한국게임학회 논문지
제6권 제1호 2006년 3월

Python 언어를 위한 Direct3D 인터페이스 모듈 개발

이강성
광운대학교 교양학부 (컴퓨터공학 전공)
gslee@mail.kw.ac.kr

Direct3D Interface Module Development for Python Language

Gang-Seong Lee
Kwangwoon Univ.

요약

본 논문에서는 게임과 3D 모델링에 많이 이용되는 라이브러리, MS사의 Direct3D를 파이썬(Python)에서 사용할 수 있도록 하는 인터페이스 구현 기술에 대하여 논한다. 현재 주로 MS에서 출시되는 언어군에서만 DirectX를 사용할 수 있지만, 본 논문에서 제시한 방법을 이용하면 파이썬 뿐 아니라 다양한 언어에 DirectX 라이브러리를 인터페이스를 제작할 수 있고, 다양한 언어를 쓰는 사용자들에 의해 이 라이브러리의 활용폭을 넓힐 수 있을 것이다. 본 논문에서는 인터페이스 모듈 작성 기법을 설명하고 파이썬을 이용했을 때의 장점과 단점들을 설명한다.

Abstract

This paper describes the implementation of Direct3D interface library for Python language. DirectX is the most popular library used for 3D games and 3D modelings. However, softwares which use the library can only be developed in the environments provided by Microsoft like Visual Studios and .NET framework. The interface module for Python, this paper presents, will extend the coverage of the useful library DirectX to a language which is not fully supported by Microsoft. The interface techniques described here can be a guide to develop interface modules for other languages too, which make their language more powerful and extensible. This paper describes the implementation techniques to develop the interface module for Python, advantages and disadvantages.

Keyword : Python, DirectX, Direct3D, dxPython

1. 서론

데스크탑 PC에서 3D 게임과 모델링에 가장 많이 사용되는 라이브러리는 OpenGL과 Direct3D이다. 최근 수년간 MS는 DirectX의 성능을 향상시키기 위해서 많은 자금과 노력을 투자하였고, 그 결과 지금은 대부분의 PC 3D 게임에서 DirectX를 이용하고 있다. OpenGL에 비해 DirectX는 몇 가지 제약점을 가지고 있는데, OpenGL이 다양한 플랫폼을 지원하는 반면, 운영체제에 크게 의존하는 DirectX는 MS 윈도우 환경에서만 수행된다. 두 번째로 OpenGL은 각종 플랫폼의 다양한 언어를 지원하고 있는 반면, DirectX는 MS에서 나온 언어 이외에서는 거의 지원되고 있지 않다. 이러한 Direct3D의 파이썬 래퍼(wrapper)개발을 통해 MS가 지원하지 않는 언어에서도 자유롭게 라이브러리를 이용할 수 있고, C/C++가 지원하지 않는 기능을 보완적으로 추가함으로써 더욱 더 자유롭고 유용하게 라이브러리를 사용할 수

있게 된다.

현재 Ruby라는 언어를 위한 dxRuby 모듈이 나와있긴 하지만[2], 이것은 제한적으로 2D만 지원하고 있다. Java는 현재 Java 3D란 이름으로 OpenGL과 DirectX를 지원하고 있지만 DX8.0을 지원하고 있는 상태이다[3].

파이썬은 깔끔한 문법과 강력하고 풍부한 자료구조 및 라이브러리를 지원하는 객체지향 고급언어이다. 파이썬은 고수준의 자료형을 가지고 있으며 동적인 자료형(dynamic typing)을 지원하여 C나 Java와 같은 정적 자료형을 지원하는 언어보다는 훨씬 더 유연하고 간결한 코드를 작성할 수 있게 해준다. 파이썬의 C에 비해 1/5~1/10 정도의 코드만 요구되는 매우 간결하면서도 이해하기 쉬운 언어이다. 파이썬은 자바와 같이 바이트코드로 컴파일 되어 실행되는 컴파일 언어이지만 행단위의 실행이 가능한 인터프리터 언어로도 동작된다[1].

