HTML5로의 이행 - 1부

(Making the Move to HTML5, Part1)

작성자: 데이비드 갈레아노, 던컨 텝스 (David Galeano, Duncan Tebbs)

작성일: 2013년 2월 7일

이 글은 3 부작 중 제 1 부로, 전직 크리테리온의 노련한 콘솔 개발자들이 HTML5 개발로의 대세 변화 과정에 대해 논하고 있다. C 에서 작업할 때와 자바스크립트에서 작업할 때의 차이점, 또 브라우저 환경에 따라 부딪치게 되는 난관들에 대해 설명하고 있다.

모든 플랫폼이 그렇듯이 HTML5 에 대해서도 혹평을 하는 사람들이 있다. 이중에는 정당한 평도 있지만 일부는 루머이거나 지난 정보에 근거한 것이다. 반면 업계의 성공한 많은 비즈니스 리더들이 HTML5 를 지원하고 있으며, turbulenz.com[1] 의 많은 게임들은 HTML5 야말로 오늘날 고급 게임에 적합한 프로그램임을 증명하고 있다. 여기서 우리가 HTML5 를 지원해야 하는 이유를 자세히 살펴보려로 한다.

HTML5 관련 표준은 고급 게임이 웹 기반에서 실행 가능하도록 하는 플랫폼으로서 빠르게 성장하고 있다. 이렇게 웹 표준을 타깃으로 만든 게임은 다양한 컨텍스트로 구현할 수 있으며, 최신 디바이스와의 연결성이 좋은 것은 물론 하드웨어까지 활용할 수 있는 장점이 있다.

터뷸렌즈(Turbulenz)에서는 많은 연구 인력과 엔지니어들을 투입하여 웹 기반의 고성능 게임을 개발해왔다. 콘솔 게임 기술을 개발한 경험이 풍부한 게임 개발자의

[1] 참조링크: https://turbulenz.com/

관점에서 HTML5 및 관련 기술들에 접근하여 정확성과 높은 성능을 구현해내고 있다.

이 글에서는 콘솔에서 웹으로 이동하면서 우리가 경험한 것들, 즉 잘된 사례들과 이 기술들을 사용한 시기, 최적화 전략, 문제 대응 방법에 대해 논하고자 한다.

터뷸렌즈를 사용해서 제작한 게임들 중 일부는 게임 네트워크인 turbulenz.com[2] 에서 해볼 수 있다. 이것들은 우리의 HTML5 플랫폼 및 인프라를 사용하여 온라인 게이머들도 고품질 3D 게임을 즐길 수 있도록 제작한 것이다. 당사는 2009 년 초에 일렉트로닉 아츠(Electronic Arts)의 몇몇 디렉터들과 리드 프로그래머들이 모여 설립했으며, 현재는 세계적으로 우수한 엔터테인먼트사들의 개발자들과 사업가들이 모인 기업으로 성장하였다.

1 부에서는 현재 HTML5 의 대세와 게임 관련 기술들에 대해 다루고, 향후 게임 개발자들이 기대할 수 있는 개발 환경과 워크플로우(workflow)에는 어떠한 것들이 있는지 얘기하려고 한다.

[2] 참조링크: https://turbulenz.com/

게임 플랫폼으로서의 브라우저

브라우저는 새로운 OS 가 되어 많은 기회를 가져다 주었지만, 문제점들도 안고 있다. 과거에 게임은 전통적으로 고립된 상태에서 하는 것이었고, 많은 자원을 투입하여 기계의 최고 성능을 이끌어내는 것이 관건이었다. 또한 브라우저에서 사용하는 게임 코드는 다른 탭이나 윈도우즈에서 실행 중인 다른 게임이나, 해당 기기에서 실행중인 다른 모든 프로세스와 경쟁해야 한다.

어떤 점에서는 각각의 브라우저를 각기 다른 플랫폼으로 다루어야 마땅하다. 모든 브라우저에서 공통으로 가능한 일부 인터페이스도 있고 항상 사용 가능한 리소스도 있지만, 기능을 더욱 개선하기 위해서는 항상 런타임 특성을 점검하고 멀티플 코드 경로와 제 2 의 해결책(workaround)을 제공해야 한다.

