Direct2D 사용하기

매킨토시나 Windows 3.1 이 등장하면서 운영체제는 GUI(Graphical User Interface)의 시대를 맞게 된다. 이는 기존의 문자 기반의 CUI(Character-based User Interface)와는 달리 마우스라는 새로운 입력 장치를 도입하여 클릭하면서 더욱 편리한 인터페이스를 제공하였는데, 자연스레 그러면서 프로그램을 좀 더 멋지게 꾸미고 싶은 욕망도 점점 커져갔다. 또한 당시 MUD라고 불리던 텍스트 게임들이 그래픽으로 바뀌어가기 시작했고, 동영상과 같은 미디어들을 컴퓨터를 통해서 재생하고 볼 수 있게 되었다.

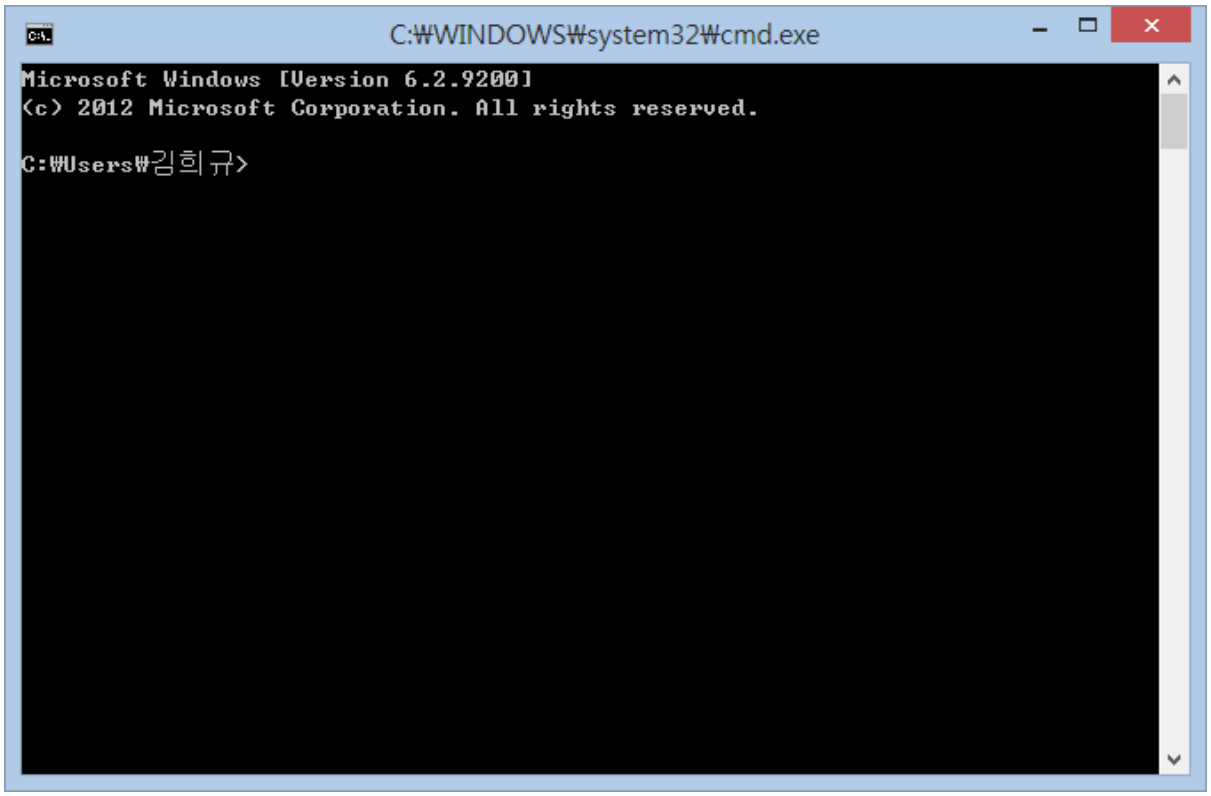

그림 1. CUI 기반의 Microsoft Command Line Window

뿐만 아니라 컴퓨터의 성능도 발전하고 NVIDIA와 같은 기업들이 그래픽 가속기를 개발하면서 컴퓨터의 렌더링 성능이 점점 높아졌다. 이에 따라 기존에 논문으로만 존재했던 기술들을 실제로 볼 수 있게 되었다. 그래픽은 컴퓨터의 보급에 결정적인 역할을 하였다.

각각의 운영체제는 CPU를 통해서 화면에 렌더링을 하는 API들을 제공하고 있었다. 하지만 이는 당시의 느린 성능과, 운영체제마다 다른 사용법들이 문제가 되었다. 당시 그래픽 가속기를 사용하면 하드웨어 가속(Hardware Acceleration)을 통해서 더 빠른 속도로 렌더링을 할 수 있었다. 이를 위해선 어셈블리어로 그래픽 가속기를 직접 조작하는 방법 외에는 없었다. 그런데 여기서도 문제가 된 것이, 그래픽 가속기 역시 종류마다 사용법이 다르기에 각각 다른 어셈블리 코드를 작성하며 개발자들은 많은 고초를 겪었다.

