Gamasutra.com

게임 개발에 효율적인 XML 파일 판독

Christoph Luerig
2006년 4월 25일

http://www.gamasutra.com/features/20060417/sondergaard_01.shtml

초록

현대 게임 엔진은 게임을 시작하는 동안 거의 모든 서브 모듈에서 다량의 데이터를 판독한다. 일반적인 데이터 포맷 XML 은 이 분야에서 사실상의 표준으로 자리잡고 있다. XML 의 장점 가운데 하나는 용이한 가독성이다. 이러한 포맷에 대해 게임 개발업자의 삶을 보다 편리하게 만드는 표준 툴과 익스포터(exporter)는 다수가 있다. 그러나 대용량 파일에서 이러한 데이터 포맷을 이용할 때는 메모리 소모가 많고 판독 시간과 메모리 세분화가 오래 걸린다는 단점이 있다. 이는 특히 콘솔 게임 개발일 때 더욱 심각하다.

본 논문에서는 표준 DOM(문서 객체 모델)이 아니라 SAX(XML 용 단순 응용 인터페이스)를 이용하여 이러한 단점을 피할 수 있음을 증명하고자 한다. 두 모델은 XML 데이터를 분석하기 위한 접근방식이다. 후자는 게임 개발업자에게 알려지지 않았으며, 처음에는 사용하는 것이 다소 어려운 것으로 보인다. 그러나 이를 제대로 적용하면 개발에 필요한 업무량은 두 접근 모두 거의 동일하며, 제어 흐름만 다를 뿐이라는 점을 입증하고자 한다.

서론

XML [RFC 3076]은 엘리먼트와 속성을 사용자가 개별적으로 정의할 수 있는 HTML 포맷을 일반화한 것이다. 현재, 이 데이터 포맷에는 이종 데이터를 명시해야 하는 다양한 범위의 애플리케이션이 있다. 데이터 포맷은 ASCII 나 Unicode 이기 때문에, 특히 게임

개발업자의 흥미를 자극한다. 이 포맷은 툴을 이용하면 자동으로 생성되지만, 신속하고 간단한 변화를 위하여 수동으로도 편집할 수 있다

한편 XML 파일 처리에 도움을 주기 위하여 다양한 툴과 표준이 개발되었다. 특수 XML 포맷의 구문론과 의미론을 명시하기 위해 여러 표준이 작성되었다. 구형 포맷 규격에 속하는 것으로 DTD 가 있다. DTD 는 특수 XML 구문론을 명시하는 XML 포맷의 특수 확장 포맷이다. 이보다 발전한 접근방식은 SCHEMA[derVlist2002]가 택한 방식이다. SCHEMA 는 XML 파일을 위한 구문론 및 의미론 규격기술 언어이다. SCHEMA 규격 자체가 XML 로 작성된다. 의미론을 명시할 수 있는 능력은 기본값이나 제한 사항과 같은 내용을 포함한다. 여기에서 XMLSpy[XMLSpy]와 같이 사용자가 파일을 편집하는데 도움이 되도록 Schema 규격을 처리할 수 있는 XML 에디터가 있다. 데이터베이스 프로그램은 대부분 XML 포맷으로 데이터를 익스포트할 수 있는 가능성을 제공한다. 또 하나의 흥미로운 XML 기술은 XSLT [W3CR]이다. XSLT 는 XML 포맷의 데이터를 다른 데이터로 변형할 수 있는 문법 변형 언어이다. 물론 이것은 게임 개발의 관점에서 대단히 흥미로운 언어이다. 개발업자는 게임 설계가 완성되면서 게임 데이터의 포맷 변화를 극복해야 하는 경우가 많기 때문이다.

이와 같이 XML 이 제공하는 가능성을 간파하여, 본 연구진은 가장 간단하면서도 효율적인 방식으로 XML 데이터를 게임에 도입할 수 있는 방법을 집중적으로 다루고자 한다. 간단한 완구 표본으로, 불필요한 장비를 모두 제거한 자동차를 나타내는 아래의 XML 규격을 검토해 보도록 하겠다.