반면에 C/C++는 빠른 실행속도를 내는 코드생성과 저수준의 시스템 프로그램이 가능하다는 것인데 C/C++의 빠른 실행의 장점과 파이썬의 유연한 구조의 이점을 결합함으로써 우리는 새로운 개발 환경을 얻게 되며 다음과 같은 장점을 얻을 수 있게 된다. 첫째로 동적인 언어가 지원해 주는 유연성을 확보할 수 있다. 두 번째로 대화적(interactive)으로 Direct3D 라이브러리를 사용할 수 있다. 세 번째로 특별한 처리 엔진 설계 없이도 스크립팅(scripting)이 가능하다. 네 번째로 알고리즘의 디버깅이 쉬워진다. 다섯 번째로 게임 프로그래밍 분야에서 구현하기 어려운 자동화된 단위 테스팅(unit testing)을 쉽게 처리할 수 있다. 또한 본격적인 작업을 수행하기 전에 그 가능성을 미리 확인해 보는 프로토타이핑용으로도 활용하기 좋다.

C/C++ 라이브러리 혹은 C/C++ 코드를 파이썬(혹은 다른 스크립트 언어)에 붙이는 방법은 몇 가지가 있는데, 첫째 방법은 파이썬의 C/C++ 확장 규격에 맞게 직접 C코드를 작성하거나 자동화된 도구를 이용할 수 있다. C코드를 직접 작성하는 것은 Direct3D 라이브러리의 함수가 너무 많아 현실적으로 구현에 어려움이 따른다. 자동화된 도구를 이용하는 방법을 이용하는 것이 좋은데, 그 방법으로는 SWIG를 이용하거나 Boost.Python을 이용할 수 있다.

SWIG(Simplified Wrapper and Interface Generator)는 C나 C++ 혹은 Objective-C로 쓰여진 이진 라이브러리 혹은 소스 프로그램을 PHP, C#, Perl, Tcl/tk, Python, Lua, CLisp, Guile, Ruby, Mzscheme와 같은 언어로 연결시키는 인터페이스 컴파일러이다[4-6]. Boost.Python은 주로 C++ 라이브러리의 파이썬 인터페이스 모듈 제작을 위해서 사용된다. C++를 주로 지원한다는 점에서 SWIG에 대한 이점이 있겠지만 아직은 다양한 언어를 지원하지는 못한다[6]. 본 연구에서는 SWIG를 이용하여 인터페이스 모듈을 구현하였다. SWIG는 다양한 언어를 지원하기 때문에, 인터페이스 모듈을 같은 방법으로 다른 언어에도 적용시키기가 용이하기 때문이다.

2. SWIG를 이용한 래퍼 모듈의 작성

2.1 래퍼 모듈 작성의 원리

SWIG는 ANSI C/C++의 선언을 스크립트 언어 인터페이스로 만들어주는 컴파일러이다. SWIG의 인터페이스 파일을 잘 작성해주면 파이썬 코드와 C 인터페이스 코드를 자동으로 생성해주며, 컴파일만 하면 즉시 사용가능 하게 된다. SWIG의 간단한 동작 원리를 알아보자.

파이썬에서의 함수 호출을 어떻게 C 함수 호출로 연결할 수 있을까? 또, C 함수에서 리턴한 결과 값이 어떻게 파이썬 자료형으로 변환되어 파이썬 함수에서 리턴될까? 이것은 중간층(glue layer)에 있는 래퍼(wrapper)함수 덕분에 가능하다. 예를 들어, 다음과 같은 팩토리알을 계산하는 fact() 함수를 보자.

```
// FILE : fact.h
int fact(int n);

// FILE : fact.c
#include "fact.h"
int fact(int n) {
   if (n <= 1) return 1;
   else return n*fact(n-1);
}
```

이 함수는 SWIG가 생성하는 Python/C API를 이용하는 다음의 래퍼 함수 wrap_fact()에 의해서 호출된다.

```
// FILE : fact_wrap.c, code generated by SWIG
```

```
PyObject *wrap_fact(PyObject *self, PyObject *args) {
  int n, result;
  if (!PyArg_ParseTuple(args,"i:fact",&n)) // 파이썬
  함수 호출시 전달한 자료형을 C 자료형으로 변환한다
  return NULL;
  result = fact(n); // 이 부분에서 C 함수를 호출한다
  return Py_BuildValue("i",result); // C 자료형을 파
  이썬 자료형으로 변환해서 리턴한다.
}
```