브라우저와 브라우저 버전, OS 와 OS 버전을 다루는 테스트 매트릭스는 방대하다. 게다가 하드웨어 리소스는 네이티브 애플리케이션처럼 직접 엑세스할 수 없음에도 개발자들은 그래픽 드라이버 버그와, 이 드라이버와 인터페이스하는 브라우저 레이어의 버그 문제를 해결해야 한다.

브라우저 업데이트가 빨라지고 자동으로 업데이트되는 브라우저도 있다 보니, 테스트가 필요한 특정 지원 버전을 리스트로 만들어 관리하기는 어려워지고 있다. 터뷸렌즈에서는 항상 최신 버전의 브라우저로 작업을 하며, 많이 사용하는 브라우저 버전은 리스트를 만들어 관리하고 있다. 이 글을 쓰고 있는 지금 이 리스트에는 인터넷 익스플로러 8, 9, 10, 파이어폭스 3.6, 최근에는 사파리 5.1 과 맥 OS X, 최신 버전의 크롬까지 포함되어 있다. 우리는 느린 넷북부터 고성능 데스크톱에 이르기까지 다양한 하드웨어에서 Nvidia, AMD, 인텔 등 흔히 사용하는 비디오 카드들을 모두 사용하여 테스트한다.

브라우저의 상대적인 시장 점유율은 항상 유동적이고, 인구 구성과 일치하지도 않는다. 먼저 타깃 시장을 정하고 이 시장과 유사한 고객 그룹의 사이트에 해당하는 브라우저 스탯(stat)을 찾아내는 것이 더 바람직할 것이다.

개발 플랫폼으로서 브라우저는 다른 플랫폼에 비해 몇 가지 이점이 있다. 일반적으로 이터레이션(iteration)이 매우 빠른 것이 특징이다. (자바스크립트는 보통 실행 과정에 컴파일 단계가 없으며, 페이지를 즉각 리프레시(refresh)하여 최근 버전의 코드를 로드해 실행한다). 웹 개발자들은 디버깅과 프로파일링을 위한

다양한 도구를 사용해왔으며, 이는 효율적인 개발 환경 조성에 큰 도움이 되고 있다. 이에 대해서는 추후 다시 자세히 설명하기로 한다.

일반적으로 개발자들이 브라우저에 코딩을 할 때는 방어적으로 할 것을 권한다. 다양한 브라우저로 테스트를 해야 하며, 정기적으로 새 버전의 상태를 확인해야 한다.

게임 개발을 위한 자바스크립트

브라우저에서 사용할 수 있는 다목적 프로그래밍 언어는 하나밖에 없다. 자바스크립트가 그것인데, 공식적으로는 ECMScript 라는 이름으로 알려져 있다. 이 글에서 프로그래밍 언어에 대해 설명하려는 것은 아니지만, 우리가 C/C++ 기반에서 자바 스크립트에 접근했을 때 발견한 특징적인 현상들이나 예상 외의 현상들에 대해서는 고찰해볼 만한 가치가 있다.

특히 처음 C/C++에서 자바스크립트로 넘어가면서 개발자들은 여러 가지 도전 과제에 당면하게 된다. 자바스크립트의 구조는 C 와 비슷하지만 언어 자체는 기능적 언어인 리스프(Lisp)와 공통점이 더 많다. 클로저(closure)와 일급함수(first-class functions)가 있기 때문이다.

클로저는 매우 강력하여 비동기(asynchronous) 프로그래밍도 훨씬 쉽고 명확하게 할 수 있다. 그러나 개발자가 작업 과정을 정확하게 인지하지 않으면 교묘한 에러들이 발생할 수 있다.

예를 들어 루프(loop) 내부에 클로저를 만드는 것은 흔히 하는 실수이다. 해당 루프를 수행하는 중에 계속 변화하는 변수를 참조하는 것을 말한다. 이렇게 되면 클로저에서 참조한 변수가 클로저가 만들어진 시점이 아니라 실행된 시점의 값을 갖기 때문에 혼란을 초래한다.

