상태라면 FINDLOW 는 사육사에게서 동물이 얼마나 떨어져 있는가를 ?dist 에 저장해서 반환하는 ObjectDistance 함수를 호출한다. 그러므로 FINDLOW 는 가장 가까운 거리(?dist)에 있는 동물(?what)을 반환하는 필터의 역할을 수행하는 셈이다. 또한 FINDLOW 는 백트랙이 가능하므로 계획이 나중에 실패하더라도 다시 시도해서 두 번째로 가까운 동물을 찾는다거나 하는 것이 가능하다.

ICE, HIPE 의 디자인을 살펴보면 이벤트를 지원하고 있으며 이벤트를 발생시키기 위한 이벤트/인자를 받을 때까지 대기하는 것이 가능하다는 것을 알 수 있다.

EVENT Arrived(<OBJECT >from,< OBJECT >to)
Event Escaped(<chicken> who)

동물원 사육사 문제에서 흥미로운 이벤트는 바로 Arrived 이며 Escaped 에서는 별로 할 일이 없다.

다음 코드는 이 예제에서 정의된 행동을 나타내고 있지만 이 코드들이 어떤 일을 수행하는지 상세하게 살펴보는 것은 의미가 없다. 문맥과 주석을 보고 내용을 알 수 있을 것이다.

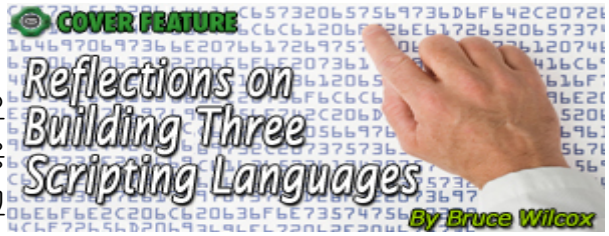
```
ACT intercept(< OBJECT > what)    // get next to this object
PRE
POST  GoIntercept(zookeeper,what)          // asynch engine call
      handle                               // returns:
      {
            Arrived(zookeeper,what) {println("Arrived " ) }
            Escaped(what){ println("*******Lost him") }
      }

ACT   pickup(<chicken>what)                // holding nothing, pick up chicken
PRE   holding(nothing)                     // works ONLY if he is not holding anything
      ObjectDistance(what, Zookeeper,?dist)
      Dist <= PICKUP_RANGE                 // is he in range to pick up
POST  DoPickup(what)                       // tell engine to do a pickup
      holding(what)                        // tell db zookeeper is hold this chicken
      status(what,immobile)                // tell db the new state of this chicken

ACT   putincage(<cage> where)              // drop chicken into adjacent cage
PRE   holding(?what)                       // figure out what zookeeper is holding
      what != nothing                      // make sure he is holding something
      ObjectDistance(where, Zookeeper,?dist)
      Dist <= PICKUP_RANGE                 // in range to drop
POST  DoPutInCage(where)                   // put chicken down in cage
      holding(nothing)                     //  zookeeper is empty
      // chicken was immobile when held and is still immobile when caged
```

기본적인 번역기/런타임 패키지에 하나 더하자면 소스 수준의 디버거(Source-level debugger)가 있다. 혹시라도 소스 수준 디버거가 없다면 인터프리터에서 루프 안에 있는 어떤 것을 디버깅한다는 것은 악몽이 되고 말 것이다.

언어 3: FLIRT



LimeLife 는 처음에는 필요 없다고 생각했지만 응용 프로그램의 포팅을 관리하고 포팅을 제안 받았을 때 손쉽게 사용할 수 있는 기술이 필요했다. 코드를 J2ME(Java)로 작성하고 나중에 다시 Brew(C)로 코딩하는 일은 매우 지루한 작업이고 휴대전화의 다양한 화면 크기로 인해 각각의 화면 크기 별로 코드를 작성하는 노력이 필요한 작업이다. 코드를 두 가지의 다른 언어로 작성해야 할 때는 API 가 그리 좋은 방법이 되지 못한다.

제작할 것인가, 구매할 것인가, API를 사용할 것인가?