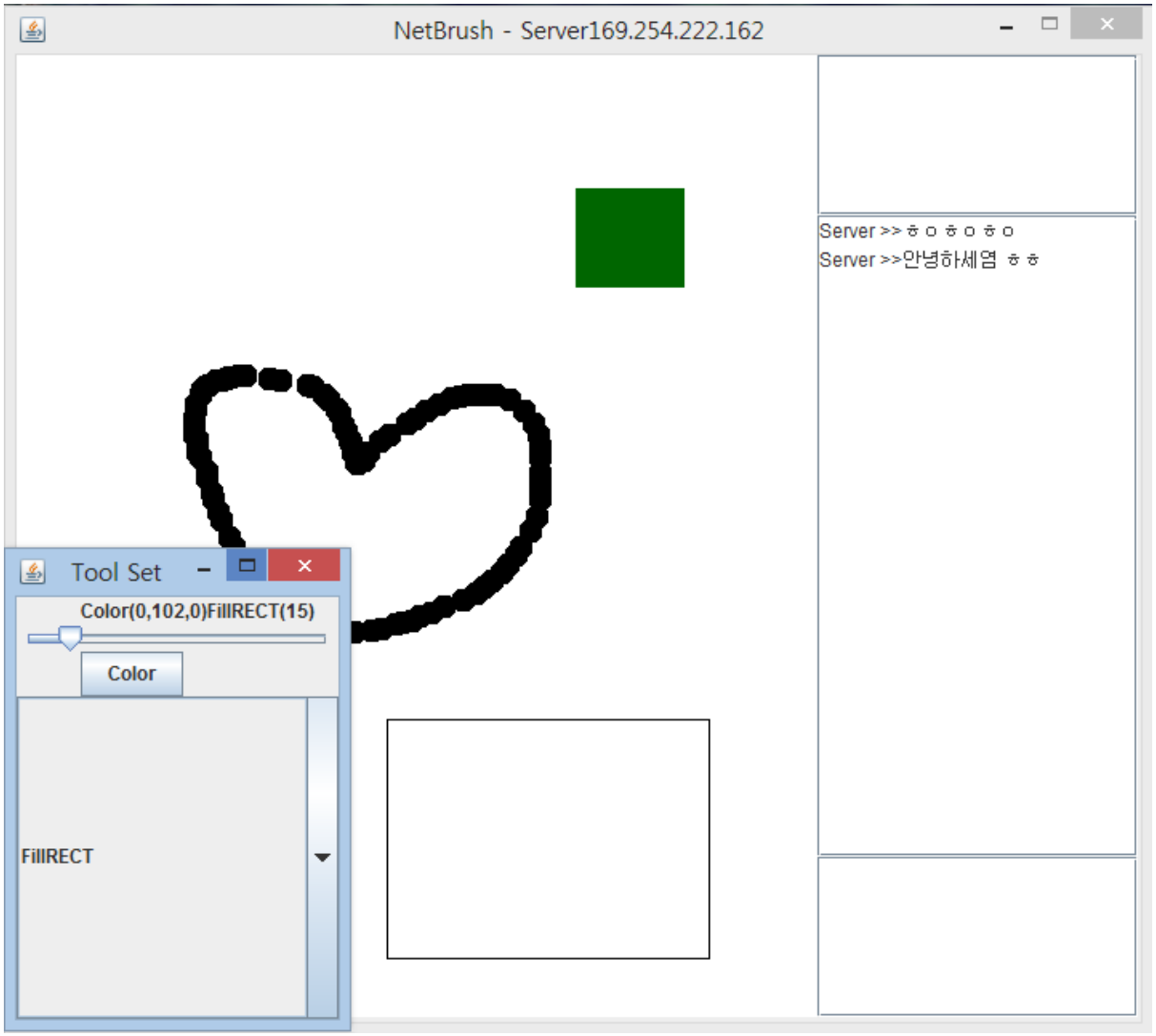

그림 2. GUI 기반의 윈도우 어플리케이션

그러한 상황에서, 개발자들은 모든 운영체제에서 공통적으로 동작할 수 있고, 하드웨어 가속 기능을 사용할 수 있는 그래픽 API를 원했고, 이에 등장한 것이 SGI(Silicon Graphics International)에서 개발한 OpenGL(Open Graphics Library)이다. OpenGL은 순수하게 C언어로 구현되어 있었으며, 화면에 렌더링하는 기능만을 포함하고 플랫폼 종속적인 기능들은 전혀 넣지 않아 다양한 운영체제에서 사용 가능했다. SGI는 자신들의 API가 더 널리 퍼지게 하기 위해선 자유 소프트웨어로 배포해야 한다고 생각했고, 그들의 예상은 정확하게 들어맞았다. 개발자들은 OpenGL API의 표준 규격들을 아무런 제약 없이 구현하여 배포할 수 있다. 예를 들어, 그래픽 카드 제조회사인 NVIDIA는 그래픽 카드에 OpenGL의 규격에 맞는 기능들을 추가하고 그 API를 구현하여 드라이버로 배포한다. 이러한 방식으로 OpenGL은 온갖 언어와 플랫폼을 뛰어넘는 그래픽스 업계의 사실상의 표준으로 자리잡을 수 있게 되었다.

Microsoft는 OpenGL의 대항마로서 윈도우용 게임을 만드는 개발자들을 위해서 1995년 DirectX를 만들었다. 처음에는 Game SDK라는 이름으로 발표되었지만, 그 기능들을 DirectPlay,

DirectDraw와 같이 이름 지었는데 이를 언론에서 DirectX('Direct어쩌고' 라는 의미)라고 부른 뒤 아예 이름으로 바뀌었다고 한다. DirectX는 SDK(Software Development Kit)이다. SDK란 소프트웨어 개발에 필요한 다양한 도구들을 모아둔 것을 말한다. API가 단순히 프로그래밍 라이브러리인 데 비하면, 현재 최신 버전의 DirectX SDK(June 2010)에는 명령 프롬프트, 예제 코드들과 브라우저 어플리케이션 등이 들어있다.

DirectX는 OpenGL과는 달리 Windows 계열의 운영체제와 Xbox, PlayStation과 같은 일부 콘솔 플랫폼에서 사용 가능하다. 또한 프로그래밍 언어는 C++이고, COM이라는 Windows 고유의 프로그래밍 패턴을 따른다. 이 COM역시 깊이 파고들면 책 한 권은 거뜬히 나오는 분량이므로 부록에서 간단하게만 다룰 것이다. 또한 OpenGL과 달리 이미지나 글꼴, 윈도우 관리와 같은 보조 기능들도 제공된다. 또한 Windows에 특화된 API들을 제공하기 때문에 Windows 개발을 한다면 DirectX가 더 편리할 것이다. 하지만 반드시 알아둬야 하는 건, DirectX와 OpenGL의 그래픽스 성능은 차이가 거의 없다는 것을 알아두어야 한다. 아마 DirectX vs OpenGL이라고 검색하면 관련 자료를 매우 많이 찾을 수 있다. OpenGL역시 순수 OpenGL에만 보조기능들이 없을 뿐 수많은 사람들이 오픈 소스로 다양한 보조 라이브러리들을 제작하여 배포하고 있다. 하지만 아무래도 성능은 좀 구린 게 사실이지만 OpenGL의 이점을 생각하면 눈감아줄 수 있다.

Direct2D는 D2D1.h와 보조 기능들로 이루어진 D2D1Helper.h로 구성되어 있다. D2D1Helper.h의 모든 내용들은 네임스페이스 D2D1 안에 들어 있다. 구분을 위해 예제에서는 네임스페이스를 using하지 않고 D2D1:: 을 붙일 것이다. Direct2D는 2D 그래픽 API로써, 단순히 윈도우에 그림을 그리는 것뿐만 아니라 DXGI 표면이나 비트맵, Windows API의 DC(Drawing Context)에도 그리기 작업을 수행할 수 있게 해준다. 하지만 우리는 윈도우를 대상으로 한 2D 그리기만을 살펴볼 것이다.

그리기 작업을 수행하기 위해서는 먼저 Direct2D 팩토리를 생성하고, 팩토리를 통해서 그리기 명령을 수행하는 렌더타겟(Render Target)을 생성한다. 렌더타겟은 그리기 작업 외에도 다양한 리소스 들을 생성할 수 있다. 팩토리 역시 렌더타겟 외에도 도형 객체를 만들거나 시스템 DPI를 읽어오는 기능들을 가지고 있다. DPI에 대해선 뒤에서 설명한다.

일단 Direct2D 유틸리티 함수를 넣어둘 헤더와 소스 파일인 d2dutils.h 와 d2dutils.cpp를 생성한다. 앞으로 이 두 파일에 다양한 기능들을 구현한 함수들을 집어넣을 것이다. 그리고 아래의 예제를 입력해 보자.

d2dutils.h

```
// include guard
#ifndef _d2dutils_h_
#define _d2dutils_h_

#include <d2d1.h>
```

```
#include <d2d1helper.h>

// 정적 라이브러리 링크
#pragma comment(lib, "d2d1.lib")

namespace d2d {

// Direct2D 팩토리와
// 윈도우 렌더타겟을 생성해 주는 함수
HRESULT Initialize (
	HWND hwnd,
	ID2D1Factory **factory,
	ID2D1HwndRenderTarget **rt
	);

// COM 인터페이스를 해제해주는 함수
template <typename T>
inline void SafeRelease(T t) {
	if(t)
		t->Release();
}

template <typename T>
inline void SafeDelete(T t) {
	if(t)
		delete t;
}
}
#endif
```

모든 기능들은 구분을 위해 네임스페이스 d2d 안에 들어있다. SafeRelease 함수와 SafeDelete 함수는 포인터가 NULL이 아닌 경우에만 해제와 삭제를 수행하게 하는 보조 함수이다. Initialize 함수는 윈도우 핸들로부터 Direct2D 팩토리와 윈도우에 그림을 그리는 렌더타겟(ID2D1HwndRenderTarget)객체를 생성하고, 인자로 온 주소 값에 넣어 준다. 반환형은 HRESULT 타입으로, 초기화 과정에서 생긴 문제를 나타내는 상수를 반환해 준다. 만약 없다면 S_OK가 반환된다. 이 함수의 구현은 아래와 같다.