자바스크립트에는 일급함수가 있어서 다른 함수에서 리턴되고 변수에 할당되고 딕셔너리에 저장되며, 파라미터(parameters)로 넘기는 것이 가능하다. 스트링(String)은 런타임에 함수로 컴파일될 수 있지만, 스트링이 알려지지 않은 소스에서 넘어와서 코드의 다른 부분도 방해할 수 있기 때문에 안전 문제로 이 방법은 추천하지 않는다.

자바스크립트는 객체지향적이지만 클래스가 없다. 자바스크립트는 컨스트럭터(constructors)가 있으며 프로토타입 중심의 상속(inheritance)을 한다. 객체는 키값(key-value) 딕셔너리처럼 모든 것을 네임(name)으로서 최적 상태로 저장하고 검색할 수 있다. 데이터를 저장하거나 검색하는 코드 네임의 오타 등 잘못된 네임으로 값에 접근하는 것 또한 흔히 하는 실수이다.

객체는 다른 객체의 프로토타입으로 배치될 수도 있다. 객체에서 속성(property)이 발견되지 않으면 런타임에서 프로토타입 객체를 점검할 것이다. 함수가 객체에 저장될 수 있고 객체들은 프로토타입을 공유할 수 있기 때문에, 프로토타입 메커니즘에 따르면 메소드(method)와 코드는 익숙한 방식으로 재사용할 수 있다. 함수도 객체처럼 작동하며 네임으로 속성을 저장하고 검색하는 데 활용될 수 있다.

자바스크립트 객체에는 디스트럭터(destructor)가 없다. 자바스크립트 런타임은 더 이상 참조할 값이 없으면 디스트럭션을 시작하고, 자바스크립트 코드는 디스트럭션이 되어도 전혀 를알림(notification)을 받지 않는다. 우리는 자바스크립트 코드에서 메모리 누수 (프로그램이 필요하지 않은 메모리를 계속 점유하고 있는 현상) 문제가 심각하다는 것을 발견했다. 일부 자바스크립트 프로파일러에서는 객체에 힙 스냅샷들(heap snapshots)의 카운트(counts)를 제공하기도 하는데, 이는 도움이 되기는 하지만 이렇게 하기 위해서는 객체가 리터럴이 아닌 (non-literal) 컨스트럭터로 만들어져야 한다. 이 때 이들을 구분하기위해 새로운 오퍼레이터가 필요하다. 일부 프로파일러에서는 또한 힙 스냅샷들에서 레퍼런스 트래킹(reference tracking)을 제공하기도 하는데, 이는 해당 객체가 가비지 컬렉트(garbage collect)되지 않은 이유를 밝히는 데 도움이 된다.

가장 좋은 사례로는 객체 소유(ownership) 정책을 분명하게 정의하는 방법을 추천한다. 이렇게 하면 주어진 객체의 참조값을 점유하는 (또는 내보내는) 코드가 어디에 있는지 찾아내기가 용이해진다.

자바스크립트는 스태틱 타입(static type)이 없기 때문에 프로그램이 다할 때까지 다양한 타입의 변수를 갖는 것이 허용된다. 또한 같은 오퍼레이션에서 다른 타입이 사용될 경우 일부는 자동 변환이 일어나기도 한다. 예를 들어 스트링에 숫자를 더해서 스트링으로 변환된 숫자의 스트링 병합(concatenation)을 일으킬 수도 있다. 이러한 에러는 찾기도 힘들고 성능에도 문제를 일으키게 된다.

또 한 가지 버그의 민감한 원인은 파라미터를 빅엔디안 오더(big-endian order)의 32 비트 부호 붙임 정수(signed integer)로 변환하는 자바스크립트 비트와이즈(bitwise) 방식 오퍼레이션과 둘이 보완하는 형식 때문이다. 예를 들어서 다음 두 공식은 둘 다 '참'이다.

(0x80 << 24) !== 0x80000000

(0x80 << 24) === -2147483648

만일 무부호 정수(unsigned integar)가 사용되었다면 둘 다 '거짓'이었을 것이다. 트리플-이퀄(triple-equal) 오퍼레이터를 사용했다는 데에 주목해보자. 피연산자가 다른 타입이었다면 타입 변환(type conversion)이 일어나지 않는다. 반면 더블 이퀄(double-equal) 오퍼레이터는 타입 변환을 수행하며, 에러를 코드 속에 숨길 수도 있다.

