ISBN 89-5884-673-8 98560

볼륨 렌더링 프로그램 User Interface 개발

(Development of User Interface for Volume Rendering Program)

이 중 연 (Joong Youn Lee)
jylee@kisti.re.kr
Visualization Team, Supercomputing Center

한 국 과 학 기 술 정 보 연 구 원
Korea Institute of Science & Technology Information

<제목 차례>

<표 차례>

<그림 차례>

I. 서론

가. 유저 인터페이스란?

유저인터페이스(User Interface, 이상 UI)란 사용자들이 컴퓨터 시스템 또는 프로그램에서 데이터 입력이나 동작을 제어하기 위해 사용하는 명령어 또는 기법을 말한다. 사용자가 프로그램과 의사소통을 하고 쉽고 편리하게 사용할 수 있도록 하는 것이 목적이다. 특히 동작의 목록을 아이콘이나 메뉴로 보여주고 사용자가 마우스로 작업을 할 수 있도록 하는 그래픽 유저 인터페이스(GUI)는 사용자 편의를 극적으로 향상시켰다. 과거에는 프로그램 수행 시간의 대부분이 CPU 연산에 있었기 때문에 최대한 빠르게 프로그램이 실행되도록 알고리즘 및 코드를 최적화하고 시스템 성능을 높이는데 주력했다. 하지만 최근에는 컴퓨터 하드웨어 성능이 급격히 높아져서 사용자와의 상호작용 시간이 전체 수행시간에 차지하는 비중이 높아졌다.

나. 볼륨 렌더링 프로그램의 유저 인터페이스

볼륨 렌더링(volume rendering)이란 볼륨 데이터라 불리는 3차원의 불연속적으로 샘플된 데이터를 인간이 쉽게 인지할 수 있는 2D 이미지로 투영해서 가시화하는 기법을 말한다. 물론, 여러 개의 2차원 슬라이스로 변환한 뒤 슬라이스별로 가시화하는 방법도 있겠지만, 이러한 경우에는 각 슬라이스의 상관관계를 파악하기 힘들다는 단점이 존재한다. 볼륨 데이터를 가시화하기 위해서는 매우 큰 컴퓨팅 능력이 필요하기 때문에 지금까지는 슈퍼컴퓨터나 고성능 서버 또는 전용 그래픽스 워크스테이션을 이용해서 가시화해왔다. 그러나 2000년대에 들어서면서 NVIDIA GeForce 시리즈나 ATI Radeon 시리즈 등의 일반 PC용 그래픽스 카드들의 성능이 급격히 발전하면서 범용 PC에서 볼륨 렌더링을 하려는 시도가 계속되고 있다. 일반적으로 PC 그래픽스 하드웨어를 이용한 볼륨 렌더링은 3차원 텍스쳐 매핑(3D texture mapping)을 이용하여 많이 구현되는데, 지금까지 보다 빨리 그리고 보다 양질의 영상을 얻기 위한 많은 노력이 있어왔다. 특히, PC용 그래픽스 하드웨어에서 꼭지점 및 프래그먼트 프로그래밍(vertex / fragment programming)이 가능해지면서 과거에는 고가의 그래픽스 워크스테이션 또는 볼륨 렌더링 전용 하

드웨어에서나 가능하던 실시간 볼륨 렌더링이 일반적인 PC에서도 가능해지게 되었다. 이렇게 일반 PC에서 실시간 볼륨 렌더링이 가능해지면서 유저 인터페이스의 중요성도 점점 높아지고 있다. 과거에는 하나의 볼륨 렌더링 영상을 얻기 위해 몇 초에서 몇 시간까지 필요했기 때문에 일반적으로 배치 작업으로 많은 영상들을 얻어 왔으며 이에 따라 사용자와 렌더링 프로그램이 실시간으로 상호작용할 수 없었다. 그러나 최근의 그래픽스 하드웨어의 발전으로 인해 실시간으로 볼륨 렌더링이 되면서 사용자와의 상호작용의 필요성이 증대되어 쉽고 편리한 유저 인터페이스가 필수적이 되고 있다.

II. 볼륨 렌더링 유저 인터페이스 설계 내용

가. 중요 인터페이스들

(1) 메인 윈도우

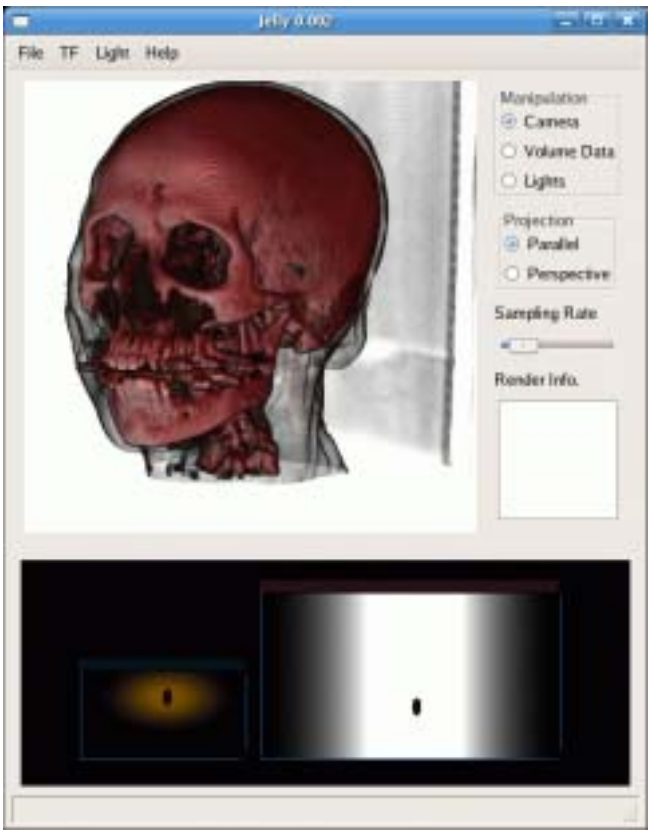

그림 II-1. 볼륨 렌더러 메인 윈도우

메인 윈도우에는 우선 좌측 상단에 렌더링 윈도우가 있고 오른쪽에 각종 설정 위젯들이 있다. "Manipulation" 항목에서는 렌더링 윈도우에서 마우스 작업을 할 때의 작업 대상을 선택할 수 있다. "Camera"를 선택하면 렌더링 윈도우에서 카메라가 움직이고, "Volume Data"를 선택하면 볼륨 데이터가 움직이며, "Lights"를 선택하면 선택한 광원이 움직인다. "Projection" 항목에서는 렌더링 윈도우의 투영 모드를 선택할 수 있고, "Sampling Rate" 슬라이드 바는 볼륨 렌더링의 샘플링 레이트를 설정할 수 있도록 한다. "Render Info." 에는 렌더링에 오류가 발생할 경우 정보가 출력되도록 했으며, 가장 하단에서는 Transfer Function을 설정할 수 있다.