파이썬이 전달한 자료형은 PyArg_ParseTuple에 의해서 C 자료형으로 변환되고(n), C 함수가 호출되며, C 함수가 리턴한 결과를 Py_BuildValue 함수를 이용하여 다시 파이썬 자료형으로 변환시키는 과정을 거친다. fact_wrap.c의 생성은 SWIG 인터페이스 파일만 선언되어 있으면 SWIG에 의해서 자동으로 생성된다. SWIG 인터페이스 파일 fact.i의 간단한 작성 예는 다음과 같다.

```
%module fact    // 모듈 이름
%{
#include "fact.h"// fact_wrap.c 에 포함될 헤더 코드
%}
%include "fact.h" // export 되는 클래스, 함수, 변수들
```

이렇게 swig 파일이 준비되었으면, 다음과 같이 SWIG를 실행시킨다. SWIG는 실행 결과로 fact_wrap.c 와 fact.py 파일을 생성시키는데, fact_wrap.c 파일은 C 인터페이스 파일이고, fact.py 파일은 파이썬에서 fact_wrap.c 모듈을 호출하게 해주는 파일이다.

```
$ swig -python fact.i
```

이제 파일이 생성되었으면 fact_wrap.c 를 컴파일 해서 파이썬 C 모듈을 만들 순서이다. 다음과 같이 setup.py를 준비한다.

```
# setup.py
from distutils.core import setup, Extension
setup(ext_modules=[Extension('_fact',
['fact_wrap.c', 'fact.c'])])
```

그리고 python setup.py build 명령을 내린다. 그러면 _fact.pyd 라는 파이썬 DLL 파일이 생성된다. 파이썬에서 fact를 실행시킨 예를 다음에 보인다.

```
>>> import fact
>>> fact.fact(4)
24
```

2.2 변환된 Direct3D 함수들

실제로 Direct3D 라이브러리의 모든 함수들이 위와 같이 단순하게 변환되지는 않지만, 기본적인 원리는 동일하다. 다음의 헤더 파일에 있는 struct, class, 함수, 심볼들이 대부분 변환되었다. 1700개 이상의 struct와 class가 변환되었고 3000개 이상의 함수 심볼들이 변환되었다.

2.3 대화적인 Direct3D 사용의 예

다음에 Direct3D를 사용하는 실제적인 간단한 대화식 실행의 예를 보인다. 파이썬의 장점을 쉽게 파악할 수 있는 부분이다. 행렬 값을 사원수로 변환하는 예이다.

d3d9.h	d3dx9math.h
d3d9caps.h	d3dx9mesh.h
d3d9types.h	d3dx9shader.h
d3dfont.h	d3dx9shape.h
d3dutil.h	d3dx9tex.h
d3dx9anim.h	d3dx9xof.h
d3dx9core.h	dxfile.h
d3dx9effect.h	dxutil.h

```
>>> from dx.baselib import *
>>> m=D3DXMATRIX(0.215193,-0.969920, 0.113783,
    0.000000,0.976260,0.216600,-0.000000,0.000000,
    -0.024646,0.111082,0.993506,0.000000,-
    88.995270,-350.280182,247.734116,1.000000)
>>> D3DXQuaternionRotationMatrix(m)
<D3DXQUATERNION>
  <x>-0.0356640815735</x>
  <y>-0.0444441325963</y>
  <z>-0.624842226505</z>
  <w>0.778668582439</w>
</D3DXQUATERNION>
```

이러한 기능을 C/C++로 테스트하려면 프로젝트를 개설하

고 형식에 맞는 파일과 함수를 만들어서 컴파일, 실행해야 하는 번거로운 과정을 거쳐야 하지만, 파이썬에서는 단 2줄의 코드로 즉각적인 응답을 얻어낼 수 있다.

2.4 파이썬 인터페이스 모듈의 구조

앞서와 같은 방법으로 만들어진 파이썬 인터페이스 모듈은 dx라는 패키지(package) 이름을 갖으며, 다음과 같은 파일들로 구성된다.

```
dx
├── baselib.py
├── color.py
├── customvertex.py
├── _customvertex.pyd
├── d3d9.py
└── _d3d9.pyd
```

baselib.py는 기본적인 초기화 함수들과 d3d9, customvertex 모듈을 모두 임포트(import)한다. color.py는 색에 관한 심볼들이 정의된 모듈이며 d3d9.py는 d3d9.pyd에 대한 파이썬 인터페이스 파일이다. d3d9.pyd는 Direct3D 라이브러리에 대한 인터페이스 모듈이다. customvertex.py는 customvertex.pyd 파일에 대한 파이썬 인터페이스 파일이며, customvertex.pyd는 주로 많이 사용하는 Vertex 포맷을 미리 정해놓은 인터페이스 모듈이다.