자바스크립트 변수가 불값(Boolean value)을 취할 수도 있지만, 다른 모든 기본 타입을 조건식으로도 사용할 수 있으며 이때 다음 값은 '거짓'이 된다. **null**, **undefined**, the empty string, the number zero, the number NaN.

자바스크립트에는 블록 범위(block scope)가 없다. 글로벌 범위(global scope)와 유효 범위(function scope)만 있을 뿐이다. 변수들은 마치 함수 상부에서 선언된 것처럼 함수 내 어디서나 창조될 수 있고, 어디서나 접근이 가능하다. C++에 익숙하지만 자바스크립트에는 익숙하지 못한 개발자라면 여기서 혼란을 느낄 것이다.

자바스크립트에서 예외는 지원되지만 그 언어는 예외적 계층 구조를 지원하지 않는다. '스로 오퍼레이터(throw operator)'는 객체, 숫자, 스트링 등 어디에나 쓸 수 있으며, 코드를 둘러싼 첫 번째 '트라이/캐치(try/catch)'로 잡을 수 있다. 특정 타입의 객체들만 잡는 것은 불가능하다.

이 언어는 본래 형태로부터 눈에 띄게 변화하고 있으며, 일부 특성들은 모든 브라우저에 표준화되어 있지 않다. 최근에 더해진 특성들은 사용하기 전에 점검이 필요하다. 예를 들어 객체 속성을 위한 게터(getter)와 세터(setter)를 정의하는 기능인 **Object.defineProperty** 는 최근에 매우 유용한 툴로 사용되고 있다. 그러나 서포트(support)가 존재하지 않을 때에는 추가로 코드 패스를 제공해야 한다. **Object.defineProperty** 처럼 함수가 존재하는지 테스트만 해봐도 지원되는

특성을 점검할 수 있는 경우도 있다. 이 경우가 아니라면 코드는 '트라이/캐치(**try / catch)** 블록' 내에서 실제로 이 특성을 사용해봄으로써 예외가 나타나는지 점검해야 한다.

자바스크립트는 다이내믹한 특성과 기능적 요소들을 가진 강력한 언어지만, 큰 프로젝트에 사용하기에는 너무 복잡한 언어다. 다수의 개발자들이 수십만 가지 코드 라인의 코드 베이스에서 동시에 작업할 때는, 올바른 원칙과 도구가 없으면 수많은 실수가 발생하게 마련이다.

터뷸렌즈에서는 코드 품질에 집중하는 개발 과정을 도입하여 개발 과정에서의 실수를 최소화하고 있다. 엄격한 코딩 스탠다드, 코드 리뷰, 자동 유닛 테스트를 수시로 수행한다. 또한 정기적으로 스태틱(static) 분석 툴을 사용하여 코드에 흔히 일어나는 실수가 없는지 체크한다. 아직까지 C 또는 C++에 활용되는 수준의 검사 툴은 찾아볼 수 없지만, 몇 가지 툴, Closure Linter[3]와 JSHint[4], JSLint[5]는 상당수의 버그를 잡아내는 데 도움이 됐으며, 앞으로 더욱 개선될 것으로 기대된다.

터뷸렌즈에 새로 입사한 소프트웨어 엔지니어들은 모두 더글러스 크록포드(Douglas Crockford)의 <JavaScript: The Good Parts[6]>를 한 권씩 받는다. 이 책은 자바스크립트에 대한 훌륭한 입문서로, 흔히 하는 실수들과 우수 사례들을 많이 소개하고 있다.

변환 도구

다른 언어를 자바스크립트로 변환하는 도구에 관심이 있을 개발자들이 있을 것이다. 실제로 특정 언어를, 예를 들어 런타임 오류를 최소화하기 위한 스태틱(static) 타입에서 이후 자바스크립트로 변환하는 것이 가능하다.