(2) 광원 관련 윈도우

그림 II-2. 광원 추가 윈도우

그림 II-3. 광원 수정 윈도우

광원 관련 윈도우에는 광원 추가 윈도우와 광원 수정 윈도우가 있다. 두 윈도우는 거의 동일한 인터페이스를 가지고 있는데, 우선 가장 상단에 현재 선택된 광원의 ID가 보이고 그 우측에 광원의 종류를 선택할 수 있도록 했다. 현재는 Ambient Light와 Directional Light 중에서 하나를 선택할 수 있다. 그 하단에는 광원의 색깔을 선택할 수 있다. 우측의 Change 버튼을 누르면 그림 II-4와 같은 표준 색깔 선택 윈도우가 나타난다. Directional Light의 경우 광원의 방향을 설정해야 하는데 이것은 가장 하단의 Direction 슬라이드 바를 이용해서 할 수 있다.

그림 II-4. 색깔 선택 윈도우

(3) Shading Factor 관련 윈도우

그림 II-5. Shading Factor 수정 윈도우

볼륨 데이터의 각 물질에 대해 퐁 쉐이딩을 하기 위해선 앰비언트(ambient), 디퓨즈(diffuse), 스펙큘러(specular) 광원에 대한 계수를 설정해야 한다. 이 윈도우는 이러한 쉐이딩 계수를 설정하기 위한 윈도우인데, 메인 윈도우의 가장 하단에 있는 Transfer Function 에디터의 각 위젯을 오른쪽 클릭해서 설정할 수 있다.

나. 마우스

- 렌더링 윈도우

왼쪽 마우스 버튼 : 회전

오른쪽 마우스 버튼 : 빠른 메뉴

Shift + 왼쪽 마우스 버튼 : 이동

휠 / Ctrl + 왼쪽 마우스 버튼: 줌인 / 줌아웃

- TF 에디터

왼쪽 마우스 버튼 : 위젯 선택

마우스 드래그 시 위치 이동

선택 상태에서 delete 누르면 삭제

Shift + 클릭 : 위젯 종류 변경

Shift + Insert / Ctrl + C : 복사

Shift + Delete / Ctrl + X : 잘라내기

Ctrl + Z : 취소

위젯의 컨트롤 포인트를 선택할 경우에는 크기 조정

오른쪽 마우스 버튼 : 빠른 메뉴

위젯 종류 변경

색깔 및 투명도 변경

Shading factor 변경

다. 메뉴

File : 파일 처리 관련 메뉴 모음

Open : XML 설정 파일 읽기

Save : XML 설정 파일 저장하기

Save Image : 렌더링 영상 저장하기

Exit : 끝내기

TF : Transfer Function 관련 메뉴 모음

Load : Transfer Function 읽기

Save : Transfer Function 저장하기

Add Widget : 위젯 추가

Light : 광원 관련 메뉴 모음

Add : 광원 추가

Remove : 광원 삭제

Modify : 광원 설정 변경

On/Off : 광원 켜기 / 끄기

Help : 도움말 관련 메뉴 모음

Help : 도움말

About : 프로그램 설명

III. TFUI

TFUI란 Transfer Function User Interface의 약자로 본래 미국의 University of Utah에서 개발한 Simian 시스템의 Transfer Function 유저 인터페이스인데, 이것을 University of Illinois at Chicago(이상 UIC)의 EVL(Electronic Visualization Laboratory)에서 개발한 네트워크 기반 타일 디스플레이용 볼륨 렌더링 프로그램인 Vol-a-Tile에서 사용하기 위해 분리한 프로그램이다. Simian의 Transfer Function 에디터는 본래 3D Transfer Function을 지원했지만 EVL에서는 이를 2D Transfer Function만 지원하도록 수정했고, 몇 가지 복잡한 위젯을 제거해서 단순화 했다. 이 에디터를 이용하면 Transfer Function의 복셀값 별 색깔 및 투명도를 설정할 수 있는데, 사용자가 쉽게 위젯들을 설정할 수 있도록 배경에 렌더링할 볼륨 데이터의 2D 히스토그램을 표시해 준다.

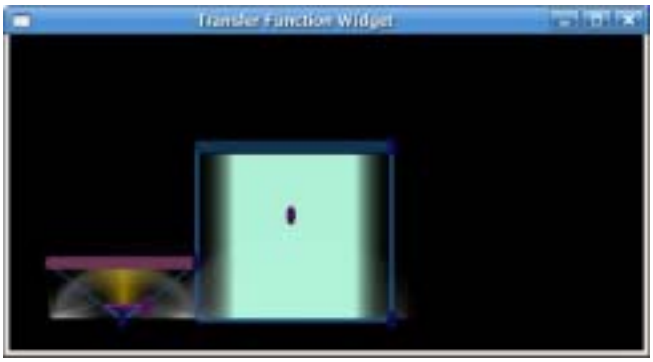

그림 III-1. TFUI 화면

가. 위젯

위젯에는 총 4가지 종류가 있는데, 다음과 같다

- Triangle

Levoy의 2D transfer function을 위젯으로 만든 것이다. gradient의 크기가 클 수록 opacity가 커지고 gradient의 크기가 작을 수록 opacity가 작아져서 물질간 경계 영역을 강조한다.

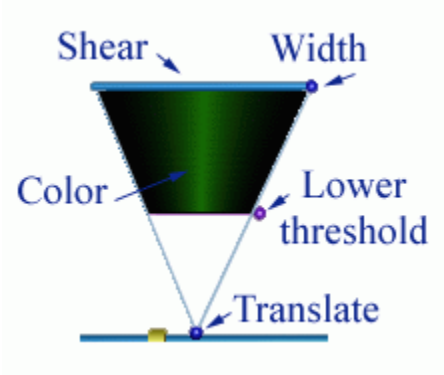

그림 III-2. Triangle 위젯

- Elliptical

일반적인 목적의 transfer function을 생성할 때 적합하다.