#pragma 이후의 전처리기 코드는 정적 라이브러리를 읽어주는 기능을 한다. Direct2D와 관련된 기능들은 전부 dll 파일 안에 들어있는데, 이 dll 파일에서 로드하는 내용들을 lib파일이 가지고 있다. 라이브러리를 링크하면 자동적으로 dll파일에서 Direct2D API를 로드하며, 만약 dll 파일을 찾지 못한다면 dll파일을 찾을 수 없다는 에러 메시지와 함께 실행이 취소 될 것이다.

d2dutils.cpp

```
#include "d2dutils.h"
using namespace d2d;

HRESULT d2d::Initialize (
	HWND hwnd,
	ID2D1Factory **factory,
	ID2D1HwndRenderTarget **rt
	)
{
	HRESULT hr = S_OK;

	//
	// Direct2D 팩토리를 생성
	hr = D2D1CreateFactory (
		D2D1_FACTORY_TYPE_SINGLE_THREADED,
		factory );


	if(SUCCEEDED(hr))
	{
		RECT client;
		GetClientRect(hwnd, &client);

		hr = (*factory)->CreateHwndRenderTarget (
			D2D1::RenderTargetProperties (),
			D2D1::HwndRenderTargetProperties(
					hwnd,
					D2D1::SizeU(
						client.right - client.left,
						client.bottom - client.top )
					),
			rt );

	}

	return hr;
}
```

먼저 D2D1CreateFactory 함수로 팩토리 객체를 생성한다. 첫 번째 인자는 팩토리의 타입인데

D2D1_FACTORY_TYPE_SINGLE_THREAD 는 팩토리를 싱글스레드 기반의 객체로 생성하란 의미이다. 이렇게 될 경우에 멀티스레드 보안 기능들을 사용할 수 없게 된다. 여기선 싱글스레드만을 다룬다.

이후 SUCCEEDED(hr) 매크로 함수로 반환 결과인 hr이 성공했는지를 알아낸다. 만약 성공했다면 ID2D1Factory 객체가 생성된 것이므로 팩토리 객체를 통해서 윈도우 렌더타겟 객체를 생성한다. 윈도우 렌더타겟 객체를 생성하기 위해선 윈도우 핸들(HWND)과 크기를 전달해야 하는데, 윈도우의 크기를 알아내기 위해서 GetClientRect 함수를 썼다.

CreateHwndRenderTarget() 메서드는 윈도우 렌더타겟을 생성하기 위해서 2개의 인자를 받는다. 첫 번째는 D2D1_RENDER_TARGET_PROPERTIES 구조체로, 렌더타겟의 속성들의 대한 정보를 지니나 D2D1::RenderTargetProperties() 함수를 이용하면 간편하게 기본 값들을 사용할 수 있다. 두 번째 인자는 D2D1_HWND_RENDER_TARGET_PROPERTIES 구조체로 윈도우 핸들과 관련된 인자들이 있다. 필수적으로 윈도우 핸들과 윈도우의 크기가 들어가야 한다. 역시나 D2D1::HwndRenderTargetProperties() 함수로 간단하게 만들 수 있다.

그러면 이제 Direct2D를 이용하기 위한 WinMain 코드를 보자. 기존의 SimpleWindow.cpp 에서 바뀐 코드는 두껍게 표시했다. 또한 WNDCLASS 구조체를 채울 때 hbrBackground 필드를 더 이상 채우지 않는다.

CustomWinMain.cpp

```
#include <Windows.h>
#include "d2dutils.h"


int width = 800,
	height = 600;
HWND hwnd;
ID2D1Factory *factory2d = 0;		// 팩토리
ID2D1HwndRenderTarget *rt = 0;		// 윈도우 렌더타겟

// 화면을 그릴 때 호출되는 함수
void OnDraw()
{
}

// 리소스들을 생성하는 함수
bool Setup()
{
	return true;
}
```

```
// 리소스들을 제거하는 함수
void CleanUp()
{
	d2d::SafeRelease(rt);
	d2d::SafeRelease(factory2d);
}

//
// 메세지 프로시저 함수
// 이벤트가 발생했을 때 이 함수가 호출된다.
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wp, LPARAM lp)
{
	switch(msg)
	{
	// 윈도우가 그려질 때
	case WM_PAINT:
		OnDraw();
		return 0;
	//
	// 윈도우가 파괴되었을 때
	case WM_DESTROY:
		// 나가라!
		PostQuitMessage(0);
		return 0;

	// 처리되지 않은 메시지는
	default:
		// 기본 처리 함수를 이용한다.
		return DefWindowProc(hwnd, msg, wp, lp);
	}
}

int WINAPI WinMain(HINSTANCE hinst, HINSTANCE, LPSTR, int)
{
	//
	// 윈도우 클래스를 등록한다.(hbrBackground 사라짐)
	WNDCLASS wc = {0};
	wc.lpszClassName = L"Unipen Window";
	wc.style = CS_VREDRAW | CS_HREDRAW;
	wc.lpfnWndProc = WindowProc;
	wc.hInstance = hinst;
	RegisterClass(&wc);

	DWORD style = WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX;
	RECT rect = {0, 0, width, height};
```

```
	AdjustWindowRect(&rect, style, false);

	hwnd = CreateWindow (
		wc.lpszClassName,		// 창 클래스
		L"Custom Direct2D Window",	// 창 제목
		style, // 창 스타일
		CW_USEDEFAULT,			// 화면에서의 X 좌표
		CW_USEDEFAULT,			// 화면에서의 Y 좌표
		rect.right - rect.left,		// 가로 크기
		rect.bottom - rect.top,		// 세로 크기
		0,				// 부모 윈도우
		0,				// 메뉴
		hinst,				// 인스턴스 핸들
		0				// 초기 전달값
		);

	// 그리기 준비
	if(FAILED(d2d::Initialize(hwnd, &factory2d, &rt))
		|| !Setup())
	{
		// 실패할 경우 에러 메세지
		MessageBox(0, L"Setup-Failed", L"Error", MB_ICONERROR);
	}
	else
	{
		//
		// 윈도우 창을 띄운다.
		ShowWindow(hwnd, SW_SHOW);

		//
		// 메시지 루프를 돈다.
		MSG msg = {0};
		while(GetMessage(&msg, 0, 0, 0))
		{
			TranslateMessage(&msg);
			DispatchMessage(&msg);
		}
	}

	// 청소
	CleanUp();
	return 0;
}
```

먼저 전역 변수로 width, height, hwnd가 추가 되었다. width와 height은 윈도우의 가로, 세로 크기이고, hwnd는 윈도우 핸들이다. 또한 factory2d와 rt 는 Direct2D 팩토리와 윈도우 렌더타겟 객체이다. Draw()함수는 화면에 그릴 때 호출되는 함수로, 이 안에 핵심적인 내용들이 많이 들어갈 것이다. Setup() 함수는 그리기에 필요한 자원들을 생성하는 함수로, 성공한다면 true를, 실패한다면 false를 반환하게 될 것이다. CleanUp() 함수는 Setup() 함수에서 만든 자원들을 청소하는 함수이다.

WindowProc() 메시지 프로시저 함수에는 WM_PAINT 라는 이벤트를 처리하는 내용이 추가되었다. WM_PAINT 메시지는 윈도우가 그려질 때 호출된다. Windows는 시스템 자원을 낭비하지 않기 위해서 필요한 상황이 아니면 윈도우를 다시 그리지 않는다. 만약 윈도우가 최소화 되었다가 다시 켜지거나, 윈도우 크기가 변경되는 등과 같은 상황에서 윈도우를 그려야 할 경우 WM_PAINT 메시지가 전달되면서 그리기 작업을 수행하게 된다. 위에선 OnDraw() 함수를 호출한다.