많이 사용되는 도구로는 haXe[7]와 Dart[8], Emscripten[9]이 있다. TypeScript[10] 프로젝트는 비교적 최근에 나왔는데, 자바스크립트에 타입(type) 관련 정보까지

---

[3] 참조 링크: https://developers.google.com/closure/utilities/

[4] 참조 링크: http://www.jshint.com/

[5] 참조 링크: http://www.jslint.com/

[6] 참조 링크: http://shop.oreilly.com/product/9780596517748.do

[7] 참조 링크: http://haxe.org/

[8] 참조 링크: http://www.dartlang.org/

[9] 참조 링크: https://github.com/kripken/emscripten/wiki

명시할 수 있도록 기능이 확장되었다. TypeScript 는 에러 점검에 도움이 되는 것은 물론, 자바스크립트 구버전과 호환 가능(backwards compatible)한 것이 특징이다.

이를 사용하고자 한다면 그 용도를 명확하게 알아야 한다. 자동으로 발생하는 자바스크립트 코드는 최상의 성능을 내지 못하며, 직접 작성한(Hand-written) 자바스크립트 코드보다 사이즈가 크게 늘어나기도 한다. 특정 환경에서 잘 활용된 도구가 다른 환경에서는 그렇지 못할 수도 있는데, 이 때는 멀티플 코드 패스(paths)가 필요하다. 디버깅(debugging) 문제와 본래의 오류 코드로 다시 추적하는 작업은 매우 어렵고, 안전한 자바스크립트를 생성하기 위해 원래 소스 코드를 약간 수정하여 컴파일러를 '속이는(trick)' 것이 필요할 수도 있다. 특정 브라우저의 특성에 대한 서포트가 없다는 점 또한 한계이다.

그러나 이 도구들은 분명히 흥미로운 특징들을 가지고 있으며 (예를 들어 C++를 가비지 컬렉션(garbage collection) 하지 않는 자바스크립트로 변환하는 능력), 계속해서 개선되고 발전하고 있는 것도 사실이다. 이들을 가능한 방법으로 염두에 두고 발전 과정을 지켜보는 것이 좋으리라고 본다.

에디터

스태틱 타입 언어를 위한 에디터는 다이내믹 언어에서는 제공하기 힘든 많은 기능을 제공할 수 있다.

자바스크립트를 지원하는 우수한 에디터들은 다음과 같은 특성들을 제공한다.

- 구문 하일라이팅 (Syntax highlighting)
- 다음 사항에 대한 자동 완성 기능
    - 디폴트 글로벌 객체(Default global objects).
    - jQuery[11] 등 잘 알려진 외부 라이브러리.
    - 현재의 함수 내에서 정의되는 변수와 함수.
    - 현재의 파일 내에서 정의되는 글로벌 변수와 글로벌 함수.

- 현재 파일 레벨에서의 리펙터링(Refactoring).

---

[10] 참조 링크: http://www.typescriptlang.org/

[11] 참조 링크: http://jquery.com/

- JSLint 나 JSHint 와의 통합.

커스텀 객체(custom objects)를 위한 자동 완성 요소, 다른 파일에서 온 글로벌 변수 등은 다이내믹 언어가 부족하다. 일부 에디터들은 현재 자신들의 자바스크립트 엔진을 사용하는 프로젝트에 있는 모든 자바스크립트 코드를 분석하여 실행하기 시작했다. 이는 현재 시점 코드 내부의 변수와 속성(property)의 활용 가능성을 판단하는 데 도움이 될 것으로 기대된다. 그러나 이 에디터들이 부정확하거나 불완전한 제안을 하는 경우도 적지 않다. 특정한 방식으로 자신의 코드를 작성하는 것은 에디터들의 제안을 보완하는 데 도움이 된다. 예를 들어 컨스트럭터 내의 모든 속성을 초기화하는 것도 한 방법이다. 이 경우에 에디터는 현재 객체를 만드는 데 어떤 컨스트럭터가 사용되었는지 추적할 수 있으며 속성을 제안하기 위한 점검을 수행할 수도 있다.