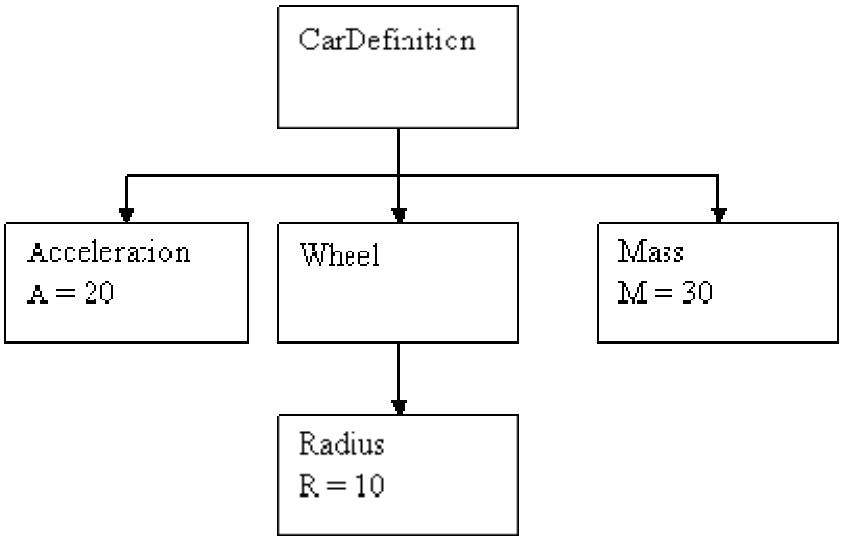
```
<?xml version="1.0" encoding="utf-8"?>
<CarDefinition>
  <!--This is a simple xml test file to check the car definintion
stuff-->
  <Acceleration a="20.0" />
  <Wheel>
    <Radius r="10.0" />
  </Wheel>
  <Mass m="30.0" />
</CarDefinition>
```

이는 옵션 분석에 대한 아래 설명에서 예제로 이용될 것이다. 단순화를 위하여 제시된 예제에서 <Radius>10.0</Radius>와 같은 텍스트 구성소는 사용하지 않는다. 제시된 접근법을 확장하여 이러한 구성소를 수용하는 것은 간단하다.

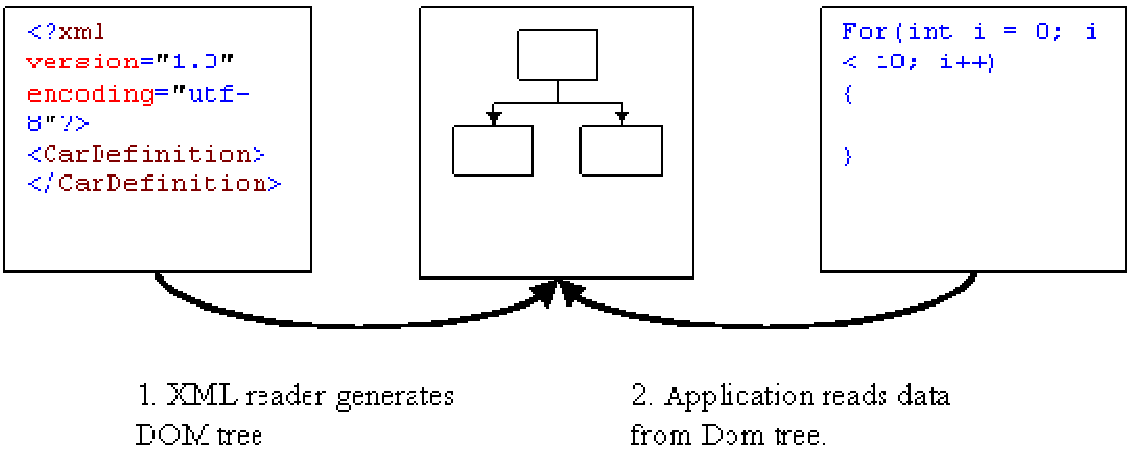
XML 파일 판독을 위하여 퍼블릭 도메인 파서가 존재한다. 이러한 파서는 두 범주로 나뉜다. 게임 개발업자에게 가장 공통된 파서는 DOM(문서 객체 모델)이다. DOM 이 있으면 파서는 완전한 XML 파일을 판독하여 메모리 내에 해당 파일의 계층적 표현을 생성하며, 이는 애플리케이션으로 분석할 수 있다. SAX(XML 용 간이 API)는 XML 파일을 스트림으로 판독하여, 사용자 애플리케이션에 의하여 처리해야 하는 이벤트를 생성한다. 아래 항목에서 두 모델을 보다 상세히 분석 및 비교해보자.

DOM 모델

문서 객체 모델은 가장 보편적이다. 이러한 XML 파서는 메인 메모리에서 애플리케이션에 의하여 처리될 수 있는 문서의 나무구조를 생성한다. 처리가 완료되면 나무는 메모리에서 제거된다. 위에서 설명한 모델은 메모리에서 다음과 같이 DOM 나무로 표현할 수 있다.