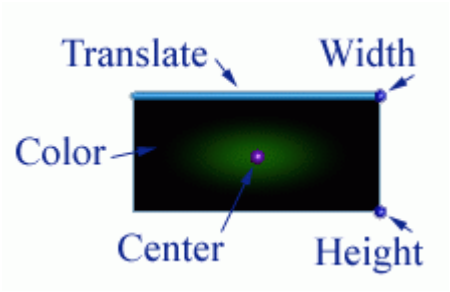

그림 III-3. Elliptical 위젯

- 1D

일반적인 1D transfer function과 같이 그라디언트(gradient) 크기에 따른 투명도(opacity) 변화가 없다.

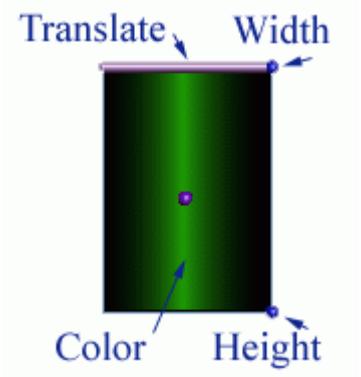

그림 III-4. 1D 위젯

- **Default**

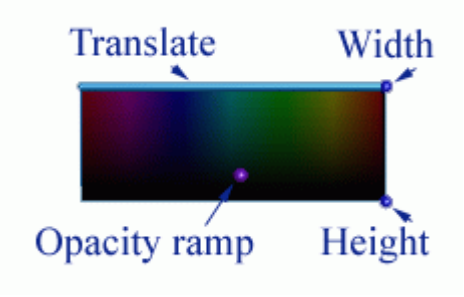

그림 III-5. Default 위젯

shift 또는 ctrl 키를 누르고 마우스 클릭을 하면 각 위젯의 종류의 변경이 가능하고, 위젯을 선택한 상태에서 delete 키를 누르면 선택한 위젯이 삭제된다. 위젯의 안쪽 부분을 마우스 왼쪽 버튼으로 클릭한 후 좌우로 이동하면 색깔이 바뀌고, 위아래로 이동하면 레벨이 바뀐다. 또한, 마우스 오른쪽 버튼으로 클릭한 후 위아래로 이동하면 투명도가 변경된다.

나. TFUI Library

TFUI를 볼륨 렌더링 유저 인터페이스에 추가하기 위해 TFUI를 공유 라이브러리로 만들었다. 기존 TFUI는 C++로 작성되었는데, 이를 Python에서 접근 가능하도록 하기 위해 클래스들에 대한 많은 수정이 있어야 했다. TFUI 프로그램은 2개의 전역 변수에 중요한 데이터가 저장되어 있는데, 이는 tfGlobal 타입의 tfData와 tfWidget 타입의 *tfw 이다. 각 클래스들의 정의는 다음과 같다.

```
class tfGlobal
{
public:
   tfWindow win;
   tfDeptex deptex;       /* 실제 TF를 저장한 진짜 텍스쳐 */
   tfDeptex dispDeptex;  /* TF를 빠르게 디스플레이하기 위한 작은 크기의 텍스쳐
*/
   sfDeptex sftex;        /* shading factor들을 저장한 텍스쳐 */

   int histOn;
```

```
   int picking;
   tfPick pick;          /* 현재 선택된 object */
   tfMouse mouse;
   tfEnv env;
   bool interaction;
   bool updateVol;
   bool histUpdate;
   unsigned char *histo;    /* histogram 데이터 */
   unsigned int  histName;  /* histogram이 텍스쳐 이름 */
   int histoWidth;
   int histoHeight;

   tfGlobal();
};

class tfWidget : public tfBaseWidget
{
public:
   tfWidget(unsigned int w = 256, unsigned int h = 256);
   virtual ~tfWidget();

   void draw();
   void init();
   int  pick();
   int  key(unsigned char key);
   int  mouse(int button, int state,int x, int y);
   int  move(int x, int y);
   int  release();
   void getColorTable(float *buffer);
   void getSFTable(float *buffer);
   void saveDepTex(const char* filename);
   void saveDepTexPPM(const char* filename);
   void readDepTex(const char* filename);
   void readHistogram (const char *filename);
   void  clearDepTex(tfDeptex *deptex);
   LevWidget *tris;                  /* levoy widgets */
   int  numWidget;

private:
   void  drawFrame();
   void  mouse2plane(int x, int y);
   void  loadDepTex(tfDeptex *deptex);
   void  loadHist2D();
   void  create2DDepTex(tfDeptex *deptex);

   enum                              /* subobject names */
     {
        TFobjUnknown, tfsheet1
     };

   float width, height;              /* width and height of widget
*/
```

```
  float screenwidth, screenheight; /* fraction of screen space
this widget occupies */
  float screenpos[2];              /* posn relative to screen */
  float screenbar, screenball;    /* slider and ball radius wrt
screen space */
  float mousepnt[3];       /* position of mouse click on tf plane
*/
  int   pickedTri;         /* the segment picked by the user */
};
```

그림 III-6. tfGlobal Class와 tfWidget Class의 구조

이 TFUI의 두 전역변수의 멤버들 중 python에서 접근해야하는 멤버들과 여러 인터페이스 함수들을 python에서 불러올 수 있도록 SWIG를 이용해서 공유 라이브러리를 만들었다. 또, TFUI 라이브러리를 구동하기 위한 내부 함수들 및 위젯들을 생성, 삭제하고 정보를 얻기 위한 전역 함수들을 정의했다. 전역변수의 멤버들 중 접근해야 하는 멤버들과 각 전역 함수들의 정의는 다음과 같다.

```
class tfGlobal
{
  tfDeptex deptex;
};

class tfw
{
  tfw(int width, int height);
  numWidget;
  init();
  getColorTable(float *buffer);
  getSFTable(float *buffer);
  setSFTable(float *buffer);
  readHistogram(char *);
};

void disp();
int  mouse(int button, int state, int button, int x, int y);
void motion(int x, int y);
void key(unsigned int key);
void reshape(int width, int height);
int  pick();

void createNewWidget(char *name, float *points, float *thresh,
float opacity, float *color, float *sf);
void removeAllWidgets();
int  getWidgetInfos(int index, char *type, float *points, float
*thresh, float *color, float *opacity, float *shadingfactors);
```