3. C 함수와 파이썬 함수의 주요 차이점들

파이썬과 C언어의 구조적인 차이로 인하여 함수를 사용하는 방법에 있어서도 약간의 차이가 존재한다. C와는 다른 주요한 차이점들을 간단히 정리하면 다음과 같다.

3.1 리턴값

C 함수는 결과 값을 한 개만 리턴 하는 것이 가능하기 때문에 여러 리턴 값이 필요한 경우에는 포인터나 레퍼런스를 함수 인수로 전달한다. 그리고 Direct3D C 라이브러리는 대부분의 경우 리턴 값을 HRESULT 이란 형으로 전달하는데, 이 것은 함수의 수행이 정상적으로 끝났는지 오류가 있었는지를 나타낸다. 예를 들어, 주전자 메시를 생성해내는 함수의 프로토타입을 보자.

```
HRESULT WINAPI D3DXCreateTeapot(
    LPDIRECT3DDEVICE9 pDevice,          // in
    LPD3DXMESH *ppMesh,                 // out
    LPD3DXBUFFER *ppAdjacency           // out
);
```

이와 같은 함수 호출 방식은 단 하나의 값을 리턴할 수 있다는 C의 제약점에서 나온 것이므로, 파이썬에서는 출력 객체인 경우에는 리턴값을 직접 받도록 함수를 모두 조정했다.

```
mesh, adjacency = D3DXCreateTeapot(device)
```

C함수의 리턴 값 HRESULT는 단지 정상적인 함수리턴인지의 여부를 검사하는 것이므로, 파이썬에서는 RuntimeError 예외(exception)가 발생한다. 예외가 발생하면 예외 상황을 보고하고 프로그램이 중단되는데, 이것은 try-except 절을 이용하여 예외처리를 할 수 있다. 예외가 발생하는 경우에는 에러 코드 값을 함께 전달한다.

3.2 C 포인터를 이용한 메모리 참조

C에서는 종종 버퍼에 대한 포인터를 이용하여 메모리를 직접 참조하게 되는데, 파이썬에서는 C에 대응하는 포인터 자료형이 없다. 내부적으로는 레퍼런스란 개념으로 사용하고 있지만 주소를 나타내는 포인터를 직접적으로 사용하지 않는다. 따라서 C 포인터 자료형을 인수로 넘겨야 할 경우에는 다른 방법이 필요한데, 이를 해결하기 위해서 중간에서 변환역할을 해주는 래퍼(wrapper) 클래스를 이용한다. 이 래퍼 클래스는 C의 인덱싱을 파이썬의 메쏘드로 변환해주는 역할을 한다. 예를 들어 다음은 C에서 인덱스 버퍼에 값을 저장하는 예이다.

```
WORD* indices = 0;
ib->Lock(0, 0, (void**)&indices, 0);
indices[0] = 0; indices[1] = 1; indices[2] = 2;
indices[3] = 0; indices[4] = 2; indices[5] = 3;
```

파이썬에서 이러한 기능을 사용할 수 있도록 별도의 래

핑(wrapping) 클래스 WordBuffer, DWordBuffer를 준비했다. 즉 파이썬에서는 다음과 같이 사용 가능하다.

```
indices = WordBuffer(IB.Lock(0, 0, 0))
indices[0] = 0; indices[1] = 1; indices[2] = 2;
indices[3] = 0; indices[4] = 2; indices[5] = 3;
```

즉, 파이썬에서는 indices[0]=0 같은 문은 indices.__setitem__(0,0) 이란 메쏘드로 전환되어 호출된다.