파서는 파일에서 메모리로 문서를 판독한 이후 필요한 데이터를 추출하기 위하여 이와 같이 애플리케이션을 위한 구조 내에서 탐색할 수 있는 방법을 제공한다. 이를 위한 시스템의 제어 흐름은 아래와 같다.



DOM 나무 판독기의 구현을 위하여 TinyXML [SFTXml]을 이용한다. 이를 위해서는 소수의 라인만 초기화하면 된다. 이 경우 판독기를 초기화하고 프로세싱을 시작하기 위한 코드는 아래와 같다.

```
TiXmlDocument doc("CarDefinition.xml");
doc.LoadFile();

TiXmlHandle hDoc(&doc);
TiXmlHandle carHeader = hDoc.FirstChildElement("CarDefinition");

Car testCar;
testCar.Parse(carHeader);
```

“LoadFile” 명령과 함께 완전한 나무가 메모리에 생성된다. 모든 추가 콜은 나무에서 데이터를 추출하기 위한 명령이다. 이러한 추출은 데이터가 필요한 객체에 의하여 수행된다. 모든 XML 엘리먼트에 대해 추가 정보를 요청할 수 있다. 여기서는 자동차 정의 블록의 추출을 위하여 이를 수행한다. 자동차 선언은 다음과 같다.

```
class Car
{
```

```
public:
  void Parse(TiXmlHandle handle);

protected:
  Wheel m_wheel;
  float m_acceleration;
};
```

파스(Parse) 방식은 멤버 변수에 채울 데이터를 나무에서 추출하는 작업을 담당한다. 이러한 방식의 구현은 아래와 같이 간단하다.

```
void Car::Parse(TiXmlHandle handle)
{
  TiXmlElement* section;

  section = handle.FirstChildElement( "Acceleration" ).Element();
  section->QueryFloatAttribute( "a" , &m_acceleration);

  section = handle.FirstChildElement( "Mass" ).Element();
  section->QueryFloatAttribute( "m" , &m_mass);

  m_wheel.Parse(handle.FirstChildElement( "Wheel" ));
}
```

이 코드에서 구성소 Handle 과 포인터의 사용을 확인할 수 있다. 핸들은 NULL 용인 포인터로 DOM 나무 구조 내에서의 탐색을 간편하게 만든다. 위의 예제와 마찬가지로 가속화 엘리먼트에 대해 질문하는 구조가 존재하지 않을 경우, 질문 명령은 NULL 포인터를 리턴시킬 것이다. 이러한 포인터에서 이루어지는 후속 연산은 예외가 될 것이다. 이 경우 핸들은 용인되며, 존재하지 않는 엘리먼트에 대해 핸들에서 추가 핸들을 질문할 수 있다. 데이터에 실제로 접속하는 최종 단계가 되어야 NULL 포인터가 명백해진다. 이와 같은 방법으로 구체적인 엘리먼트가 체크를 코드 전체로 확산시켜야 할 필요성을 제거하는 장소로 누락 데이터에 대한 오류 체크를 국한시킬 수 있다. 가독성을 위하여 오류 체크는 본 샘플에서 누락하였지만, 실제 제작 코드에서는 당연히 포함시켜야 한다. 핸들은 TinyXML 구현에서 대다수 유형의 포인터를 덮을 수 있다.

자동차 자체는 이제 휠 정보가 담겨 있는 아동 엘리먼트의 포인터를 인계하고 휠에 대한 제어를 인계하여 DOM 나무에서 데이터를 추출한다.

```
class Wheel
{
public:
  void Parse(TiXmlHandle handle);
protected:
  float m_radius;
};

void Wheel::Parse(TiXmlHandle handle)
{
  TiXmlElement* section;
  section = handle.FirstChildElement("Radius").Element();
  section->QueryFloatAttribute("r",&m_radius);
}
```