그림 III-7. TFUI 라이브러리 중 외부에서 접근해야 하는 전역함수 및 변수들

TFUI 자체에서는 모든 인터페이스를 마우스 또는 키보드로 입력받고 내부적으로 데이터를 처리, 화면 또는 파일로 출력하게 했기 때문에 다른 프로그램에서 내부의 데이터에 접근할 수 있는 방법이 없다. 따라서 이를 쉽게 처리할 수 있도록 TFUI 라이브러리에 여러 인터페이스 함수를 추가했다. 또한, 기존의 TFUI에서는 ambient 계수, diffuse 계수, specular 계수, shininess 계수 등의 shading factor들의 처리를 전혀 하지 않는다. 이러한 문제를 해결하기 위해 내부적으로 이러한 shading factor를 처리하도록 코드를 추가했고 인터페이스 함수 역시 작성했다. 각 인터페이스 함수들의 정의는 다음과 같다.

```
float getOpacity();
void  setOpacity(float opacity);

void  getColor(float *red, float *green, float *blue);
void  setColor(float red, float green, float blue);

void  getDensityRange(float *start, float *end);

float getAmbient();
void  setAmbient(float ambient);

float getDiffuse();
void  setDiffuse(float diffuse);

float getSpecular();
void  setSpecular(float specular);

float getShininess();
void  setShininess(float shininess);
```

그림 III-8. TFUI 라이브러리의 내부 변수 인터페이스 함수들

다. 문제점

TFUI의 경우 일단 볼륨렌더링 인터페이스에의 삽입의 구현에 초점을 맞췄고, 기존 TFUI class의 설계의 복잡함으로 인해 전체적으로 일관되지 못한 설계를 가졌다. 전반적으로 tfData와 tfw의 두 전역변수들을 통해 데이터가 관리되는데, 이에 접근하기 위한 방법들이 어떤 데이터는 추상화된 전역함수를 통해서 관리되고 어떤 데이터는 class의 변수에 직접 접근하는 등 일관되지 못하다. 이를 해결하기 위해서는 전반적인 refactoring이 필요하다.

IV. 구현

가. TVAPI

일반 사용자가 쉽게 볼륨 렌더링 프로그램을 개발할 수 있도록 볼륨 렌더링에 필수적으로 필요한 핵심 기능들을 API로 개발했다. TVAPI는 그래픽스 하드웨어의 GPU를 이용한 볼륨 렌더링 라이브러리로 C 언어와 NVIDIA의 Cg 쉐이더를 이용했다. 주요 함수들은 모두 OpenGL의 함수들을 참조해서 만들었기 때문에 TVAPI 역시 OpenGL과 마찬가지로 일종의 State Machine으로 동작한다.

나. wxPython

전체적인 유저 인터페이스의 구현은 wxPython을 이용했다. wxPython은 Python 언어를 위한 GUI 툴킷으로 쉽고 간편하면서 안정적으로 그래픽 유저 인터페이스를 개발할 수 있게 한다. 이 툴킷은 wxWidget을 기반으로 Python의 extension 모듈로 개발되었고, wxWidget의 풍부한 리소스들과 Python의 간편하면서도 강력한 언어적 특징을 모두 갖췄다.

다. OpenGL과 wxPython

wxPython의 윈도우에서 OpenGL을 사용하기 위해서는 우선 윈도우가 wxGLCanvas이어야 한다. 이는 wxPython.glcanvas 모듈을 통해 사용이 가능하다. 또, 초기화 시점에 OpenGL도 함께 초기화되어야 하며 PAINT 이벤트가 발생했을 때 OpenGL 명령을 실행해서 그림을 그리도록 해야 하고 프로그램의 성격에 따라 SIZE나 IDLE 이벤트 때에도 viewport를 변경한다거나 애니메이션을 실행시킨다거나 하는 처리를 해주어야 한다. 이때, OpenGL 명령들은 pyOpenGL을 사용해도 되고 C/C++로 작성된 외부 라이브러리에서 호출해도 된다. 예제 코드는 아래와 같다. 여기서 wxGLCanvas 클래스를 초기화할 때 attribList List를 설정해서 인자로 넣는데, 이는 생성되는 OpenGL 윈도우의 속성을 설정하는 것이다. 이 예제에서는 프레임 버퍼의 픽셀 형식을 RGBA로 하고 Alpha 버퍼와 Stencil 버퍼의 크기를 8bit로 하며 더블버퍼링을 하도록 했다.

```
from wxPython.glcanvas import *
from wxPython.wx import *

class tvRenderClass(wxGLCanvas):
   def __init__(self, parent):
      attribList = [WX_GL_RGBA, WX_GL_MIN_ALPHA, 8, WX_GL_STENCIL_SIZE,
8, WX_GL_DOUBLEBUFFER]
      wxGLCanvas.__init__(self, parent, -1, (0, 0), parent.GetSize(), 0,
"tvRenderCanvas", attribList, wxNullPalette)

      EVT_PAINT(self, self.OnPaint)
      EVT_SIZE(self, self.OnSize)
      EVT_IDLE(self, self.OnIdle)

   def OnPaint:
      ...

   def OnSize:
      ...

   def OnIdle:
      ...
```

그림 IV-1. wxPython에서 OpenGL을 사용하기 위한 윈도우 생성 코드

위의 예제 코드와 같이 OpenGL 명령을 그릴 윈도우의 클래스를 생성하고 main 함수에서 이를 호출하면 된다.

라. Python 포팅

위에서 설명한 TVAPI와 TF Library는 모두 C/C++로 개발되었다. 이를 wxPython에서 사용하기 위해서 모두 Python으로 포팅해야만 했다.

(1) C와 Python의 연동

흔히들 Python을 접착언어(Glue Language)라고 부른다. 접착언어란 C나 Fortran, Java와 같은 다른 언어들과 쉽게 붙어서 프로그램을 작성할 수 있는 Python의 특징에서 나온 것이다. 초보 Python 개발자들의 경우 이러한 Python의 접착언어적인 특징을 보면 누구나 쉽게 Python과 기타 다른 언어들을 연동시킬 수 있을 것이라고 생각한다. 그러나 막상 실제로 연동시키려고 레퍼런스 매뉴얼이나 웹사이트를 뒤져보면 이것이 착각이었다는 것을 깨닫게 된다. Python과 C의 연동이 복잡한 가장 큰 이유는 두 언어의 자료 구조 및 함수에의 인자 전달 방법이 틀리기 때문이다. C에서 제공하지 않는 Python

의 자료구조들 (복소수, 튜플, 사전), 반대로 Python에서 제공하지 않는 C 자료구조들(float 포인터, void 포인터 등)을 상대 언어로 전달하기 위해서는 여러 복잡한 단계를 거쳐야 한다. 또, class나 structure가 매우 복잡하거나 union, enum 등과 같은 자주 사용하지 않는 자료구조의 경우 역시 전달하는데 복잡한 방법을 사용해야 한다. 여기서 한 가지 생각해볼 점은 C와 Python의 연동의 복잡함으로 인해 Python의 가장 큰 장점인 개발의 간편함이 퇴색한다는 것이다.