3.3 정점 구조체의 처리

정점(Vertex)인 경우 C에서는 그 저장 포맷을 자유롭게 선언할 수 있다.

```
struct Vertex
{
    Vertex() {}
    Vertex(float x, float y, float z)
    {
        _x = x; _y = y; _z = z;
    }
    float _x, _y, _z;
    static const DWORD FVF;
};
const DWORD Vertex::FVF = D3DFVF_XYZ;
```

그리고 값을 설정하기 위해서 다음과 같이 한다.

```
Vertex* vertices;
vb->Lock(0, 0, (void**)&vertices, 0);

// vertices of a unit cube
vertices[0] = Vertex(-1.0f, -1.0f, -1.0f);
vertices[1] = Vertex(-1.0f, 1.0f, -1.0f);
vertices[2] = Vertex( 1.0f, 1.0f, -1.0f);
```

파이썬에서는 C#에서 사용하는 정점 구조체 개념을 이용한다. 즉, 사용가능한 정점 구조를 미리 준비해두고 다음과 같이 쓸 수 있게 했다.

```
buf = PositionOnlyBuffer(vb.Lock(0, 0, 0))
buf[0] = PositionOnly(-1.0, -1.0, -1.0)
buf[1] = PositionOnly(-1.0, 1.0, -1.0)
buf[2] = PositionOnly( 1.0, 1.0, -1.0)
```

현재 PositionOnly, PositionNormal, PositionNormalTextured, PositionTextured, PositionColored, PositionColoredTextured, PositionNormalTextured2 와 그에 대응하는 버퍼 클래스들을 준비해놓고 있다. 또한 사용자가 원하는 별도의 정점 클래스가 있다면 customvertex.h, customvertex.cpp 파일과 customvertex.i 를 이용해서 추가할 수 있도록 하고 있다.

4. 메쏘드의 확장

4.1 C 메쏘드의 확장

SWIG는 C 래퍼코드를 생성하면서 클래스나 구조체에 대한 새로운 메쏘드를 C레벨에서 추가하는 것을 가능하게 한다. 따라서, 클래스 조작에 도움이 될 간단한 편의함수를 추가해서 사용할 수 있다. 간단한 예로, D3DXMATRIX 구조체 행렬의 초기화를 수행하기 위해서 다음과 같이 외부 함수 ZeroMemory를 호출해야 한다.

```
D3DXMATRIX mat;
::ZeroMemory(&mat, sizeof(D3DXMATRIX));
```

파이썬에서는 Clear() 메쏘드를 추가하여 다음과 같이 쉽게 사용할 수 있도록 하였다.

```
>>> mat = D3DXMATRIX()
>>> mat.Clear()
```

4.2. 파이썬 메쏘드의 확장

SWIG는 또한 파이썬 레벨에서의 메쏘드도 확장할 수 있도록 기능을 제공한다. 주요한 파이썬 메쏘드의 확장은 출력 형식인 __repr__ 메쏘드의 추가이다. 이 메쏘드는 문자열 변환시 혹은 print 문에 의한 출력시 출력 양식

을 정해주는 메쏘드로서 대화형으로 값을 확인하거나 디버깅할 때 유용하게 활용된다. 클래스나 구조체의 멤버가 또 다른 구조체일 경우에도 재귀적으로 그 값을 출력하게 하였다. 다음 예에서 컬러값(Diffuse, Ambient 등)의 예를 참조하기 바란다.

```
point = InitPointLight() # D3DLIGHT9 객체를 리턴
print point
```

출력은 다음과 같다.

```
<D3DLIGHT9>
    <Type>1</Type>
    <Diffuse><D3DCOLORVALUE> <r>1.000000</r>
    <g>1.000000</g> <b>1.000000</b>
<a>1.000000</a></D3DXCOLOR></Diffuse>
    <Specular><D3DCOLORVALUE>
    <r>0.600000</r> <g>0.600000</g>
    <b>0.600000</b>
<a>0.600000</a></D3DXCOLOR></Specular>
    <Ambient><D3DCOLORVALUE>
    <r>0.600000</r> <g>0.600000</g>
    <b>0.600000</b>
<a>0.600000</a></D3DXCOLOR></Ambient>
    <Position><D3DVECTOR> <x>0.0</x>
    <y>0.0</y> <z>0.0</z></D3DVECTOR></Position>
    <Direction><D3DVECTOR> <x>0.0</x>
    <y>0.0</y> <z>0.0</z> </D3DVECTOR></Direction>
    <Range>1000.0</Range>
    <Falloff>1.0</Falloff>
    <Attenuation0>1.0</Attenuation0>
    <Attenuation1>0.0</Attenuation1>
    <Attenuation2>0.0</Attenuation2>
    <Theta>0.0</Theta>
    <Phi>0.0</Phi>
</D3DLIGHT9>
```