몇몇 휴대기기용 응용 프로그램 개발사에서는 표준화된 UI 생성기와 자신들의 코드를 런타임에서 데이터의 형태로 휴대기기 쪽으로 다운로드 할 수 있도록 Java 를 리호스트(Rehost)하는 스크립트 언어와 같은 제품을 제작하기도 한다. J2ME/Brew 네트워크 휴대기기용 언어는 주로 UIEvolution 시스템을 많이 구매하는 편이다. 하지만 UIEvolution 은 값이 비싸고 오직 스크립트만 처리할 수 있으며(레이아웃은 처리하지 못한다.) Java 자체와 맞먹는 크기의 코드를 생성한다. 나는 그것보다 더 좋은 언어를 만들 수 있으리라 생각해서 FLIRT 를 제작했다.

FLIRT 는 단순히 Java/Brew 코드에 1:1 로 대응하는 스크립트 언어가 아니다. FLIRT 는 휴대전화에 특화된 스크립트와 디스플레이를 결합시킨 언어이다. 그러므로 크기가 매우 중요한 요소였다. Java 가 매우 간략화된 코드를 생성하도록 디자인된 언어기는 하지만 FLIRT 는 Java 의 코드보다 50%가량 더 작은 코드를 생성한다. Java 는 큰 기종이건 작은 기종이건 간에 범용 언어로 작동하도록 설계되었다. 하지만 휴대전화에는 상당한 제약사항들이 존재하며 잘 디자인된 스크립트 언어가 이런 제약사항들에 더 적합할 것이다.

휴대기기용 응용 프로그램에서 주로 "재사용(Reusable)"되는 구성요소는 보통 UI 를 관리하는 코드이므로 나는 FLIRT 를 각 화면별 상태 기술(State Description)부터 조직화시켜 나갔다. 상태 기술에는 어떤 소프트키(softkey)를 사용하고, 요청된 이벤트에 대한 응답으로

어떤 스크립트 코드를 실행하며, 어떤 순서로 화면에 그래픽 요소들이 그려지는지가 포함된다.

내가 처음 결정한 디자인 결정사항(Design Decision)은 1 바이트가 넘어가는 것들(32 비트 정수, 색깔, 글꼴, 문자열, 그래픽 등)은 모두 테이블(Tables)을 활용해 간접적으로 접근하도록 모든 스크립트 인자를 1 바이트로 제한한 것이었다.

두 번째 디자인 결정사항은 지역 변수(Local Variables)를 제거한 것이었다. 모든 사용자 데이터는 일반적으로 바이트, 32 비트 정수, 문자열, 그래픽 등을 저장하는 전역 2 차원 배열에 보관되게 된다. [0][]은 서로 관련이 없는 값들을 임시적으로 보관하며 다른 1 차원 요소들은 관련된 정보들의 집합을 보관한다. 이런 관련된 집합의 예로 strings[3][]이라는 차원에 메뉴에서 사용할 모든 문자열의 집합을 보관할 수 있다. 그러므로 메뉴를 정의하는 스크립트 요소는 메뉴 문자열을 3 이라는 값을 가지는 하나의 바이트로 참조할 수 있게 된다. 물론 이 언어는 차원에 자동적으로 레이블(Label)을 할당하므로 숫자를 아예 사용하지 않을 수도 있다.

일례로 전역 문자열 배열은 다음과 같이 선언할 수 있다.

```
stringSets=
{
        {
        sVMENU_POPUP_INSTRUCTIONS: null,
        sBACKUP_GALLERYTITLE: null,
        sBACKUP_GALLERYNAME: null,
        sVERSION:   "MIDlet-Version",
        }
sTRENDS_MENU: {
  "the look",
  "hot finds",
  "beauty buzz",
  "deals& steals",
  },
}
```

위의 코드에서 stringSets[0]에는 아무런 레이블도 할당되지 않지만 멤버(Member)들에는 고유한 레이블(콜론이 뒤에 찍혀있는 이름)이 할당되며 간혹 초기값(Initial Values)이 할당되는 경우도 있다. stringSets[1]에는 sTRENDS_MENU 라는 레이블이 할당되었으며 메뉴 항목의 이름이 될 수도 있는 무작위 문자열들의 목록을 담고 있다.

사용자 스크립트는 데이터 배열에 이름이 할당된 초기값을 가지고 있을 수도 있으며 사용할 글꼴, 색깔, 소프트키를 기술하는 배열이 따로 있을 수도 있다. 이런 모든 항목들을 사용할 때는 1 바이트 값(이름)을 사용해 접근할 수 있다.