C와 Python을 연동하기 위해서는 Python의 Python/C API 또는 외부 프로그램/모듈인 SWIG, pyrex, ctypes 등 을 사용하면 된다. 전통적으로 C API와 SWIG가 가장 많이 사용되었는데, 최근에는 이 두 API보다 쉽고 간단한 pyrex와 ctypes도 널리 사용되고 있다.

(2) C와 Python의 연동 이유

Python은 그 자체만으로도 강력한 언어이다. C나 C++에 비해 알고리즘의 구현이 간단해서 개발시간을 단축시켜 줄 뿐만 아니라 완벽한 객체지향 및 다양한 built-in 클래스 및 함수들도 지원한다. 이러한 강력함으로 세계 각지에 많은 유저들을 확보하고 있으며 이들 Python 개발자들이 작성한 다양한 종류의 외부 라이브러리/클래스들을 쉽게 구할 수 있는 것 또한 사실이다. 그러나 이러한 Python의 강력함에도 불구하고 C 개발자 입장에서는 Python에 대해 여러가지 불만이 존재한다. 아래의 몇 가지 불만들은 앞으로 소개할 C와 Python의 연동으로 해결이 가능하다.

1. Python은 C에 비해 느리다. 일반적인 프로그램에서는 상관이 없지만 수치연산등과 같이 속도가 중요한 프로그램에서 Python의 느림은 치명적이 될 수 있다.

2. Python은 소스코드가 사용자에게 노출된다. (물론 이를 숨길 수 있는 다른 방법도 있지만) 사용자에게 노출되기 원치 않는 코드들은 C나 C++로 작성해서 숨길 수 있다.

3. 개발자들이 미리 만들어 놓았던 수많은 C 코드들을 다시 Python으로 재작성하는 것은 프로그래밍 리소스의 낭비이다.

이러한 이유로 많은 Python 개발자들은 프로젝트의 일부분은 C로 작성하고 (또는 기존에 작성해 놓은 C 코드들을 가져오고) 이를 Python과 연동해서 프로젝트를 진행하고 있다. 속도가 중요한 몇몇 부분이나 사용자로부터 감추고 싶은 코드들은 C로 작성해서 Python과 연동시키고 있는 것이다. 물론, 기존에 작성해 놓았던 C 코드들 역시 Python으로 재작성하지 않고 Python과 연동해서 재사용하고 있다. 또한, C로 작성된 대부분의 유명 라이브러리들이 이미 Python Wrapper가 작성되어 Python에서도 쉽게 사용이 가능하게 되어 있기도 하다.

(3) Python/C API

Python에서 제공하는 Python/C API는 Python과 C/C++와의 연동을 가능하게 한다. 이 API에서는 C 뿐만 아니라 C++와의 연동 역시 지원하지만 이름을 간단하게 하기 위해 Python/C API로 짧게 지어졌다. 이 API는 C로 작성된 모듈을 Python에서 사용할 수 있도록 하는 기능(extension)뿐만 아니라 Python으로 작성된 모듈을 C에서 사용할 수 있도록 하는 기능(embedding)까지 지원한다. 일반적으로 extension의 경우는 직관적이고 쉽게 구현이 가능하지만, embedding의 경우는 extension에 비해 복잡하고 덜 직관적이다. 이는 embedding 기능이 구현된지 얼마 되지 않았기 때문이라고 한다. extension의 경우엔 굳이 이 API를 사용하지 않더라도 이를 지원하는 다른 방법들이 얼마든지 있다. SWIG, pyrex, ctypes 등이 바로 그것들인데, 이 방법들 역시 Python/C API에 기반을 두고 만들어진 것으로 결국은 이 Python/C API를 추가한 Python 코드를 생성해낸다.

(4) SWIG

(가) 소개

SWIG는 Simplified Wrapper and Interface Generator의 약자로써, C나 C++로 작성된 프로그램들을 다른 프로그래밍 언어와 연결해서 사용할 수 있도록 하는 소프트웨어 개발 툴이다. SWIG는 Perl, PHP, Python, Tcl, Ruby, PHP와 같은 스크립트 언어뿐만 아니라 C#, Lisp, Java, Modula-3 등의 비스크립트 언어까지도 지원한다. SWIG로 C/C++ 프로그램과 다른 언어를 연결하기 위해선 SWIG 인터페이스 파일을 작성해야 한다. 아래의 그림 IV-2와

같은 C 프로그램을 연결하기위한 SWIG 인터페이스 파일은 그림 IV-3과 같다.

```
/* File : example.c */

double  My_variable  = 3.0;

/* Compute factorial of n */
int  fact(int n) {
       if (n <= 1) return 1;
       else return n*fact(n-1);
}

/* Compute n mod m */
int my_mod(int n, int m) {
       return(n % m);
}
```

그림 IV-2. example.c

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
%}

extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
```

그림 IV-3. example.i

SWIG 인터페이스파일에는 C 코드의 함수와 변수에 대한 선언이 있다. %module 지시어는 SWIG에 의해 생성되는 모듈의 이름을 나타낸다. %{, %} 블럭에는 C 헤더파일이나 추가적인 C 선언문들과 같은 추가적인 코드들을 넣는다. 이렇게 생성한 SWIG 인터페이스 파일을 이용해서 Python 모듈을 생성하는 방법은 그림 IV-4와 같다.

swig는 SWIG 인터페이스 파일을 이용해서 example_wrap.c 파일과 example.py를 생성한다. example_wrap.c 파일은 example.c에 정의된 함수와 변수들에 대한 wrapper를 생성하는 코드를 포함한다. 이제 example.c와 example_wrap.c 두 개의 파일을 컴파일하고 공유 라이브러리를 생성하면 Python을 위한 모듈이 생성된다.

```
unix > swig -python example.i
unix    >    gcc    -c    -fpic    example.c    example_wrap.c
-I/usr/local/include/python2.0
unix > gcc -shared example.o example_wrap.o -o _example.so
unix > python
Python 2.0 (#6, Feb 21 2001, 13:29:45)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.