4.3 C 확장 모듈의 결합

고급언어로 풍부한 기능과 사용의 편의성을 얻는다면 얻는 것은 일반적으로 속도이다. 파이썬은 C언어 비해서 비교가 안될만큼 빠른 개발 속도와 코드의 유지보수 및 확장 기능이 뛰어나다. 하지만 손해보는 것은 역시 속도이다. 그럼에도 불구하고 파이썬이 수치연산이나 계산을 많이 필요로 하는 모델링등에 활용될 수 있는 이유는 바로 C언어와의 결합성 때문이다.

파이썬과 Direct3D의 결합으로 API 함수의 실행속도는 C와 동일하지만 그 호출과정의 부하로 약간의 속도저하가 일어난다. 속도저하의 양은 파이썬 코드가 어떻게 작성되느냐에 크게 의존하기 때문에 여기서 일정한 수치로 결정짓기 힘들지만, 일반적으로 파이썬에서의 루프나 연산이 적으면 적을수록 C에 근접한 속도를 낸다. 초당 프레임수를 기준으로 판단할 때 지금껏 테스트해본 예제로는 많아야 10% 정도의 감소효과가 있는 정도이다. (일반적으로 속도의 증감을 체감하려면 20%의 차이는 있어야 한다.)

하지만 파이썬은 많은 루프 연산에는 취약하여, 예를 들어 5000개의 눈송이를 동시에 만들어내는 파티클 효과와 같이 객체 수가 크게 증가하면 파이썬의 속도는 크게 저하된다(10배정도). 이런 경우 파티클에 대한 클래스를 C언어로 만들고 파이썬에서 호출해서 사용하면 C언어의 속도를 얻음과 동시에 프로그래밍의 유연성도 함께 얻을 수 있다.

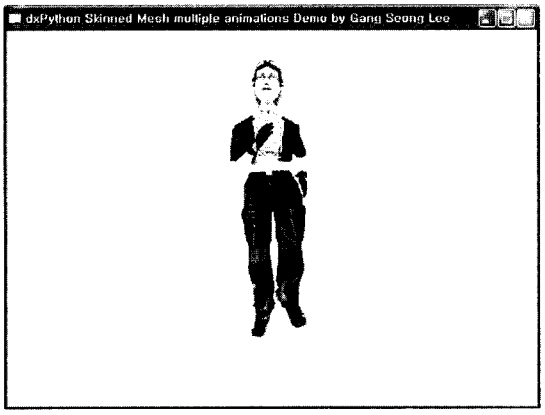
이러한 확장 모듈의 작성 예는 배포 패키지에 함께 포함되어서 사용자가 쉽게 참고할 수 있도록 하였다.

5. 실험

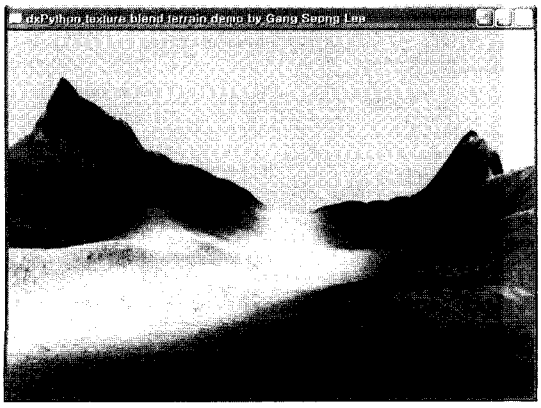
Direct3D 파이썬 모듈은 WindowsXP, Python-2.4.1, SWIG-1.3.25, MS Visual C++ 7.1환경하에서 개발되었고, DirectX 버전은 9.0c를 대상으로 하였다. CPU는 Intel Pentium IV 1.8G, 그래픽 카드는 NVIDIA GeForce 6600과 NVIDIA GeForce2 MX/MX 400 에서 각각 테스트 되었다.