이후에, ShowWindow 함수를 통해 윈도우를 띄우기 직전에 Direct2D 초기화 함수를 호출해서 팩토리와 렌더타겟을 만들고, 우리가 선언한 Setup() 함수를 호출하여 자원 생성을 한다. 그것들 중 하나라도 실패했다면 에러 메시지를 띄우고 그렇지 않다면 윈도우를 띄우고 메시지 루프를 실행한다. 그리고 WinMain이 종료되기 직전에 CleanUp() 함수를 호출해 줌으로써 자원들을 제거한다.

앞으로는 전체적인 소스 코드의 분량을 줄이기 위해 가급적 새로 추가된 전역 변수와 OnDraw(), Setup(), CleanUp() 세 함수만을 보여줄 것이다. 그 이외에 내용들이 추가 되어야 한다면 그 때 따로 알리도록 하겠다.

화면 지우기

렌더타겟이 그리기 위해선 먼저 BeginDraw() 메서드를 호출해서 그리기 시작했다는 것을 알려야 한다. 이후에 EndDraw() 함수로 그리기 작업을 다 완료했음을 알린다. 이런 과정을 거치는 이유는 Direct2D가 다른 그래픽 API와 연동할 때 오류를 피하기 위함이다. BeginDraw()와 EndDraw() 사이에서 Draw***, Fill***, 혹은 Clear()와 같은 메서드를 호출해서 그리기 작업을 수행한다.

먼저 간단하게 화면을 지우는 예제를 살펴보자. 화면을 지우는 메서드는 렌더타겟의 Clear() 메서드로, 화면을 지울 색상을 인자로 받는다. 여기서 '지운다' 라는 말은 실제론 화면을 특정 색으로 완전히 채우는 것과 같다.

색상과 이미지

우선은 색상에 대해서 알아보아야 한다. 색상은 빛의 파장이 우리 눈에 보이는 현상이다. 이 빛을 나타내는 방식에는 여러 종류가 있는데, 이 종류를 색상 형식(Color Format)이라고 한다. 대표적인 색상 형식으로는 RGB, CMYK, HSB(혹은 HSV)등이 있다. 이 중 우리가 사용할 색상 형식은

대표적으로 컴퓨터 그래픽스 분야에서 많이 사용되는 RGB이다.

RGB는 빛의 3원색인 빨강(Red), 초록(Green), 파랑(Blue)를 혼합하여 색상을 나타내는 방식을 의미한다. 이 때 R, G B를 각각 색상의 원소(Component)라고 한다. RGB를 컴퓨터에서 나타낼 때에는 제한된 메모리를 사용해야 한다. 한 색상을 나타내기 위해서 사용되는 메모리의 크기를 색상 깊이(Depth)라고 한다. 예를 들어 RGB각 원소에 1바이트의 메모리를 사용한다면 R8G8B8과 같이 비트로 표시하고, 색상 깊이는 24비트라고 한다. 24비트는 16777216가지의 수를 표현할 수 있으므로 나타낼 수 있는 색상 역시 그와 비슷하다. 대부분 16비트 이상의 깊이를 가지는 색상들은 우리가 눈으로 구별하기가 쉽지 않다. 그렇기 때문에 이를 트루 컬러(True Color)라고 한다.

RGB의 특징은 모든 요소가 0에 가까워질수록 어두워지고, 최대값에 가까워질수록 밝아진다. 또한 특정 원소의 값이 클수록 그 원소에 가까운 색이 나타난다. 아래의 표는 RGB색상의 예시이다.

Red(0 ~ 255)	Green(0 ~ 255)	Blue(0 ~ 255)	Color
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	255	White
255	255	0	Yellow
255	0	255	Magenta
0	255	255	Cyan
0	0	0	Black
240	255	240	Honeydew
70	146	180	Steel Blue

Direct2D는 색상을 나타내기 위해 D2D1_COLOR_F 구조체를 제공한다. 이 구조체는 아래와 같이 구성되어 있다.

```
struct D2D1_COLOR_F {
        float r, g, b, a;
}
```

기본적으로 컴퓨터 그래픽스 분야에서 화면에 색상을 나타내기 위해서 RGB를 많이 사용한다. 이유는 모니터의 작은 화소(Pixel)이 빨강, 초록, 파랑 3개의 자그만한 화소들로 나뉘어져서 색상을 나타내기 때문이다. 하지만 위 구조체에선 a라는 멤버 변수가 하나 추가되어 있다. 이 값은 색상의 불투명도(Opacity)를 나타내기 위해 존재한다. 불투명한 색상은 기존의 배경 위에 그려질 때, 배경색의 일부와 혼합된다. 이 때 혼합되는 방식을 블렌딩(Blending) 혹은 컴포지트(Composite)라

고 한다. 우리가 사용할 Direct2D에서는 블렌딩 방식을 직접 바꿀 수는 없다.

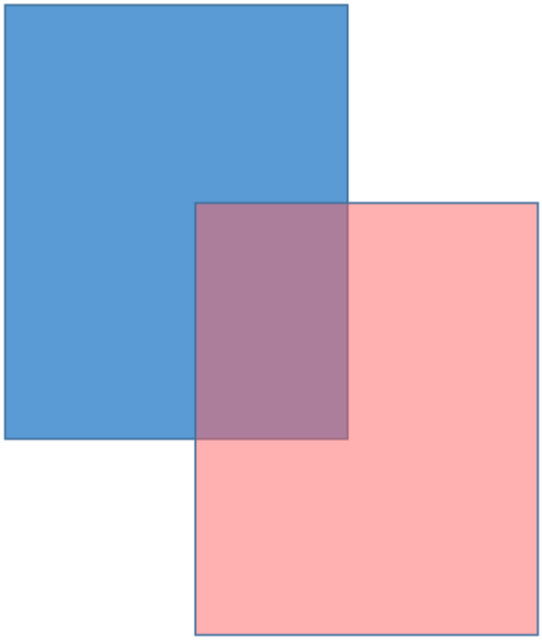

그림 3. 파란색 사각형 위에 반투명한 분홍색 사각형을 그렸다.

D2D1_COLOR_F 구조체는 별다른 기능이 존재하지 않아 사용에 불편함이 많다. 그래서 D2D1Helper.h에서는 D2D1::ColorF 클래스를 제공해서 손쉽게 색상 객체를 사용할 수 있게 도와준다.

```
ID2D1Factory *factory2d = 0;            // 팩토리
ID2D1HwndRenderTarget *rt = 0;    // 윈도우 렌더타겟

// 화면을 그릴 때 호출되는 함수
void OnDraw()
{
	// 렌더타겟에 그릴 때는 반드시 BeginDraw() 메서드와
	// EndDraw() 메서드의 호출 사이여야한다.

	rt->BeginDraw();

	// 색상 생성, 빨간색(R = 1, G = 0, B = 0, Alpha = 1)
	D2D1::ColorF color(1, 0, 0, 1);
	rt->Clear(color);

	rt->EndDraw();
}
```

예제 1. ClearWindow.cpp

위 예제는 ID2D1RenderTarget::Clear() 메서드를 이용해서, 윈도우의 화면을 빨간색으로 바꾸는 예제이다. 색상을 나타내는 대에는 D2D1::ColorF 클래스를 사용했다. 위에서도 말했지만 OnDraw() 이외의 소스 코드는 생략되어 있다. 이 클래스의 생성자를 이용해서 간단히 색상 객체를 생성할 수 있다. 결과는 아래와 같다.