런타임과 성능

자바스크립트는 엄격한 보안 통제와 외부세계와 한정된 인터렉트만 하는 API 한도 내에서 가상머신(Virtual Machine)으로 작동한다. 최근 자바스크립트 엔진은 적기공급생산(JIT) 컴파일레이션을 원칙으로 하기 때문에 기존의 번역된 바이트코드(bytecode)에 비하면 훨씬 빠르게 머신 코드를 생성한다. 최근에 나온 엔진들은 매번 출시될 때마다 성능이 개선되고 메모리 사용이 감소되는 등 빠른 속도로 진보하고 있다. 물론 예상대로 성능은 잘 작성된 C/C++ 수준에 비하면 떨어진다.(우리는 브라우저에 따라 자바스크립트가 4 배에서 10 배까지 더 느리다는 것을 발견하였다.) 그러나 이 차이는 새 버전이 출시될 때마다 점점 줄어들고 있다.

브라우저 내 작은 코드 스니핏(snippets)들의 성능을 평가하고 주어진 오퍼레이션을 수행하는 가장 빠른 방법을 찾기 위해 우리는 jsPerf[12]를 사용한다. 불행히도 때때로 모든 브라우저(모든 버전들을 통틀어)에서 가장 빠른 코드 스니핏이 없는 경우도 있는데, 이때는 시장 점유율에 근거하여 타협할 수밖에 없다.

메모리 할당은 큰 게임에서 문제가 된다. 512 메가바이트 정도면 콘솔에서는 잘 맞지만, 브라우저에서는 기가바이트 이상이 필요할 수도 있다. 모든 자바스크립트 객체는 동급의 C++에 비하면 과부하가 높다.

---

[12] 참조 링크: http://jsperf.com/

사용된 객체 타입도 메모리 크기에 영향을 준다. 자바스크립트 엔진은 숫자를 32 비트 부호 붙임 정수형(signed integers) 또는 64 비트 부동소수점(floating point)로 수행하며, 대부분의 경우 둘 다 비슷한 양의 메모리를 차지한다. (8 바이트 정도)

큰 숫자 어레이(Array)의 메모리 사용을 줄이기 위해 가능하면 타입 어레이(typed arrays [13])를 추천한다. 우리는 어떤 데모에서 3 D 벡터의 어레이를 Float32Array 로 바꾸어 메모리를 20 퍼센트나 절약한 사례가 있다.

타입 어레이는 일반적으로 보통 어레이에 비해 더 좋은 성능을 보인다. 대부분 JIT 컴파일러들은 타입 어레이에서 사용한 메모리를 직접 나타내는 방법을 알고 있기 때문에, 데이터 타입이 정확하게 예측될 수 있을 때는 매우 효율적으로 코드를 생성하여 접근하고 오퍼레이트할 수 있다.

위에서 언급한 것처럼 자바스크립트는 가비지 컬렉션을 사용하여 더 이상 참조가 필요 없는 객체를 폐기한다. 가비지 컬렉터들은 '마크-앤드-스윕(mark-and-sweep)' 알고리듬을 사용하여 폐기해도 안전한 객체를 찾아낸다. 일부에서는 멀티플 제너레이션(multiple generation) 개념을 활용하여 수명이 짧은 객체들을 배치하고 해산하며, 인크리멘털 알고리듬(incremental algorithms)을 사용하여 '마킹'과 '스위핑'의 비용을 배분할 수도 있다. 이 또한 적극적으로 개발되고 있는 분야이다.

주어진 시간에 활동 중인 전체 객체의 수는 가비지 컬렉션 비용에 직접 영향을 주는 요인이다. 수백만 개의 활동 객체가 있다고 할 때, 오래된 가상기계들이라면 컬렉션 동안 전체 세계를 몇 초간 멈추게 할 수도 있을 것이다. 이 상황은 점점 개선되어 요즘에는 수백만 객체도 1000 분의 1 초의 수백 배 정도만 지체될 뿐이다. 그러나 이 정도도 게임에서는 '스킵(skip)'으로 간주된다. 가비지 컬렉션은 보통 연속적으로, 또는 고정된 시간 내에 많은 객체가 만들어질 때 일어난다.(예를 들어 엔진에서 매 10 초마다 '풀 스윕(full sweep)'을 발동하기도 한다.)