예를 들어 화면 어딘가에 일련의 문자열을 출력하는 서브루틴(Subroutine)을 호출하는 것이라고 생각하기 쉬운 TextRect 디스플레이 구성요소를 살펴보자. TextRect 는 실제로 코드가 아니고 인터프리트 된 데이터에 불과하고 스크립트 언어를 사용해 수정이 가능한 데이터로 구성되어 있다. 즉, 스크립트에서는 내용을 스크롤하고, 주변으로 이동시키거나, 어떤 곳을 가리키고 있는 가에 따라 표시 문자열을 변경한다던가 하는 등의 일이 가능하다는 것이다. 어쨌거나 TextRect 를 호출할 때 구성요소를 완전히 선언하려면 15 개의 인자가 필요하다. 각 인자는 항상 1 바이트의 크기를 가지므로 15 바이트의 메모리를 사용한다.(J2ME 에서 함수 호출 시 사용하는 양의 절반)

각 바이트는 다음과 같은 사항을 나타낸다.

1. 구성요소의 종류(TextRect)
2. 스크립트가 TextRect 와 상호작용을 할 수 있도록 하고 작업과 관련되지 않은 구성요소가 반응하지 않도록 하기 위한 ID 레이블
3. 수행 방식을 조작하기 위한 변경자 비트(Modifier Bits) 값(왼쪽, 오른쪽, 중앙 정렬이나 스크롤과 같은 수행 방식을 포함)
4. 표시할 문자열
5. 화면의 어떤 구성요소에서 상대적으로 얼마만큼 떨어진 위치인가(즉, 다른 구성요소가 작거나 큰 그래픽으로 표시되면 화면 크기에 따라 자동적으로 크기가 조절된다는 소리임)
6. 이 구성요소를 표시하기 위해 x 와 y 축으로 얼마만큼의 영역을 예약할 것인지
7. 어떤 글꼴을 사용할 것인지(일반적인 상태와 강조표시 상태를 가지는 사용자 정의 글꼴 포함)
8. 현재 스크롤 정도와 스크롤 제한(문자열이 필드보다 더 길다면 줄이나 페이지 단위로 스크롤 가능)

세 번째 디자인 결정사항은 화면 레이아웃을 상대적으로 처리한 것이다. FLIRT 의 디스플레이 구성요소들은 상대적인 좌표를 사용해 디자인되었으므로 다양한 화면 크기와 데이터에 따라 적합한 크기로 표시되게 된다.

이벤트에는 소프트키 이벤트(Softkey Events), 키패드 이벤트(Keypad Events), 의사 사건(Pseudo Events)이 포함된다. 지역 변수가 없으므로 사용자가 호출할 수 있도록 전역 데이터 배열의 특정 지점을 예약해놓는 경우를 제외하고는 "호출 인자(Calling Arguments)"도 없다. 그러므로 이벤트를 발생시키는 것과 특정 상태의 이벤트 코드에 대한 서브루틴을 호출하는 것은 동일한 메커니즘을 사용한다.

FLIRT 는 공용 라이브러리 자원(Common Library Resource)처럼 작동하므로 사용자 정의 글꼴, 문자열 래핑(Textwrapping), 네트워킹(Networking), 정렬(Sorting), RMS(파일) 접근과 같은 것들을 포함해서 컴파일 하거나 따로 컴파일 할 수 있다. 인터페이스 구성요소에는 문자열 블록(Textblock), 수평/수직 메뉴(Horezontal and Vertical Menu), 마키(Marquee), 그래픽(Graphic), 스크롤 막대(Scrollbar), 화면 키보드(Screen Keyboard), 클리핑 영역(Clipping Area), 카메라 기능이 포함되어 있다.

네 번째 디자인 결정사항은 모든 그래픽 구성요소에서 기본적으로 스크롤을 지원하도록 한 것이었다. 스크립트를 사용하면 FLIRT 는 그래픽(애니메이션 포함), 문자열, 도형(진행상태 표시 막대 등)을 스크롤 할 수 있다. 디스플레이 구성요소에 연결된 스크롤 막대는 자동적으로 해당 구성요소의 스크롤 상태를 반영하게 되므로 현재 스크롤 상태를 나타내는 고전적인 수직이나 수평 스크롤 막대를 표시할 수 있다.