테스트는 주로 C/C++ 샘플 코드들을 파이썬으로 변환하면서 하였고, 참고문헌 [8] 예제를 모두 변환 테스트해보았

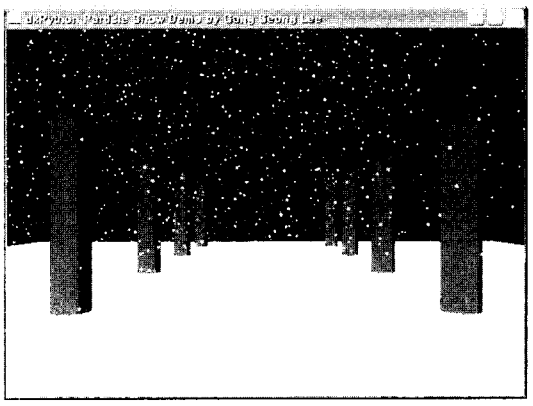
다. 40여개의 테스트 샘플에서 다음과 같은 것들이 포함된다. 버텍스, 텍스쳐, 지형, 스텐실 버퍼, 쉐이더, 사원수, 피킹, 파티클, 메쉬(스킨드 메쉬, 프로그레시브 메쉬, 메쉬 생성, 분할등), 조명, 폰트, 색상, 카메라, 블렌딩, 빌보드, 뷰포트등이다. 또한 라이브러리 확장으로 사운드, 미디 테스트 샘플도 추가되었다. 또한 각종 알고리즘의 구현 가능성을 확인하기 위해 Quadtree, Skybox, Texture Blended Terrain, Animation Blending등도 테스트 해본 결과 무리없이 잘 수행되는 것을 확인할 수 있었다. [그림 1]에 Skinned Mesh 캐릭터 애니메이션의 예에서 tiny_4anim.x를 읽어서 애니메이션 한 예를 보인다[9]. [그림 2]는 텍스쳐 블렌딩 기법을 이용하여 전체적인 지형 윤곽을 지정하면서도 세부 텍스쳐가 효과적으로 잘 표현되도록 한 지형 생성의 예이다[10]. 또한 [그림 3]은 5000개의 눈 파티클을 만들어낸 예이다[8].



[그림 1] Skinned Mesh 애니메이션의 예



[그림 2] Texture blended 지형 생성



[그림 3] 5000개의 눈 파티클 생성

5.1 실행 속도 비교

같은 조건에서 속도를 측정하기 위해서 세가지 실험을 했다. 첫 번째는 캐릭터 애니메이션이고 두 번째는 지형 그리기 그리고 마지막으로 파티클 효과이다. 실험 조건은 Pentium IV 1.7G, 그래픽 카드 NVIDIA GeForce 6600에서의 결과이다.

먼저 캐릭터 애니메이션은 [그림 1]에서 보인 tiny의 animation 초당 프레임수(FPS)를 비교 측정했고[9], 지형은 참고문헌[10]을 기준으로 비교하였다. 파티클은 참고문헌[8]의 14장에 있는 눈 파티클 예제와 비교하였다. 결과는 [표1]에 나타나 있다. C의 속도는 디버그 모드에서 측정된 것이다. 속도 측정은 10초간의 프레임수를 평균 낸 것이다.

	Python + dxPython	C + DirectX
캐릭터 애니메이션	580 FPS	640 FPS
지형그리기	350 FPS	440FPS, 340FPS, 250FPS
파티클1	123 FPS (C+Python)	105 FPS
파이클2	5 FPS (Python)	105 FPS

[표 1] Python/dxPython과 C/DirectX 의 수행 속도 비교

캐릭터 애니메이션인 경우에는 약10%의 속도 차이가 있었는데, 지형 그리기에서는 표시되는 부분에 따라서 오히려 dxPython의 프레임수가 더 나오는 경우도 있었다. 즉 C에서 하늘과 같은 단순한 텍스쳐 매핑 부분을 표시할 때는 440FPS 정도가 나오지만 블렌딩된 지형 부분만을 표시할 때는 250FPS로 크게 줄어들었다. 반면에 파이썬에서는 약 350 정도의 일정한 FPS가 출력되었다. 파티클1은 파티클 생성 부분을 C모듈로 만든 것이고, 파티클2는 순수한 파이썬으로 만든 것이다. 파티클 생성부분을 C모듈로 효율적으로 만들은 경우에 오히려 C+DirectX 보다 더 빠른 속도를 내었

고, 순수한 파이썬으로 만들었을 경우는 5FPS라는 대단히 느린 속도로 수행되었다. 파이썬만을 가지고 대단위 루프를 수행하는 것은 상당히 느리다는 것을 보여준다. 이것은 파이썬과 C의 장점을 잘 결합하면 속도에 있어서는 파이썬의 단점을 보완할 수 있다는 것을 단적으로 보여주는 예이다.