>>> import example
>>> example.fact(4)
24
>>> example.my_mod(23,7)
2
>>> example.cvar.My_variable + 4.5
7.5
```

그림 IV-4. SWIG를 이용해서 Python 모듈을 생성하는 과정

C++ 파일의 경우에는 다음과 같이 swig에 -c++ 인자를 붙이면 된다.

```
unix > swig -c++ -python example.i
```

그림 IV-5. C++를 위한 SWIG 명령

(나) SWIG 구문

- 함수

C/C++ 소스코드에 있는 전역 함수들은 모두 새로운 Python 함수로 변환된다. 다음과 같이 example 모듈에 fact라는 함수가 있다고 하자.

```
%module example
int fact(int n);
```

그림 IV-6. example 모듈

아래와 같이 일반적인 Python 모듈로 간주하고 함수를 호출하면 된다.

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

그림 IV-7. example 모듈을 Python에서 호출하는 방법

- 전역 변수

C/C++의 전역변수 역시 SWIG에서 쉽게 접근이 가능하다. 그러나 함수와는 다르게 cvar이라는 속성을 통해 접근해야 한다.

```
// SWIG interface file with global variables
%module example
...
%inline %{
extern int My_variable;
extern double density;
%}
...
```

그림 IV-8. SWIG 인터페이스 파일

```
>>> import example
>>> # Print out value of a C global variable
>>> print example.cvar.My_variable
4
>>> # Set the value of a C global variable
>>> example.cvar.density = 0.8442
>>> # Use in a math operation
>>> example.cvar.density = example.cvar.density*1.10
```

그림 IV-9. SWIG에서의 전역변수 호출

SWIG 인터페이스 파일에서 변수를 선언할 때 %immutable 지시어를 사용하면 그 변수는 읽기전용이 된다.

```
%{
extern char *path;
%}
%immutable path;
...
extern char *path;     // Read-only (due to %immutable)
```

그림 IV-10. %immutable 지시어

- constant 변수와 enum

constant 변수를 생성하기 위해선 #define 또는 %constant 지시어를 사용하면 된다. 또, enum을 생성하기 위해선 enum 지시어를 사용하면 된다.

```
#define PI 3.14159
#define VERSION "1.0"

enum Beverage { ALE, LAGER, STOUT, PILSNER };

%constant int FOO = 42;
%constant const char *path = "/usr/local";
```

그림 IV-11. constant 및 enum 선언

- 포인터

SWIG에서는 C/C++의 포인터를 완벽하게 지원한다. 그림 IV-12와 같이 파일 처리 관련 함수들에 대한 인터페이스 파일이 있다고 하자.

```
%module example

FILE *fopen(const char *filename, const char *mode);
int fputs(const char *, FILE *);
int fclose(FILE *);
```

그림 IV-12. 포인터를 사용하는 SWIG 인터페이스 파일

이 함수들은 다음과 같이 Python에서 자연스럽게 호출이 가능하다.

```
import example
f = example.fopen("junk","w")
example.fputs("Hello World\n", f)
example.fclose(f)
```

그림 IV-13. 포인터를 필요로 하는 export 함수의 호출

'0'또는 NULL 포인터는 Python에선 None로 표현된다. fclose 함수에 NULL 포인터를 인자로 넣을 때는 그림 IV-14와 같이 한다.

```
example.fclose(None)
```

그림 IV-14. None 지시어

- Input & Ouput 포인터 인자

C/C++의 함수에 인자로 들어가는 포인터들은 2가지 종류가 있다. 하나는 아래의 add 함수와 같이 값이 함수 내부에서 변경되는 경우(Output으로 사용되는 경우)이고, 나머지 하나는 sub 함수와 같이 포인터의 값이 함수 내부에

서 변경되지 않고 단순히 Input 값으로만 사용되는 경우이다

```
void add(int x, int y, int *result) {
   *result = x + y;
}

int sub(int *x, int *y) {
   return *x-*y;
}
```

그림 IV-15. C 함수의 인자로서의 포인터의 용례

그런데 Python에서는 함수의 Output 값을 return 값으로 받아들이므로 문제가 발생한다. 이러한 경우 typemaps.i 라이브러리의 INPUT 및 OUTPUT 지시어를 통해 이를 처리할 수 있다. 그림 IV-16과 같이 add 함수의 마지막 인자에 OUTPUT 지시어를 넣으면 Python에서 return 값으로 돌려받게 된다.

```
%module example
%include "typemaps.i"

void add(int, int, int *OUTPUT);
int  sub(int *INPUT, int *INPUT);
```

그림 IV-16. typemaps.i의 INPUT, OUTPUT 지시어

```
>>> a = add(3,4)
>>> print a
7
>>> b = sub(7,4)
>>> print b
3
>>>
```

그림 IV-17. INPUT, OUPUT 지시어를 포함한 export 함수의 사용 예

- 행렬

때때로 C/C++의 함수는 인자로 단순한 포인터가 아닌 행렬을 요구하기도 한다. 다음의 예를 보자.

```
int sumitems(int *first, int nitems)
{
    int i, sum = 0;
    for (i = 0; i < nitems; i++) {
        sum += first[i];
    }
    return sum;
}
```

그림 IV-18. 행렬을 인자로 요구하는 C 함수

이러한 경우 carrays.i 라이브러리를 이용하면 쉽게 해결이 가능하다.

```
%include "carrays.i"
%array_class(int, intArray);
```

그림 IV-19. carrays.i 라이브러리

%array_class(type, name) 매크로는 필요로 하는 크기의 행렬을 생성해서 int *나 double *과 같은 일반적인 포인터로 C/C++ 함수에 전달되도록 한다.

```
>>> a = intArray(10000000)         # Array of 10-million integers
>>> for i in xrange(10000):        # Set some values
...     a[i] = i
>>> sumitems(a,10000)
49995000
```

그림 IV-20. 행렬을 인자로 요구하는 export 함수의 호출

이 방법으로 생성한 행렬은 C에서와 같이 인덱스에 대한 범위 체크를 하지 않는다. 따라서 행렬의 범위를 넘어가는 인덱싱을 할 경우 segmentation faul t 에러 등이 일어나는 등 프로그램의 실행에 문제가 있을 수 있으므로 매우 조심스럽게 다루어야 한다

- 클래스

C++ 클래스는 Python의 클래스로 변환이 가능하다. 다음과 같은 클래스가 있다고 하자.