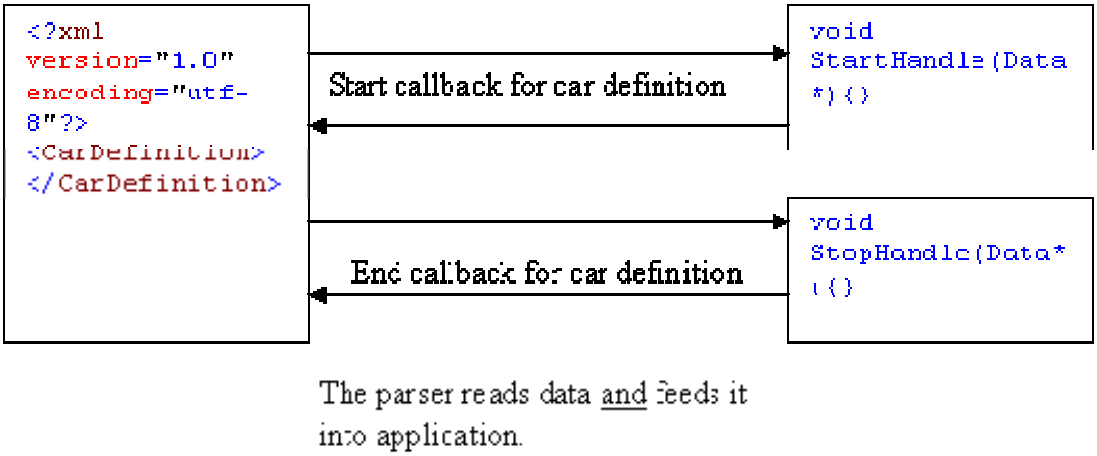
이 작업이 끝나면 자동차에 대한 완전한 정보가 메모리에 포함되며, DOM 나무는 더 이상 사용되지 않기 때문에 파기할 수 있다.

아래 항목에서 대안을 논의한 다음, 이러한 접근방식의 장단점을 상세히 검토하겠다.

SAX 모델

앞서 언급한 바와 같이 SAX 모델은 “XML 용 간이 API”를 뜻한다. 이 모델의 목적은 특정 상황에 맞게 쉽게 조절할 수 있는 경량의 API 를 제공하는 것이다. 또한 이 모델은 자원에 대한 수요가 낮다. XAX 파서의 개념은 문서의 DOM 나무를 문서에 생성하는 것이 아니라 XML 파일을 스트림하고 특정 엘리먼트를 만날 때 특정한 콜백 기능을 사용하는 것이다. 애플리케이션은 이러한 콜백에 반응하여 정확한 위치에 데이터를 저장해야 한다. 반응을 수행할 경우, 제어는 파서로 리턴된다.

DOM 나무와 SAX 모델의 중요한 차이는 SAX 를 이용한 제어는 이 경우 데이터를 시스템에 공급하는 파서의 지배를 받는다. 그러나 DOM 모델의 경우 객체가 직접 데이터를 추출할 수 있기 때문에, 제어 흐름이 전혀 다르다. 위의 예제에 대한 제어 흐름은 다음과 같다.



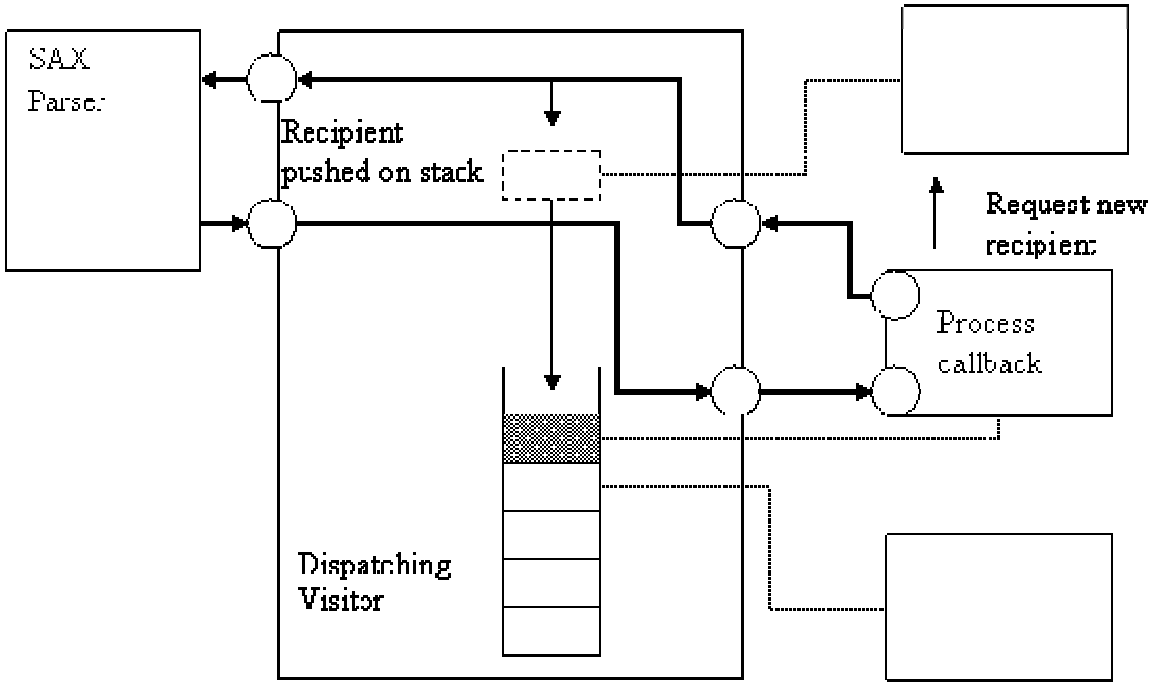
여기에서 증명에 사용되는 SAX 파서는 eXpat [SFX]이다. 이 파서는 SourceForge 에서 구할 수 있다. eXpat 를 등록하려면 콜백 핸들러 3 개가 필요하다. 첫 번째 핸들러는 엘리먼트 개시용이다. 이 핸들러는 속성에 대한 정보 일체를 제공한다. 두 번째 핸들러는 구체적인 정보를 제공하지 않지만 엘리먼트의 종료를 표시한다. 세 번째 유형의 콜백은 엘리먼트 내에 텍스트를 제공한다. 단순화를 위하여 여기서는 이 기능을 사용하지 않지만, 그 구현은 간단하다. eXpat 은 파일에서 직접 판독하지 않고 메모리에서 한다. 전체 파일을 메모리로 로드할 필요가 없다. 파서에 의해 처리되는 섹션 내에서 판독이 가능하다. 처리를 하는 동안 앞에서 언급한 C-스타일 콜백이 요청되며, 데이터가 애플리케이션에 공급된다. 모든 콜백은 임의 포인터로 맨 앞줄에 명시될 수 있는 사용자 데이터 입력을 제공할 수 있다.