5.2 메모리 사용량 비교

앞서 실험한 네 가지 경우에 대해서 메모리 사용량을 검사해보았다. 다음 [표2]에 그 결과를 보인다.

	Python + dxPython	C + DirectX
캐릭터 애니메이션	16M	5M
지형그리기	52M	48M
파티클1	18M	7M
파티클2	14M	7M

[표 2] Python/dxPython과 C/DirectX의 메모리 사용량 비교 (단위:바이트)

이미지 메모리를 많이 사용하는 지형그리기인 경우에는 그 차이가 심하지 않았지만, 대부분 C로 작성된 프로그램의 메모리 사용량이 현저하게 적었다. 이것은 앞서 설명한 바와 같이 파이썬은 기본 C 라이브러리를 인터페이스 하기위한 C 모듈 + 파이썬 모듈 + 파이썬 인터프리터가 함께 메모리에 올라가기 때문이다.

6. 결론

본 연구에서는 SWIG를 이용하여 MS DirectX의 Direct3D 라이브러리의 파이썬 인터페이스를 구현하였다. C언어와 파이썬의 차이점을 가능하면 줄이려고 노력하였고, 가능하면 C/C++ 함수 호출방식을 유지하려고 하였으나, 확실히 파이썬의 장점을 살릴 수 있는 부분을 살리려고 노력하였다.

이 모듈은 파이썬의 장점과 C/C++의 장점을 결합해주는 역할을 하여, 파이썬 언어의 간결성과 유연성 및 C/C++의 속도를 함께 얻을 수 있다. 다소간의 속도 저하 문제는 C 모듈과의 결합으로 충분히 극복가능하다는 것을 보여주었다. Direct3D 라이브러리 함수에 대한 테스트를 직접 대화식으로 수행해볼 수 있으며, 쉽게 모델링이나 게임을 위한 프로그램을 작성할 수도 있다. 또 본격적으로 C/C++ 코드를 개발하는 개발자도 프로토타이핑 도구로서 이용하면 유용하게 활용될 수 있음을 확인했다[12].

참고문헌

[1] Python Language, http://www.python.org

[2] DxRuby, http://www14.big.or.jp/~amiami/fsk/

[3] Java 3D Implementation - OpenGL vs DirectX, http://java3d.j3d.org/implementation/java3d-OpenGLvsDirectX.html

[4] David M. Beazley, Interfacing C/C++ and Python with SWIG, http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf, 1998.10

[5] SWIG home page, http://www.swig.org,

[6] SWIG documentation, http://www.swig.org/Doc1.3/SWIGDocumentation.html

[7] Boost.Python,. http://www.boost.org/libs/python/doc/

[8] Frank Luna, 'DirectX9를 이용한 3D GAME 프로그래밍 입문' , 정보문화사, 2004. 2

[9] Frank Lina, 'Skinned Mesh Character Animation with Direct3D 9.0c', http://www.moon-labs.com/resources/d3dx_skinnedmesh.pdf, 2004. 9

[10] Frank Lina, 'Basic Terrain Rendering Part II', http://www.moon-labs.com/resources/TerrainPart2.pdf, 2004. 9

[11] MSDN Library, http://msdn.microsoft.com/

[12] dxPython home page, http://dxpython.pythonworld.net/

이 강 성

1986년 2월 광운대학교 컴퓨터공학과 (공학사)
1988년 8월 광운대학교 대학원 컴퓨터공학과 (공학석사)
1993년 2월 광운대학교 대학원 컴퓨터공학과 (공학박사)
현재 광운대학교 교양학부 교수
관심분야 : 게임 프로그래밍, 신호처리, 음성인식, 컴퓨터 음악

논문투고일 - 2006년 1월 5일
심사완료일 - 2006년 2월 16일