그림 4. ClearWindow.cpp 의 결과 화면

도형 그리기

Draw*** 함수는 도형의 외각선 만을 그리는 데 비해서 Fill*** 메서드는 도형의 안까지 채우는 함수이다. 도형을 어떻게 그릴지를 나타내는 객체가 바로 브러시(Brush)이다. 선(Line)과 비트맵(Bitmap)역시 그릴 수 있다. Direct2D에서 선은 시작과 끝 두 개의 점으로 이루어진 선분(Segment)이며 비트맵은 색상들이 이루는 평면 데이터를 말한다.

Direct2D는 여러 가지 간단한 도형들을 그릴 수 있다. 각각의 도형들을 나타내는 구조체와 도

형 인터페이스를 갖고 있다. 이들은 공통적으로 ID2D1Geometry를 상속한다. Direct2D는 더 복잡한 도형도 지원한다. 경로(Path)도형도 지원하는데, 경로는 구불구불한 뱀이나 복잡한 다각형을 만드는 도구이다. 또한 도형들을 묶어서 그룹으로 만든 뒤에 특별한 규칙을 적용하여 그려낼 수도 있다. ID2D1Geometry 의 하위 클래스 객체들은 렌더타겟의 DrawGeometry()나 FillGeometry() 메서드를 통해서 그려내고, 구조체들은 Draw***(), Fill***()메서드를 통해서 그려내게 된다.

이런 도형이나, 도형 그룹 객체는 ID2D1Factory 인터페이스의 Create<도형>Geometry 메서드로 생성할 수 있으며, CreateGeometryGroup()메서드로 그룹을 생성할 수도 있다. 자세한 건 MSDN Direct2D API 문서를 참고하길 바란다.[1]

종류	구조체	인터페이스
직사각형(Rectangle)	D2D1_RECT_U D2D1_RECT_F	ID2D1RectangleGeometry
둥근직사각형 (Rounded Rectangle)	D2D1_ROUNDED_RECT	ID2D1RoundedRectangleGeometry
타원(Ellipse)	D2D1_ELLIPSE	ID2D1EllipseGeometry
경로(Path)	없음	ID2D1PathGeometry

브러시에도 여러가지 종류가 있다. 그레디언트(Gradient)란, 한 색에서 다른 색으로 서서히 변해가는 것을 의미한다.

브러시 이름	인터페이스	설명
단색 브러시	ID2D1SolidColorBrush	단색으로 그리는 브러시이다.
선형 그레디언트 브러시	ID2D1LinearGradientBrush	직선 방향으로 그레디언트를 그리는 브러시이다.
방사형 그레디언트 브러시	ID2D1RadialGradientBrush	한 점을 중심으로 방사하는 그레디언트를 그리는 브러시이다.
비트맵 브러시	ID2D1BitmapBrush	

이 브러시들은 그림 5에 순서대로 나와 있다. 이들은 렌더타겟의 Create<브러시종류>Brush() 메서드로 만들 수 있다. 선형 그레디언트와 방사형 그레디언트는 색상 중지점(Color Stop)을 지정해서 여러 가지 색으로 걸쳐서 변화하도록 할 수 있다.

---

[1] MSDN, Direct2D, http://msdn.microsoft.com/en-us/library/windows/desktop/dd370990(v=vs.85).aspx

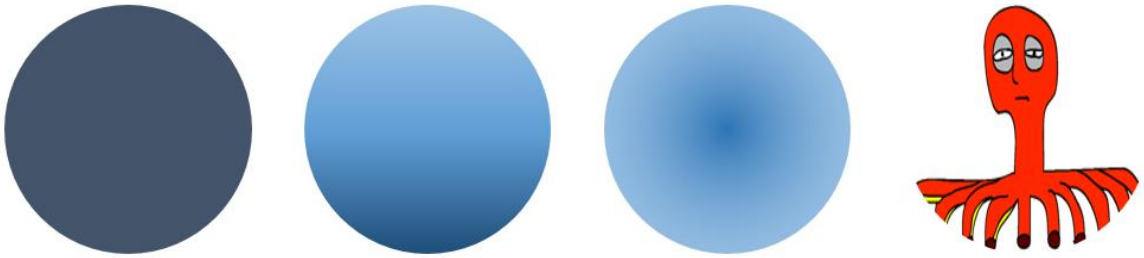

그림 5. 브러시에 따라 다른 원 그리기

아래 예제는 도형들을 그리는 예이다.

GeometryDrawing.cpp

```
// 단색 브러시
ID2D1SolidColorBrush *solidBrush = 0;
ID2D1Geometry *geom[3];

// 화면을 그릴 때 호출되는 함수
void OnDraw()
{
	rt->BeginDraw();
	// 하얀색으로 지우기
	rt->Clear(D2D1::ColorF(1, 1, 1, 1));

	for(int i = 0; i < 3; ++i)
	{
		rt->DrawGeometry(geom[i], solidBrush);
	}

	rt->EndDraw();
}

// 리소스들을 생성하는 함수
bool Setup()
{
	rt->CreateSolidColorBrush (D2D1::ColorF(0, 0, 1), &solidBrush);

	ID2D1RectangleGeometry *rect;
	assert_hr(
	factory2d->CreateRectangleGeometry (
		D2D1::RectF(0, 0, 100, 100),
		&rect )
	);
```

```
        ID2D1RoundedRectangleGeometry *roundedRect;
        assert_hr(
        factory2d->CreateRoundedRectangleGeometry (
               D2D1::RoundedRect (
                      D2D1::RectF(100, 100, 200, 200),
                      20,
                      20 ),
               &roundedRect )
        );

        ID2D1EllipseGeometry *ellipse;
        assert_hr(
        factory2d->CreateEllipseGeometry (
               D2D1::Ellipse (
                      D2D1::Point2F(300, 300),
                      100, 50),
               &ellipse)
        );

        geom[0] = rect;
        geom[1] = roundedRect;
        geom[2] = ellipse;

        return true;
}

// 리소스들을 제거하는 함수
void CleanUp()
{
        for(int i = 0; i < 3; ++i)
               d2d::SafeRelease(geom[i]);
        d2d::SafeRelease(solidBrush);
        d2d::SafeRelease(rt);
        d2d::SafeRelease(factory2d);
}
```

D2D1 네임스페이스 안의 유틸리티 함수들을 이용하면 Direct2D 구조체들을 쉽게 얻을 수 있다. OnDraw() 함수에서 렌더타겟의 BeginScene() 이후에 곧바로 Clear() 메서드가 호출되는데, 이 메서드는 렌더타겟의 대상을 한가지 색으로 지워버린다. 여기서는 윈도우 렌더타겟이므로 윈도우가 통째로 지워지게 된다. 지우는 색상을 나타내는 D2D1_COLOR_F 구조체 값은 **D2D1::ColorF 클래스**로 간편하게 얻을 수 있다. 이후 반복문을 통해 DrawGeometry() 메서드를 호출하며 geom[] 배

열의 도형들을 그려낸다. 실행하면 그림 6처럼 나온다.

Setup() 메서드에서는 제일 먼저 단색 브러시 객체를 생성하고, 이후 직사각형, 둥근 직사각형, 타원 객체를 생성한다. 각각의 구조체는 D2D1::<도형> 함수를 통해 간편하게 생성한다. 직사각형 구조체(D2D1_RECT_F)는 직사각형의 왼쪽(left), 위(top), 오른쪽(right), 아래(bottom) 4가지로 구성되어 있다. 둥근 직사각형은 직사각형 정보에 둥근 모서리의 x, y축 길이(radiusX, radiusY)를 갖고 있으며, 타원은 원의 중심점(D2D1_POINT_2F)과 가로 반지름(radiusX), 세로 반지름(radiusY)을 갖고 있다.