이러한 접근방식은 일견 다소 신경이 쓰이고 게임 개발에 적합하지 않은 것으로 보이며, 거의 사용되지 않는 이유도 이 때문일 것이다. 한편으로 파일에서 데이터를 확보기를 원하는 대규모 객체 계층으로 구성되는 내부 게임 구조가 있다. 다른 한편으로는 객체 계층의 제어를 받지 않는 방식으로 데이터를 전달하고자 하는 SAX 파서의 C-스타일 콜백이 있다. 내부의 자가 제조 데이터 포맷을 DOM 나무 파서로 바꾸는 것이 SAX 파서로 바꾸는 것보다 일견 훨씬 쉬워 보인다.

하지만 자세히 관찰하면 두 방식의 차이는 그렇게 크지 않다. 문제를 해결하고 분석 과정을 동일하게 보이도록 만든다는 개념은 두 부분으로 구분된다. 첫 번째 개념은 콜백을 처리하고 데이터를 목적 객체에 공급하는 방문객을 이용하는 것이다. 이 방문객은 콜백에서 제공하는 정보를 게임의 객체 계층 내에 있는 특정 객체와 결합할 책임이 있다. 두 번째 개념은 객체 내에 있는 데이터의 처리를 말한다. DOM 나무 모델에서, 객체는 데이터를 추출하기 위하여 나무에 몇 가지 콜을 수행한다. 이 작업은 파스 기능으로 수행한다.

위에서 제안하는 솔루션에서, 데이터 입력 기능은 위에서 언급한 방문객이 여러 차례 요청한다. 모든 콜은 데이터와 값 자체에 대한 설명을 담고 있다. “if”나 “switch” 단계에서, 데이터의 설명이 확인되고 데이터는 정확한 목적에 맞게 작성된다. 이러한 과정은 아래 단락에서 자세히 설명된다.

프로세스 내에서 데이터를 수신하는 첫 번째 사례는 방문객이다. 모든 콜백은 임의 포인터로 제공될 수 있기 때문에, 방문객이 데이터를 처리하도록 포인터를 설정한다. 콜백 자체는 방문객 내에서 해당 방법을 요청하는 C 기능이다. 방문객의 주요 임무는 파서 콜백에서 나오는 정보를 목적 객체에 발송하는 것이다. 이러한 임무를 완수하기 위하여, 수신자 스택을 포함하고 있다. 데이터는 스택 상단에 있는 수신자에게 공급된다. 새 엘리먼트를 시작할 때마다, 새로운 수신자가 스택에 추가되고, 엘리먼트가 종료되면, 수신자는 제거된다. DOM 나무에서 데이터를 추출하기 위하여 다양한 파스 기능이 요청될 경우 방문객 내에 포함되는 이러한 스택 계층은 콜 스택 계층에 해당한다. 시작 엘리먼트 형태로 데이터를 발송 방문객에게서 수신자에게로 공급할 때마다, 수신자는 다가올 데이터를 위하여 포인터를 신규 수신자에게 리턴시킬 기회가 있다. 이 경우, 이러한 내부 객체에 대한 포인터는 스택에 추가된다. 그밖의 경우에는 기존의 객체와 동일한 포인터를 스택에 추가한다. 엘리먼트가 종료되면, 수신자는 스택에서 제거되며 추가 데이터는 최초 수신자에게 전달된다. 이러한 프로세스는 아래와 같이 시각화할 수 있다.