```
class List {
public:
  List();
  ~List();
  int  search(char *item);
  void insert(char *item);
  void remove(char *item);
  char *get(int n);
  int  length;
};
```

그림 IV-21. C++클래스

이 클래스는 Python에서 다음과 같이 사용될 수 있다.

```
>>> l = example.List()
>>> l.insert("Ale")
>>> l.insert("Stout")
>>> l.insert("Lager")
>>> l.get(1)
'Stout'
>>> print l.length
3
```

그림 IV-22. Python에서의 외부 C++ 클래스의 사용

(5) Pyrex

Pyrex는 Python의 extension 모듈을 작성하기 위해 고안된 언어이다. 이 언어는 오직 Python과 C를 연결하기 위한 용도로만 사용된다. 위에 설명한 SWIG 역시 Python과 C를 연결하기 위한 쉽고 빠른 방법을 제공한다. 그러나 SWIG에서는 Python과 C에서 사용되는 데이터의 구조를 변경하고 싶을 때 Python/C API를 사용해야만 한다. 뿐만 아니라, SWIG는 새로운 built-in Python 타입을 생성하는 기능 역시 제공하지 못한다. Python은 SWIG 만큼이나 쉽고 빠르게 Python과 C의 기본 데이터 타입에 대한 변환을 가능하게 할 뿐만 아니라, 임의의 Python 데이터 구조를 임의의 C 데이터 구조로 변환하는 것도 가능하게 한다. 또한, Python에서 새로운 클래스를 생성하는 것만큼이나 간단하게 새로운 build-in Python 타입을 생성하도록 한다.

(가) Pyrex 구문

Pyrex는 쉽게 이야기하면 C 데이터 타입을 가진 Python 언어이다. 즉, 약간

의 제약사항이 있지만 문법이 Python과 동일하다. 반면, 데이터 타입의 경우엔 Python의 데이터 타입을 모두 지원하면서 동시에 C의 데이터 타입도 지원하는 것이다. 이러한 양쪽 언어의 데이터 타입들은 Pyrex에 의해 자동으로 변환되며, 자동으로 변환되지 않는 일부 타입의 경우엔 사용자가 데이터 선언 시에 직접적으로 코드를 작성하는 방법으로 변환이 가능하도록 한다. Pyrex의 컴파일러는 Pyrex 소스코드를 컴파일해서 Python/C API를 사용하는 C 소스코드로 변환해서 최종적으로 C 컴파일러를 통해 공유 라이브러리를 생성하도록 한다. 아래는 간단한 pyrex 코드의 예이다.

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
       kmax = 1000
    k = 0
    n = 2
    while k < kmax:
       i = 0
       while i < k and n % p[i] <> 0:
          i = i + 1
       if i == k:
          p[k] = n
          k = k + 1
          result.append(n)
       n = n + 1
    return result
```

그림 IV-23. Pyrex 코드의 예

위의 코드에서 보는 바와 같이 Pyrex의 문법은 Python의 그것과 동일하다. 반면, cdef라는 키워드로 C 스타일의 int 타입 변수와 행렬을 선언할 수 있다. 이와 같이 Pyrex에서는 C 타입의 변수나 함수를 정의할 때는 cdef라는 키워드를 사용한다. 반면, Python 타입의 함수를 정의할 때는 def 키워드를 사용한다. 물론, 위의 함수에서 int 타입으로 전달받은 kmax는 함수 내에서 자동으로 C 데이터 타입의 int로 변환된다.

cdef 키워드를 이용하면 일반적인 함수나 변수뿐만 아니라, 다음과 같이 Python에서는 지원하지 않는 포인터나 struct, union, enum 타입도 선언할 수 있다.

```
cdef struct Grail:
    int age
    float volume

cdef union Food:
    char *spam
    float *eggs

cdef enum CheeseType:
    cheddar, edam,
    camembert

cdef enum CheeseState:
    hard = 1
    soft = 2
    runny = 3
```

그림 IV-24. Pyrex에서의 struct, union, enum 선언

Pyrex에는 C언어의 constant를 정의하는 구문이 없다. 그러나 이것은 enum 키워드를 이용해서 구현할 수 있다.

```
cdef enum:
    tons_of_spam = 3
```

그림 IV-25. Pyrex에서의 constant 변수 정의

Pyrex는 C언어의 typedef 역시 지원한다. 키워드 ctypedef를 사용하면 되는데, 예제는 다음과 같다.

```
ctypedef unsigned long ULong
ctypedef int *IntPtr
```

그림 IV-26. Pyrex에서의 타입 정의

- 자동 타입 변환

기본적인 숫자나 문자열의 경우에는 Pyrex가 자동으로 변환을 시킨다. 각 타입별 변환되는 타입은 표VI-1과 같다.

C Types	From Python Types	To Python Types
[Unsigned] char [Unsigned] short int, long	int, long	int
unsigned int unsigned long [unsigned] long long	int, long	long
float, double, long double	int, long, float	float
char *	str	str

표 IV-1. Pyrex의 데이터 타입 변환

숫자를 constant로 사용했을 경우 Pyrex가 자동으로 적합한 Python 또는 C 타입으로 변환한다. 그러나 숫자 뒤에 L을 붙이면 (예: 568L) short로 충분하다 하더라도 long integer 타입으로 변환한다.

- C 구문과 Pyrex 구문의 차이점

* Pyrex에는 -> 연산자가 없다. p->x 대신 p.x를 사용해야 한다.

* Pyrex에는 * 연산자가 없다. *p 대신 p[0]를 사용해야 한다.

* null C 포인터는 null로 사용해야 한다. 0은 안된다. 또한, NULL은 Pyrex에서 예약어이다.

* 타입 캐스팅은 <type>value로 한다.

- C 포인터 처리 방법

Pyrex 및 Python에는 C의 포인터와 같은 타입이 존재하지 않는다. 따라서 포인터를 인자로 받는 C 함수를 extension 모듈로 작성하기 위해서는 다음과 같이 임시 저장 영역을 확보하고 그 주소를 C 함수의 포인터 인자에 넘겨준 뒤에 다시 원래 변수에 복사하는 방식으로 처리해야 한다.