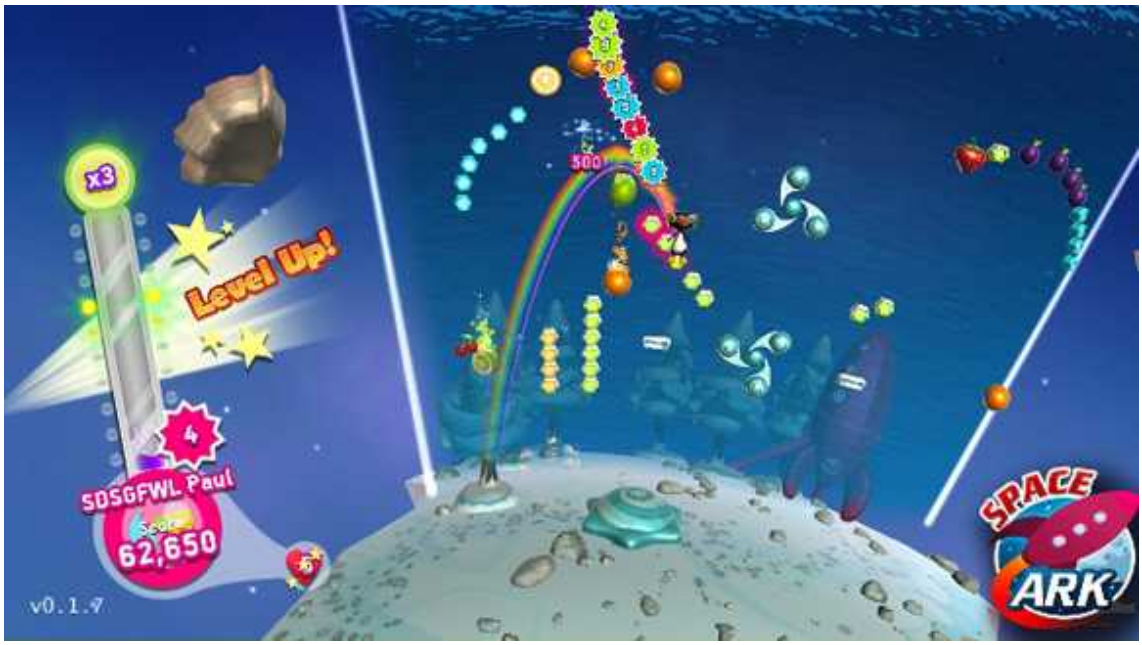
당연한 얘기지만, 우리는 성능을 향상시키기 위해서 창조되는 객체의 수를 가능한 적게 해야 한다는 것을 깨달았다. 우리가 제작한 일부 데모나 사례에서는 프레임당 객체 창조가 전혀 없는 것들도 있다.

---

[13] 참조 링크: http://www.khronos.org/registry/typedarray/specs/latest/

우리는 객체 수를 줄이는 유용한 전략을 몇 가지 발견했다. 함수에서 함수로, 또 프레임에서 프레임으로 다이내믹 어레이를 재사용하는(예를 들어 스크래치패드를 사용) 것은 매우 효율적이다. 또한 객체 어레이를 인터리브 속성(interleaved property)이 있는 플랫 어레이(flat array)로 변환하는 것도 고려해볼 만하다. 우리 웹사이트의 한 게임에서는 이 방법만으로 객체 카운트를 75 퍼센트나 줄여서 가비지 컬렉션으로 일어나는 멈춤 현상 문제를 개선한 바 있다.

어떤 경우에는 정보와 명령들을 맞춤 바이트코드(bytecode)로 인코딩함으로써 객체수의 런타임 성능에 균형을 찾기도 한다. 예를 들어 SVG path[14]를 저장하거나 특정 렌더링 형태를 위한 인스트럭션을 포함하는 하나의 스트링을 보유하는 것, 또는 이 인스트럭션들을 수시로 디코딩하는 작업들은 스트링을 언패킹(unpacking)하고 인스트럭션을 객체의 위계로서 저장하는 것에 비하면 메모리와 객체를 적게 사용할 것이다.(그러나 CPU 시간은 더 오래 걸린다.)



스페이스 아크(*Space Ark* [15])는 우리가 개발자들과 함께 긴밀하게 작업하면서 여기서 논한 방법들을 사용하여 최적화된 상태로 제작한 게임이다. 다이내믹하게 만들어진 객체의 수를 크게 줄임으로써 모던 자바스크립트 런타임에서 게임플레이를 하는 동안 가비지 컬렉션 멈춤 현상을 본질적으로 제거할 수 있었다.