이 예제에서, 콜은 새 수신자를 스택에 추가하도록 요청하는 목적 객체에 발송된다. 콜백을 처리할 경우, 추가 명령을 초기화한 객체에 대해 엘리먼트의 종료를 만날 때까지 모든 후속 콜백은 새 수신자에게 제공한다. 그런 다음 제거 명령을 실행하고, 모든 콜백은 다시 구형 객체에 전달된다.

모든 콜을 발송하는 방문객은 이 접근방식의 중심 개념이다. 발송자 등급의 표현은 다음과 같다.

```
class ParseDispatcher
{
public:
  void ParseFile(const char* name, Receiver* startObject);

private:
  stack<Receiver*> m_stackOfReceivers;
  static void XMLCALL start(void *data, const char *el,
  const char **attr);
  static void XMLCALL end(void *data, const char *el);
};
```

수신자는 정보를 얻고 발송될 수 있는 모든 등급의 추상적 등급이다. 메인 루틴은 파일을 분석하여 최초 수신자에게 인계한다. 이러한 객체는 최초로 정보를 수신한다. 이와 같이 고정된 방식은 엘리멘트의 시작과 종료를 표시하기 위하여 eXpat 에 등록되는 두 개의 C 방식이다. 추상적 수신자 등급의 표현은 다음과 같다.

```
class Receiver
{
public:
  virtual Receiver* ProcessData(const char *el, const char **attr)=0;
};
```

ProcessData 방식은 엘리먼트의 명칭과 속성 명칭 및 가치 쌍의 배열을 얻는다. 그런 다음 후속 ProcessData 콜이 동일한 객체로 향할 경우 새 수신자나 NULL 포인트를 리턴한다.

ParserDispatcher 의 구현은 다음과 같다.

```
void XMLCALL ParseDispatcher::start(void *data, const char *el,
const char **attr)
{
  ParseDispatcher* processor = ((ParseDispatcher*)data);
  Receiver* candidate = processor->m_stackOfReceivers.top();
  Receiver* followUp = candidate->ProcessData(el,attr);

  if (followUp)
    processor->m_stackOfReceivers.push(followUp);
  else
    processor->m_stackOfReceivers.push(candidate);
}

void XMLCALL ParseDispatcher::end(void *data, const char *el)
{
  ((ParseDispatcher*)data)->m_stackOfReceivers.pop();
}

void ParseDispatcher::ParseFile(const char* name, Receiver*
startObject)
{
  m_stackOfReceivers.push(startObject);

  char buf[BUFSIZ];
  XML_Parser parser = XML_ParserCreate(NULL);
  XML_SetUserData(parser, this);
  int done;
  FILE* input = fopen(name, "r");
  XML_SetElementHandler(parser, start, end);
  do
  {
    size_t len = fread(buf, 1, sizeof(buf), input);
    done = len < sizeof(buf);
    XML_Parse(parser, buf, len, done);
  }
while (!done);
  XML_ParserFree(parser);
```

```
m_stackOfReceivers.pop();
}
```

파스 방식은 파서를 생성하고, “XML_SetUserData”에 있는 임의 포인트를 통하여 직접 등록하며, 나중에 고정 방식에 레퍼런스될 수 있다. 이렇게 하여 동시에 여러 파서를 실행할 수 있다. 파일을 오픈하고 핸들러를 등록한다. 최초 수신 객체는 스택에 추가된다. 한편 루프 데이터는 큰 덩어리로 판독하여 저장 요건을 제한한다.

고정된 시작 방식에서, 사용자 데이터 포인터를 통하여 스택을 추출하고 콜을 발송한다. 리턴값 수신 여부에 따라, 새 객체나 동일한 객체를 스택에 추가한다. 엘레먼트 시작과 종료 사이에 나타나는 텍스트를 처리하고 싶다면 스택 자체를 수정하지 않고 스택에 있는 동일한 상위 객체에 이를 발송하면 된다. 최종 명령은 최상위 포인터를 스택에서 다시 제거하며, 이는 구형 객체로 리턴한다.

이것은 물론 오류 처리나 이용의 편리성 현안이 없는 최소 구현이다.