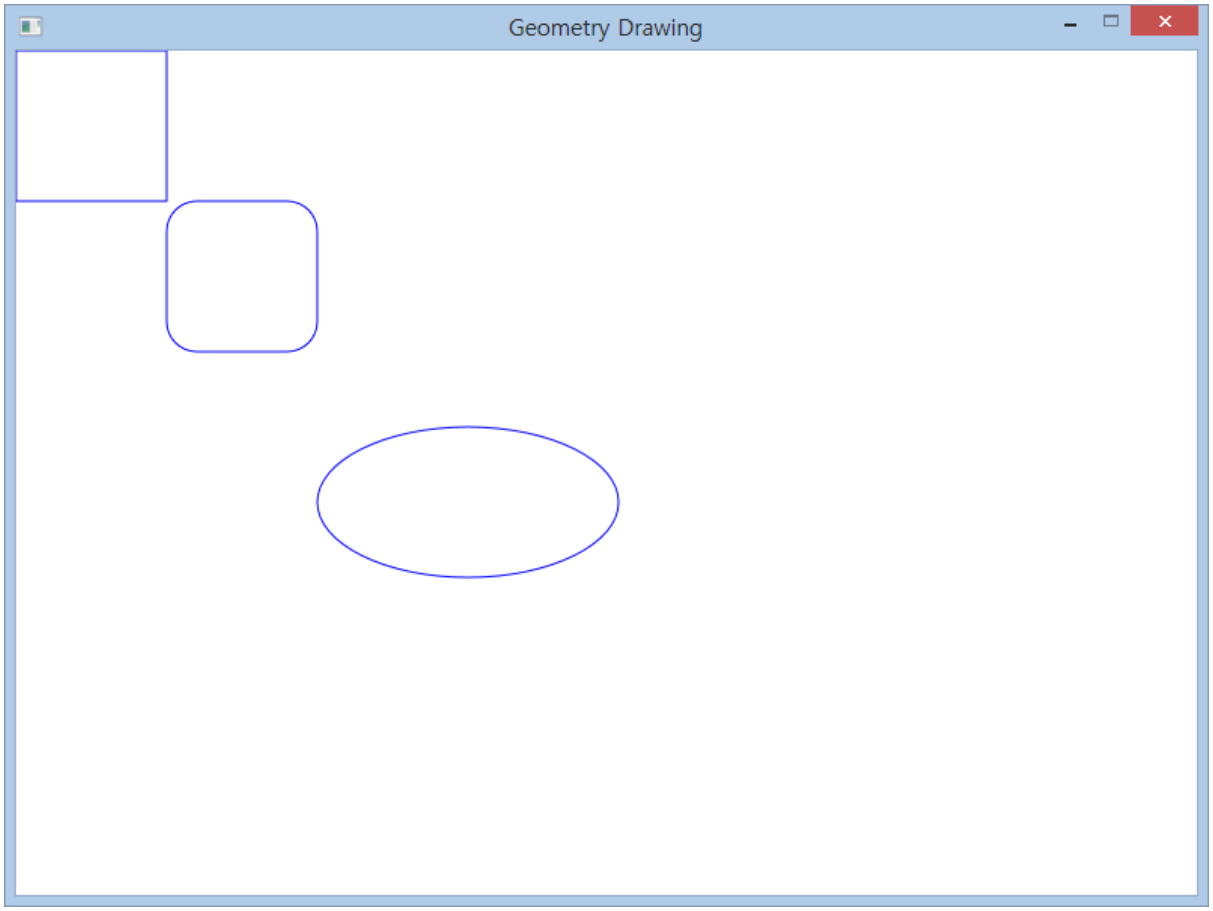

그림 6. GeometryDrawing.cpp의 결과 창

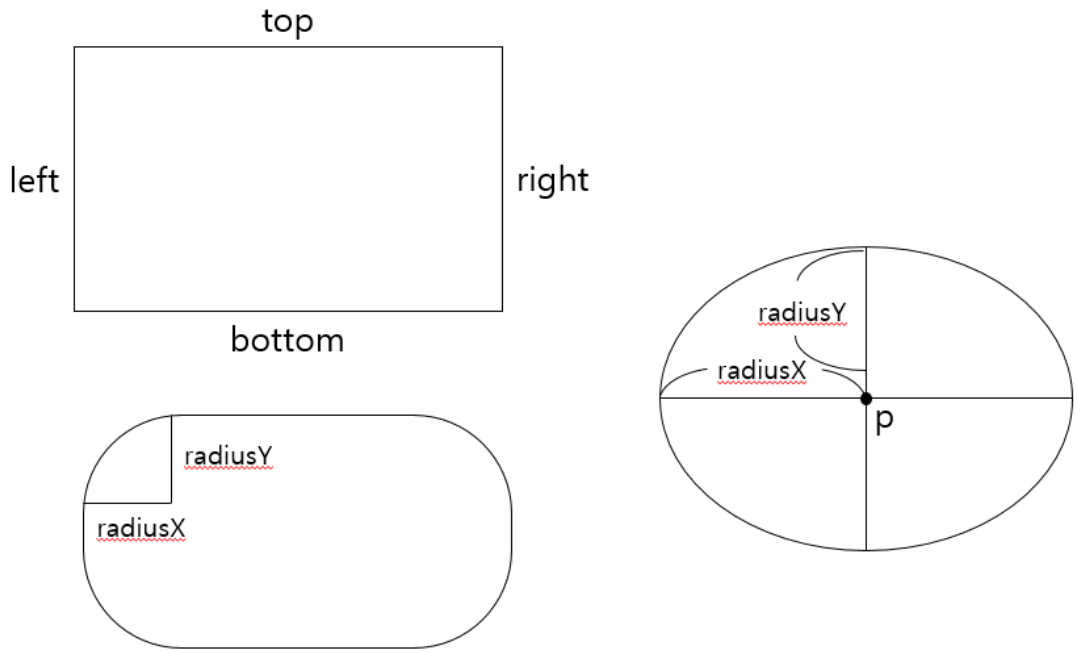

그림 7. 도형들과 각 요소

안티엘리어싱

컴퓨터 그래픽스에서의 안티엘리어싱(Anti-Aliasing)이란, 계단현상(Alias)를 해결하기 위한 드로잉 방식을 말한다. 계단현상이란 도형을 그리는 과정에서 도형의 모서리 부분이나 직선에서 발생하는 픽셀이 깨지며 계단 모양처럼 되는 현상을 의미한다. 아래의 그림은 안티알리아싱의 한 예이다.

그림 8. 안티알리어싱 처리된 문자(아래)와 처리하지 않은 문자(위)

안티알리아싱은 균열이 일어나는 부분에 필터 처리를 하여 더 부드럽게 만든다. 아래 그림을 보면 안티알리아싱 된 오른쪽 문자의 테두리 주변에 투명한 색상들이 추가되어 더 부드러운 문자를 그려내고 있음을 알 수 있다.

그림 9 그림 8를 확대한 결과

Direct2D에서는 렌더타겟의 SetAntialiasMode(D2D1_ANTIALIAS_MODE) 메서드와 SetTextAntialiasMode(D2D1_TEXT_ANTIALIAS_MODE) 메서드로 도형과 문자의 안티엘리어싱 모드를 지정할 수 있다. 도형의 안티엘리어싱은 PER_PRIMITIVE(도형 안티엘리어싱)와 ALIAS(엘리어싱) 두 가지 옵션만을 제공하지만 문자의 안티엘리어싱 모드는 최적화를 위해 다양한 값들이 존재한다.