* 하나의 변수만 받는 포인터의 경우 (arg1은 일반 변수, arg2는 포인터)

```
def test(arg1, arg2):
    cdef int tmpArgs

    _test(arg1, &tmpArgs)
    arg2[0] = tmpArgs
```

그림 IV-27. Pyrex의 일반 C 포인터 처리 방법

* 행렬을 받는 포인터의 경우 (arg1은 일반 변수, arg2는 포인터)

```
def test(arg1, arg2):
    cdef int tmpArgs[4]

    _test(arg1, &(tmpArgs[0]))

    for counter in range(3):
        arg2[counter] = tmpArgs[counter]
```

그림 IV-28. Pyrex의 행렬 C 포인터 처리 방법

(나) 컴파일

pyrex를 이용해서 Python extension 모듈을 만들기 위해서는 pyrexc라는 Python으로 작성된 컴파일러를 사용해서 C 소스코드를 생성한 뒤, 이를 다시 C 컴파일러로 컴파일하고, 마지막으로 공유 라이브러리를 생성하면 된다. 그러나 이러한 방법은 번거로우므로 보통은 Python의 distutils 모듈을 이용한다. 다음은 mymodule이라는 이름의 Pyrex 모듈을 생성하는 distutils 설정 파일이다.

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext
setup(
  name = "PyrexGuide",
  ext_modules=[
    Extension("mymodule", ["mymodule.pyx"], libraries = ["xosd"])
    ],
  cmdclass = {'build_ext': build_ext}
)
```

그림 IV-29. Pyrex 컴파일을 위한 Python 코드

이 Python 코드를 setup.py로 저장했을 경우, 이를 실행하기 위한 명령은 다음과 같다.

```
bash$ python setup.py build_ext --inplace
```

그림 IV-30. Pyrex 컴파일 실행 방법

이 모듈을 실제 Python에서 사용하기 위해선 다음과 같이 하면 된다.

```
import mymodule
print dir(mymodule)
```

그림 IV-31. Pyrex 모듈 읽어 들이는 법

(다) 문제점

Pyrex는 아직까지 C++의 클래스는 지원하지 못하므로 C++ 클래스를 위한 extension 모듈을 만들 수 없다.

(6) ctypes

ctypes는 C로 작성된 공유 라이브러리의 C 데이터 타입이나 함수들을 Python에서 불러서 사용할 수 있도록 하는 Python을 위한 외부 함수 인터페이스 패키지(Foreign Function Interface)이다. (다른 말로 wrap 라이브러리라고도 한다.) ctypes는 Python2.3 이상에서부터 사용할 수 있으며 Python2.5부터는 정식으로 포함되었다. ctypes는 SWIG나 pyrex와는 달리 Python 내에서 바로 부를 수 있는 모듈 형식으로 작성되었기 때문에 따로 컴파일러가 존재하지 않아 매우 간단하고 쉽게 사용할 수 있다.

ctypes는 Python에서 간단하게 사용할 수 있도록 cdll이라는 object를 제공한다. 이를 이용해서 공유 라이브러리를 호출할 수 있는데, cdll의 멤버 함수 중 LoadLibrary()함수를 사용해서 libc를 생성한 뒤, 이 object를 이용해서 호출하면 된다.

```
from ctypes import *
cdll.LoadLibrary("libc.so.6")
libc = CDLL("libc.so.6")
print libc.time(None)
```

그림 IV-32. ctypes 모듈 읽어들이는 방법

(가) ctypes 구문

- 데이터 호환

ctypes는 C 언어의 여러 데이터 타입과 호화되는 데이터 타입들을 정의하고 있다. 각 데이터 타입은 자동으로 대응되는 Python 타입으로 변환된다.

ctypes type	C type	Python type
c_char	char	1-char string
c_wchar	wchar_t	1-char unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	int64 or long long	int/long
c_ulonglong	unsigned int64 or unsigned long long	int/long
c_float	float	float
c_double	double	float
c_char_p	char *(NULL terminated)	string or None
c_wchar_p	wchar_t *(NULL terminated)	unicode or None
c_void_p	void *	int/long or None

표 IV-2. ctypes의 데이터 타입

ctypes의 데이터 타입을 이용해서 데이터를 변환하는 방법은 다음과 같다.

```
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
```

그림 IV-33. ctypes를 이용한 C와 Python 간의 데이터 변환 (1)

```
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
c_char_p('Hi, there')
>>> print s              # first string is unchanged
Hello, World
```

그림 IV-34. ctypes를 이용한 C와 Python 간의 데이터 변환 (2)

- 함수 호출

libc.so의 대표적인 함수인 printf를 사용해보자.

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S", u"World!")
Hello, World!
13
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: Don't know
how to convert parameter 2
```

그림 IV-35. ctypes를 이용한 export 함수 호출 (1)

마지막 문장에서 “42.5”는 Python 데이터 타입이기 때문에 C로 작성된 공유 메모리에서 이를 받지 못한다. 따라서 아래와 같이 ctypes 데이터 타입으로 해당 값의 타입을 명확히 설정해야 한다.

```
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
Integer 1234, double 3.1400001049
31
```

그림 IV-36. ctypes를 이용한 export 함수 호출 (2)

ctypes에는 argtypes라는 속성을 이용해서 공유 라이브러리의 함수들의 인자들에 대한 데이터 타입을 미리 명시적으로 설정할 수 있다.

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
```

그림 IV-37. ctypes에서의 export 함수의 인자 데이터 타입 설정

- 포인터

많은 C 함수들은 포인터를 인자로 받는다. 이 경우에는 다음과 같이 byref 키워드를 사용하면 된다. byref 키워드는 변수를 call by reference 형식으로 주소를 전달하도록 한다.

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s",
...             byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
```

그림 IV-38. ctypes에서의 call by reference

(나) 문제점

ctypes 역시 pyrex와 마찬가지로 복잡한 형식의 큰 크기의 C++ class의 경우 변환에 오류가 많았다.

V. 참고문헌

[1] Kniss J, Kindlmann G, Hansen C, Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets, Pr oceedings of IEEE Visualization 2001

[2] Kniss J, Kindlmann G, Hansen C, Multidimensional Transfer Functions for Interactive Volume Rendering, IEEE Transactions on Visualization and Computer Graphics, Vol. 8, No. 3, pp. 270-285, 2002

[3] Rappin N, Dunn R, wxPython In Action, Manning, 2006

[4] 백종현, 유승우, Python How To Program, Pearson, 2002

[5] wxPython Homepage, http://www.wxpython.org

[6] wxWidgets Homepage, http://wxwidgets.org

[7] SWIG Homepage, http://www.swig.org

[8] Pyrex Homepage, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex

[9] Michael's Quick Guide to Pyrex, http://ldots.org/pyrex-guide

[10] ctypes Homepage, http://starship.python.net/crew/theller/ctypes/