두 번째 부분은 휠과 자동차의 표현 및 구현이다. 두 부분은 적절한 작업을 위하여 수신자 등급에서 유도해야 한다. 이러한 등급의 표현은 다음과 같다.

```
class Wheel : public Receiver
{
public:
  virtual Receiver* ProcessData(const char *el, const char **attr);

protected:
  float m_radius;
};

class Car : public Receiver
{
public:
  virtual Receiver* ProcessData(const char *el, const char **attr);

protected:
  Wheel m_wheel;
  float m_acceleration;
  float m_mass;
};
```

위의 표현은 수신자로부터 유도하고 “Parse” 대신 “ProcessData” 방식을 이용한다는 사실을 제외하면, DOM 파싱 모델의 표현과 유사하다. 표현이 유사하다고 해도, 제어 흐름은 다르다는 점을 염두에 두어야 한다. 객체의 “ProcessData” 방식은 여러 차례 요청될 수 있고 한 객체의 “ProcessData”는 다른 객체의 “ProcessData”를 요청할 수 없고 포인터를 새 수신자에게 리턴시킬 뿐이다.

SAX 파서를 이용한 자동차의 구현은 다음과 같다.

```
Receiver* Car::ProcessData(const char *el, const char **attr)
{
  if ((strcmp(el,"Wheel")==0))
    return (&m_wheel);

  if ((strcmp(el,"Acceleration")==0)&&(strcmp(attr[0],"a")==0))
    sscanf(attr[1],"%f",&m_acceleration);
  if ((strcmp(el,"Mass")==0)&&(strcmp(attr[0],"m")==0))
  sscanf(attr[1],"%f",&m_mass);

  return (NULL);
}
```

다양한 속성에 대한 정보를 얻는 것은 스트링 비교 단계이다. 엘레먼트 명칭이 “휠”일 경우, 휠은 데이터를 처리해야 할 다음 객체이기 때문에 휠에 포인터를 리턴한다. 그밖의 모든 경우(“Acceleration”, “Mass”), 필요한 정보를 추출하여 해당 멤버 변수에 이를 저장한다. 또한, 휠을 마주치지 않을 경우 동일한 객체에 정보를 제공하고 싶기 때문에 NULL 포인터를 리턴한다. 휠의 경우에도 구현은 간단하다.

```
Receiver* Wheel::ProcessData(const char *el, const char **attr)
{
  if ((strcmp(el,"Radius")==0)&&(strcmp(attr[0],"r")==0))
    sscanf(attr[1],"%f",&m_radius);
  return (NULL);
}
```

이러한 구현은 DOM 나무 버전의 구현보다 크게 복잡하지 않다는 것을 확인할 수 있다. 그러나 두 버전 모두 장단점이 있다. SAX 모델의 장점은 DOM 의 장점보다 많다. 아래에서 이러한 장단점을 논의하고자 한다.

비교

앞부분에서 DOM 나무 파서와 일반적으로 널리 알려지지 않은 SAX 파서를 이용하여 XML 데이터를 게임 엔진에 판독하는 방법을 예시하였다. 여기에서는 두 접근법을 비교하여 상황에 따라 적합한 방식을 분석하고자 한다.

SAX 파서를 이용하는 구현은 정보를 수집하는 방문객으로 객체를 이동하는 발송자가 필요하기 때문에 다소 복잡하다. 한편, 이 방법은 SAX 파서의 규격을 판독한 이후에는 생각한 것만큼 복잡하지 않다. 전체 프로젝트에 대해 발송자를 한 번만 실행하면 된다. SAX 파서에 포함되는 스트링 컴페어 단계는 DOM 버전에 비해 프로그램의 가독성을 저하시킬 수 있다. 결국 코드 가독성을 높이기 위해서는 판독 과정에 일부 매크로를 이용하는 것이 합리적일 것이다

SAX 파서가 안고 있는 중요한 문제는 “ProcessData” 방식을 구현하는 사용자가 다양한 엘리먼트가 처리되는 순서를 통제할 방법이 없다는 점이다. 프로세싱 순서는 XML 파일의 순서에만 의존한다. DOM 나무 파서의 경우, 객체는 DOM 나무의 임의 위치에서 데이터를 임의적인 순서로 고를 수 있다. 게임의 맥락에서, 한 엘리먼트의 데이터가 앞으로 올 데이터가 사용할 일부 메모리를 보존할 필요가 있을 경우 이는 특히 중요하다. SAX 파서를 사용할 때는 XML 파일 내부의 순서가 정확한지 확인해야 한다. 이것은 DOM 트리 파서에서 SAX 파서로 이동할 때 문제가 될 수 있다.