DEFAULT 는 시스템 설정에 따라 다른 기본 안티엘리어싱 모드를 선택하게 한다.[2] CLEARTYPE기본 값으로 클리어타입 폰트의 안티엘리어싱 모드를 사용하게 한다. 만약 성능과 약간이나마 타협하고 싶다면 GRAYSCALE을 사용해서 품질은 조금 떨어지더라도 더 빠른 방식을 사용할 수 있다. 안티엘리어싱 기능을 사용하기 싫다면 ALIAS를 지정해서 비활성화 할 수도 있다. 글꼴을 그리는 자세한 방법에 대해서는 이후의 장에서 다루도록 하겠다.

이 장에서는 다른 API와 Direct2D를 함께 사용하는 방법에 대해서 알아볼 것이다. 사실 도형만 가지고도 모든 것을 할 수 있다. 그러나 그러지 않는 이유는 글꼴이나 이미지와 같은 것들은 도형만으로 대체하기에는 너무나도 복잡하기 때문일 것이다. 이 장에서는 WIC를 통해서 이미지를 읽어들여 Direct2D에서 비트맵을 그리는 방법에 대해서 알아보고, DirectWrite API를 이용하여 글꼴 데이터를 읽어온 후에 문자열을 그리는 방법에 대해서도 알아볼 것이다.

비트맵 그리기

WIC는 Windows Imaging Component의 약자로. 이미지와 관련된 기능들을 가진 API이다. Direct2D는 별도의 이미지 로딩 기능을 갖고 있지 않다. 그러나 WIC 객체로부터 비트맵을 생성하는 기능을 제공하기 때문에 WIC를 통해서 WIC 이미지 객체를 만든 후, Direct2D 비트맵으로 변환한 뒤 그려내면 화면에 비트맵을 그려낼 수 있다.

Direct2D에서 비트맵은 ID2D1Bitmap 클래스로 나타내며, 렌더타겟의 CreateBitmap() 메서드나 CreateBitmap***() 메서드로 생성할 수 있다. WIC를 이용해서 파일에서 이미지를 읽어온 후 비트

---

[2] 자세한 건 MSDN의 D2D1_TEXT_ANTIALIAS_MODE 열거형의 값을 찾아보기 바란다.
http://msdn.microsoft.com/en-us/library/windows/desktop/dd368170(v=vs.85).aspx

맵을 생성하는 LoadBitmap() 함수를 d2dutils에 추가해 보자.

D2dutils.h 에 추가된 코드

```
#include <wincodec.h>
#pragma comment(lib, "windowscodecs.lib")
```

D2dutils.cpp 에 추가된 코드

```
HRESULT d2d::LoadBitmap (
	const wchar_t* filename,
	IWICImagingFactory* wic,
	ID2D1RenderTarget* rt,
	ID2D1Bitmap **bitmap )
{
	HRESULT hr = S_OK;
	IWICBitmapFrameDecode *frame = nullptr;
	IWICBitmapDecoder *decoder = nullptr;
	IWICFormatConverter *conv = nullptr;

	// 디코더 생성
	hr = wic->CreateDecoderFromFilename (
			filename,
			0,
			GENERIC_READ,
			WICDecodeMetadataCacheOnDemand,
			&decoder );

	// 프레임 얻기
	if(SUCCEEDED(hr))
		hr = decoder->GetFrame (0, &frame);

	// 변환기 생성
	if(SUCCEEDED(hr))
		hr = wic->CreateFormatConverter(&conv);

	// 변환기 초기화
	if(SUCCEEDED(hr))
		hr = conv->Initialize (
				frame,
				GUID_WICPixelFormat32bppPBGRA,
				WICBitmapDitherTypeNone,
				0,
				1.0f,
				WICBitmapPaletteTypeCustom );
```

```
	// 비트맵 생성
	if(SUCCEEDED(hr))
		hr = rt->CreateBitmapFromWicBitmap (
				conv,
				bitmap );

	// 해제
	d2d::SafeRelease(frame);
	d2d::SafeRelease(decoder);
	d2d::SafeRelease(conv);

	// 결과 반환
	return hr;
}
```

LoadBitmap 은 Windows.h 안에서 매크로로 지정되어 있기 때문에 유니코드 시스템으로 설정된 경우 LoadBitmapW로 바뀐다는 점을 조심하자. 가급적 실제로는 다른 함수이름을 사용하기를 권장한다.

첫 번째 인자는 읽어올 이미지 파일의 이름이다. 두 번째 인자인 WIC 팩토리는 이미지 로딩에 필요한 WIC 객체들을 생성해 준다. 세 번째 인자인 렌더타겟은 WIC 객체로부터 비트맵을 생성해 주며, 네 번째 인자인 비트맵 객체의 더블 포인터로 전달하게 된다. 성공하면 S_OK가 반환될 것이고, 중간에 실패했다면 실패 에러 코드가 반환될 것이다.

먼저 WIC 해독기(IWICBitmapDecoder)을 생성한다. 해독기는 이미지를 로딩해주는 객체이다. CreateDecoderFromFilename() 메서드를 통해 생성하며, 첫 번째 인자는 파일 이름이고, 두 번째 인자는 이미지 형식의 GUID 0(NULL)을 주면 알아서 적당한 이미지 포멧을 선택한다. 이후 파일을 여는 옵션은 GENERIC_READ(읽기 전용)로 주고, 4번째 인자는 디코딩 옵션으로, WICDecodeMetadataCacheOnDemand라는 값을 주었다. 5번째 인자로 해독기의 포인터를 주어서 생성된 객체를 받았다.

하나의 이미지라 할지라도 여러 개의 이미지로 구성되어 있을 수 있다(GIF 처럼). 그 각각의 이미지를 프레임(Frame)이라고 한다. 일반적으로 JPG나 PNG 파일은 하나의 프레임으로 구성되어 있으므로 0번 프레임을 가져온다. decoder->GetFrame(0, &frame). 그러나 이미지 파일마다 형식이 다를 수 있다. 예를 들어 JPEG 형식은 R8G8B8인데 PNG파일은 R8G8B8A8일 수가 있다. 이들을 Direct2D 비트맵에 호환되는 형식으로 변환해주어야 한다. 그 역할을 형식 변환기(Format Converter)가 하게 된다. 이후의 코드는 포멧 변환기를 생성한 후에 변환하고, 렌더타겟의 CreateBitmapFromWicBitmap() 메서드로 WIC 비트맵 객체를 Direct2D 비트맵 객체로 만들면 된

다.

그렇게 만든 비트맵 객체를 한번 그려보도록 하겠다.

BitmapDrawing.cpp

```
IWICImagingFactory *wic = 0;
ID2D1Bitmap *bitmapFood = 0,
                    *bitmapHeegyu = 0,
                    *bitmapMinuk = 0;

void OnDraw()
{
	rt->BeginDraw();
	rt->Clear(D2D1::ColorF(1, 1, 1, 1));

	rt->DrawBitmap(
		bitmapFood,
		D2D1::RectF(0, 0, 800, 300)
		);

	rt->DrawBitmap(
		bitmapHeegyu,
		D2D1::RectF(600, 300, 800, 600),
		1,
		D2D1_BITMAP_INTERPOLATION_MODE_LINEAR
		);

	rt->DrawBitmap(
		bitmapMinuk,
		D2D1::RectF(0, 300, 300, 600),
		1,
		D2D1_BITMAP_INTERPOLATION_MODE_LINEAR
		);

	rt->DrawBitmap(
		bitmapMinuk,
		D2D1::RectF(300, 300, 600, 600),
		1,
		D2D1_BITMAP_INTERPOLATION_MODE_NEAREST_NEIGHBOR
		);


	rt->EndDraw();
}
```

```
bool Setup()
{
	CoInitialize(0);

	CoCreateInstance (
		CLSID_WICImagingFactory,
		0,
		CLSCTX_INPROC_SERVER,
		__uuidof(IWICImagingFactory),
		(void**)&wic);

	d2d::LoadBitmap (
		L"food.jpg",
		wic,
		rt,
		&bitmapFood );

	d2d::LoadBitmap (
		L"heegyu.jpg",
		wic,
		rt,
		&bitmapHeegyu );

	d2d::LoadBitmap (
		L"minuk.png",
		wic,
		rt,
		&bitmapMinuk );

	return true;
}

void CleanUp()
{
	d2d::SafeRelease(bitmapFood);
	d2d::SafeRelease(bitmapHeegyu);
	d2d::SafeRelease(bitmapMinuk);
	d2d::SafeRelease(wic);
	d2d::SafeRelease(rt);
	d2d::SafeRelease(factory2d);
	CoUninitialize();
}
```