LimeLife 의 제품은 일반적으로 아름다운 디자인을 선호하는 여성을 판매 목표로 잡고 있었으므로 단순한 메뉴는 용납될 수 없었다. 좌측에 표시된 에뮬레이터 화면은 LimeLife 의 Instyle 제품의 화면이다. 메뉴는 하나의 메뉴 구성요소로만 이루어진 것이 아니라

글자(Character) 구성요소(파란색으로 표시된 숫자), 문자열 구성요소(전체가 검은색으로 표시된 메뉴), 색깔이 입혀진 도형 구성요소(강조 막대와 구분 막대)가 포함된 여러 디스플레이 구성요소의 조합으로 이루어져 있다. 라우터(Router) 구성요소에서 디스플레이 구성요소들의 행동을 통제한다. 라우터 구성요소는 다른 구성요소들의 이름 목록을 가지고 있으며 라우터를 스크롤 하면 다른 구성요소가 선택(강조 표시)되게 된다.

위의 화면은 라우터를 포함해 70 여개의 독립적인 구성요소들로 구성되어 있다.(이 중의 반 정도는 통제용이거나 각 구성요소를 일정간격을 가지고 수직으로 정렬하는 보이지 않는 레이아웃 구성요소들이다.) 그래픽이나 글꼴의 크기가 변경되면 화면은 자동적으로 그에 맞도록 조절된다.(네트워크 연결에 실패해 어떤 그래픽을 사용할 수 없을 때도 자동으로 조절된다.) 그리고 모든 화면 크기에서 작동하며 BREW 와 J2ME 를 사용하는 모든 휴대전화에서 잘 동작한다.

위의 상태에 해당하는 스크립트 코드는 화살표, 소프트키에 반응하며 빠른 선택을 위한 숫자키에도 반응한다. 그리고 가능하다면 RMS 의 주요 그래픽을 캐시(Cache) 처리한다. 전체 상태 기술의 용량은 1093 바이트(스크립트 + 디스플레이 구성요소)이며 가장 복잡한 상태 중의 하나이다.(평균적으로는 상태 당 219 바이트이다.) 위의 제품은 83 개의 상태를 가지고 있으며 전체 크기는 18,169 바이트이다.(디스플레이 구성요소와 스크립트를 합한 크기이며 문자열이나 이미지는 포함하지 않았다.)

코드와 디스플레이 정보가 한 곳에 있는 덕분에(상태 기술) 디스플레이 코드는 하나의 커다란 switch 절에 들어있고 통제 코드는 다른 switch 절에 들어있던 예전 J2ME 로 코딩 된 응용 프로그램보다 관리가 편리하다. (J2ME 의 크기 문제 때문에 종종 객체 지향적인 코드를 작성하지 않고 거대한 switch 절을 작성하는 경우가 많았다.)

모든 화면 구성요소와 스크립트 코드가 단순히 “데이터”이기 때문에 원하기만 한다면 새로 작성되거나 수정된 상태를 서버에서 다운로드 받는 것도 가능하다.

FLIRT 를 사용하면 새로운 제품의 시제품을 매우 빠르게 제작할 수 있었다. 두 번째로 FLIRT 기반의 제품을 개발할 때 완전한 모양새를 갖춘 데모를 작성하는데 단지 1 주일 정도만 걸렸다.(조작하면 단순히 화면과 화면간의 전환만 가능하고 다른 기능은 하나도 구현되지 않은 데모) BREW FLIRT 엔진이 제작되었을 때 3 가지의 완전히 다른 FLIRT 기반 제품의 제작이 바로 시작되었다. 그 중 한 제품은 J2ME 용 데이터 파일을 제작한 후에 BREW 용 제품에 그대로 적용하기도 했다. 그렇게 해도 화면 자동 레이아웃과 모든 스크립트가 정상적으로 작동했다.

결론

새로운 스크립트 언어를 작성하는 것은 단순히 코딩만 하는 것이 아니므로 상당한 부담이 따르게 된다. 하지만 스크립트 언어를 작성해야 할 어쩔 수 없는 이유가 있다면 밑바닥부터 시작할 이유가 없다. 새로운 언어를 개발할 때는 자신의 생각을 간결하게 표현할 수 있는 방법을 창조하라. 그러면 그 언어는 프로그래밍에 있어서 아주 크나 큰 이익을 가져다 줄 것이다.

저자에 대해

저는 컨설턴트이자 특정 회사의 직원이기도 합니다.(하와이에 살고 있는 장거리 재택 근무자이며 본 아티클이 업로드 되었을 때는 영국의 글로스터에 살고 있었습니다.) 제게 연락하시려면 gowilcox@gmail.com이나 Gamasutra에 있는 제 이력서를 참조해주십시오.