---

[14] 참조 링크: http://www.w3.org/TR/SVG/paths.html#PathDataGeneralInformation

[15] 참조 링크: https://turbulenz.com/#games/space-ark

또한 캐릭터 애니메이션과 파티클 효과 등 비주얼 품질은 오리지널 Xbox 라이브 아케이드 버전 그대로다.

자바스크립트 코드의 실행은 싱글 스레드(single thread) 내에서 일어나는 것으로 볼 수 있다. 최근의 브라우저들은 멀티플 자바스크립트 프로세스 급을 창조하기 위해 API 를 지원하고 있다. 이는 Web Workers [16] 라는 스크립트로 알려져 있는데(추후 기사에서 다시 논함), 메시지를 통해서만 커뮤니케이션할 수 있다. 프로세스들 간에 직접적인 데이터 공유는 허용되지 않는다.

특히 프로그램 실행이 싱글 스레드로 되어 있기 때문에, 자바스크립트 코드가 결과 없이 너무 오래 실행되면 브라우저가 런어웨이 코드를 중지하기를 권하기도 한다. 사용자가 동의하면 코드 실행은 경고 없이 즉각 멈춘다. 사용자가 코드를 중지하지 않더라도 러닝타임에 대한 경고 대화상자가 계속 나타나면 짜증이 날 것이다. 따라서 실행이 오래 걸리는 작업은 작은 인크레멘트(increment)로, 즉 이후 실행될 함수들의 일정을 조정하기 위해 브라우저에서 제공하는 타이머(timer)와 인터벌(intervals)을 사용하는 방법을 권한다.

디버깅과 프로파일링

요즘 나오는 브라우저들은 대부분 자체 내에 디버깅 환경을 제공하고 있다. 디버깅 도구들은 대개 *Development Tools* 메뉴 옵션 하위에 숨어 있다.

디버깅 요소들은 보통 다음 항목들을 제공한다:

- HTML 트리(tree)의 운행 및 점검
- HTTP 요청 사항의 기록 및 점검
- 콘솔의 로깅, 코드 스니핏 실행을 위한 read-eval-print 루프
- 오류 검출기: 브레이크포인트(Breakpoint), 스택 트레이스(Stack trace), 변수 점검을 지원함.

브라우저에서 제공하는 프로파일러들은 보통 콜-그래프 캡처(call-graph capture, 인스트루멘테이션 형태이며, 실행 중 눈에 띄게 과부하를 초래할 수 있음)와 힙 스냅샷(heap snapshots, 스냅샷들 간에 객체 카운터, 사이즈, 참조항목 포함)을

[16] 참조 링크: http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

지원한다. 그러나 이 요소들을 실행했을 때의 품질은 브라우저마다 다르게 나타날 수 있다.

오류 검출기들이 오류를 찾아내려는 해당 코드와 똑같은 프로세스의 같은 자바스크립트 컨텍스트로 실행되는 경우도 있는데, 이렇게 되면 안정성이 떨어진다. 어떤 오류 검출기라도, 코드가 너무 복잡해지거나 점검한 변수들에 대해 너무 많은 정보를 표현하려고 하면 결국 충돌하거나 실패하게 된다. 코드 실행 시 타이머 콜백(timer callback)이 발동되는 것도 문제인데, 이는 특정 오류 검출기를 망가뜨려서 개발자들에게 혼란을 주기도 했다. 그럼에도 불구하고 이 도구들은 충분히 가치가 있으며, 브라우저가 발전함에 따라 문제도 점차 개선되고 있다.

결론

이 글에서는 HTML5 를 위한 고급 게임 개발과 관련하여 개발 환경 및 작업 흐름에 대한 세부 사항들을 전반적으로 살펴보았다. 2 부에서는 HTML5 에서 드러나는 특성 및 게임과 관련된 표준에 대해 보다 자세히 논하려고 한다. 그래픽이나 오디오 같은 게임 개발의 특정 영역에 대한 것은 물론, 다양하고 광범위한 브라우저와 플랫폼에서 최상의 품질과 성능을 구현하기 위해 추천할 만한 방법들도 알아볼 것이다.