SAX 접근방식의 중요한 단점은 성능이 문제될 때 명백해진다. SAX 모델은 메모리 소모와 실행 속도에 관한 한 우수한다. 실행 속도의 증가는 데이터를 엔진에 입력할 때 1 회의 패스만 사용된다는 사실에서 기인한다. 데이터는 프로그램에 직접 공급되기 때문에 로드된 DOM 나무를 디컴파일하는데 필요한 노력은 생략된다. 스트링 컴페어 단계는 일견 비효율적으로 보인다. 여기에서, DOM 나무에서 엘리먼트를 얻는데 사용되는 방법은 내부적으로 스트링 컴페어를 이용한다는 점을 유념해야 한다. 성능 전반에 걸쳐 SAX 파서의 주요 장점은 메모리 소모와 관련될 경우이다. SAX 파서에 필요한 메모리는 대부분 XML 파일을 판독할 때 보존되는 버퍼이다. DOM 나무 파서는 나무 전체를 저장할 수 있는 메모리가 필요하다. 이 나무는 애플리케이션이 파스 나무에서 데이터를 확보하여 프로그램에 공급한 이후에는 필요가 없다. 그러나 이러한 나무는 로딩을 하는 동안 피크 메모리 소모를 크게 늘린다. SAX 파서가 피할 수 있는 심각한 문제는 메모리 세분화이다. DOM 트리의 다양한 엘리먼트 할당으로 인하여 다양한 엘리먼트를 여러 차례 소규모로 할당하게 될 수 있다. 파서는 먼저 DOM 나무를 판독하여 할당을 수행한다. 그 이후 나무를 이용하는 객체가 추가 할당을 수행한다. 마지막으로, DOM 나무의 메모리가 제거되면 메모리 블록에 큰 구멍이 생긴다. 필자는 파싱 파일에서 메모리 내부에 나무

구조를 생성하는 전용 데이터 포맷에서 이러한 현상을 체험하였다. 두 접근방식의 장점을 요약하면 다음과 같다.

SAX 의 장점

- 낮은 메모리 소모
- 낮은 메모리 세분화
- 빠른 실행 속도

DOM 의 장점

- 이해의 편리성
- 구현의 용이함
- XML 파일 내 파일의 순서가 무관함

한 기술의 장점은 다른 기술의 단점이며 그 역도 마찬가지이다.

최종적으로 SAX 파서 적용은 성능이 문제가 되거나 앞으로 문제가 될 경우 유용하다고 말할 수 있다. DOM 에서 SAX 로 변경하려면 데이터가 각 객체에 정확한 순서로 공급되는지 확인하기 위하여 코드 작업을 다시 해야 한다. 성능은 게임을 개발하는 동안 항상 주요 현안이 되기 때문에, SAX 파서를 사용하는 것은 반드시 필요해 보인다.

결론

본 논문에서는 XML 파일을 분석하는 2 개의 기본 접근방식을 설명하였다. 널리 적용되고 있는 DOM 나무 모델 외에, SAX 파서 응용 방법을 설명하였다. SAX 파서 응용은 일견 복잡해 보일 수 있다. 익숙하지 않은 SAX 파서의 제어 흐름을 채택하는 데서 발생하는 어려움을 극복하기 위하여, 방문객 패턴을 토대로 데이터를 애플리케이션으로 판독하는 작업을 쉽게 만드는 개념을 소개하였다.

면밀한 검토를 거친 끝에, SAX 파서는 특히 다량의 데이터를 판독해야 하는 게임에 사용하면 유용한 성능상의 장점이 많다는 것을 알 수 있다. 콘솔 개발을 할 때는 메모리 소모와 세분화 문제에 직면할 수 있는데, 이러한 접근법이 적합한 것으로 보인다.

게임의 최종 마스터에서는 XML 데이터가 사용되지 않지만, 게임을 개발하는 동안 SAX 파서를 사용하면 소요시간을 크게 줄일 수 있다. 또한 제한된 메모리로 개발 키트를 사용할 경우에도 필요할 수 있다.

참고 문헌

[RFC 3076] http://rfc.net/rfc3076.html
[derVlist2002] Eric van der Vlist: *XML Schema*; 2002 O'Reilly Associates
[XMLSpy] http://www.altova.com/products_ide.html
[W3CR] http://www.w3.org/TR/xslt

[SFTXml] http://sourceforge.net/projects/tinyxml

[SFX] http://sourceforge.net/projects/expat