OnDraw() 함수에서는 DrawImage 메서드로 이미지들을 그린다. 이미지는 총 3개(bitmapFood, bitmapMinuk, bitmapHeegyu)를 읽는다. DrawImage 메서드의 프로토타입은 아래와 같다.

```
void DrawBitmap(
     ID2D1Bitmap *bitmap,
     CONST D2D1_RECT_F &destinationRectangle,
     FLOAT opacity = 1.0f,
     D2D1_BITMAP_INTERPOLATION_MODE interpolationMode
       = D2D1_BITMAP_INTERPOLATION_MODE_LINEAR,
     CONST D2D1_RECT_F *sourceRectangle = NULL

void DrawBitmap(
     ID2D1Bitmap *bitmap,
     CONST D2D1_RECT_F *destinationRectangle,
     FLOAT opacity = 1.0f,
     D2D1_BITMAP_INTERPOLATION_MODE interpolationMode
       = D2D1_BITMAP_INTERPOLATION_MODE_LINEAR,
     CONST D2D1_RECT_F *sourceRectangle = NULL
```

첫 번째 인자는 그릴 비트맵이고, 두 번째 인자는 그려질 목적지(Destination)을 의미하고, 세 번째 인자는 이미지의 불투명도를 지정한다. 네 번째 인자는 이미지의 보간(Interpolation)을 지정하는 인자이고, 마지막 인자는 이미지의 그려질 부분(원본 영역)을 지정한다. 이 인자는 기본적으로 NULL인데 이러면 전체 이미지가 그려지게 된다.

Setup() 함수에서는 IWICImagingFactory 객체를 생성하고 이미지를 로딩한다, 객체를 생성할 때 CoCreateInstance 함수를 사용했고, CoCreateInstance() 함수를 사용하기 위해 CoInitialize(0) 함수를 호출해서 COM 초기화를 진행했다. CoInitialize를 호출했기 때문에 Cleanup() 함수에서 CoUninitialize() 함수로 생성한 리소스들을 해제하게 하였다. 이후 구현한 LoadBitmap 함수로 이미지를 로드했고, Cleanup() 함수는 모든 리소스를 해제했다. 실행 창은 아래와 같다.

그림 10. BitmapDrawing.cpp의 실행 창

보간

보간이란, 이미지가 원래보다 크거나 작게 그려질 그려내는 방식을 의미한다. Direct2D에서는 2가지 보간 방식을 지원한다.

- D2D1_BITMAP_INTERPOLATION_MODE_NEAREST_NEIGHBOR, 최단입점 보간
  픽셀 중 가장 가까운 픽셀의 값을 사용한다.

- D2D1_BITMAP_INTERPOLATION_MODE_LINEAR, 선형 보간
  픽셀의 인접 비율을 계산해서 비율만큼 색상들을 섞는다.

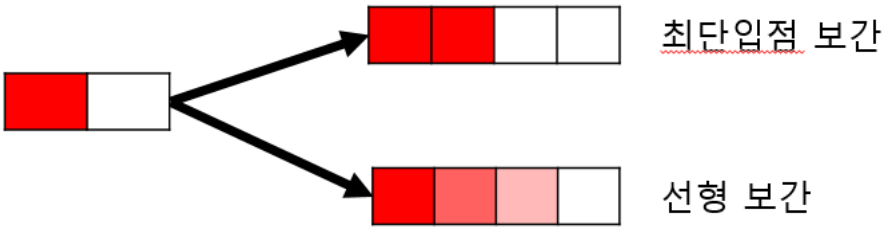

그림 11. 최단입점 보간과 선형보간

그림 11는 가로 2짜리 이미지를 가로 4으로 늘려서 그렸을 때를 보인 것이다. 최단입점 보간에서는 빨간색과 하얀색의 경계가 뚜렷하게 유지되지만 선형 보간에서는 색이 서서히 변화하며 경계 부분이 흐릿해 진다. 위 그림만 보면은 조금 이상할 수도 있지만 그림 12를 자세히 보면 눈치챌 수 있을 것이다. 왼쪽의 선형 보간된 이미지가 오른쪽의 최단입점 보간된 이미지보다 더 부드럽다, 그러나 더 흐릿하기도 하다.

그림 12. 선형 보간(위)와 최단입점 보간(아래)

오늘날의 그래픽카드들은 위 2개의 보간 외에도 삼중선형 보간(Trilinear) 이방성(Anisotropic) 등 더 다양한 고급 보간들을 제공하여 더욱 부드러운 이미지를 그려낼 수 있도록 도와준다. 하지만 보간된 이미지의 품질이 좋을 수록, 성능은 더 느리다는 점을 반드시 염두 해 두어야 한다.

변환하기

기존의 DrawImage()나 DrawGeometry() 같은 메서들로는 그저 도형을 지정된 위치에 그리는 역할만을 수행한다. 변환(Transformation)이란, 쉽게 말하면 이러한 그리기 작업에 수학적인 연산을 통해서 변화를 주는 것을 말한다. 이를 자세히 알기 위해선 행렬(Matrix)에 대해서 알아야 하

지만, 그에 대한 내용은 나중으로 미루고, Direct2D가 제공하는 행렬을 이용해서 변환을 수행하는 기능을 사용해 볼 것이다.

Direct2D에서는 3행 2열의 행렬을 사용해서 변환을 수행한다. 그 데이터를 담는 구조체가 바로 D2D1_MATRIX_3X2_F이다. 그러나 이 구조체는 단순히 6개의 float값만을 가질 뿐, 그 이상의 기능을 제공하지 않아 직접 우리가 값을 채워 넣어야 한다. 그렇기에 D2D1Helper.h 에서는 이 기능을 제공하기 위해 D2D1::Matrix3x2F 클래스를 제공한다.

이 클래스의 메서드들로 알맞은 값을 채운 뒤, ID2D1RenderTarget 클래스의 SetTransform() 메서드로 행렬을 주면 변환이 적용된다, Matrix3x2F클래스가 제공하는 변환 기능으로는 회전(Rotation), 평행이동(Translation), 척도변환(Scale), 기울이기(Skew)가 있다. 각각 기능에의 이름과 알맞은 정적 메서드가 클래스 안에 존재한다.

회전

```
static Matrix3x2F Matrix3x2F::Rotation (
            float angle,
            D2D1_POINT_2F centerPoint = D2D1::Point2F()
)
```

회전은 특정 점을 중심으로 회전시키는 변환이다. Angle은 회전할 각을 의미하며, centerPoint는 회전의 중심점을 의미한다, centerPoint의 기본 값은 원점인 (0, 0)이다. 아래 그림은 원점을 중심으로 약 45도 회전한 빨간 사각형과, 다른 점을 기준으로 약 60도 회전한 초록 사각형이다.

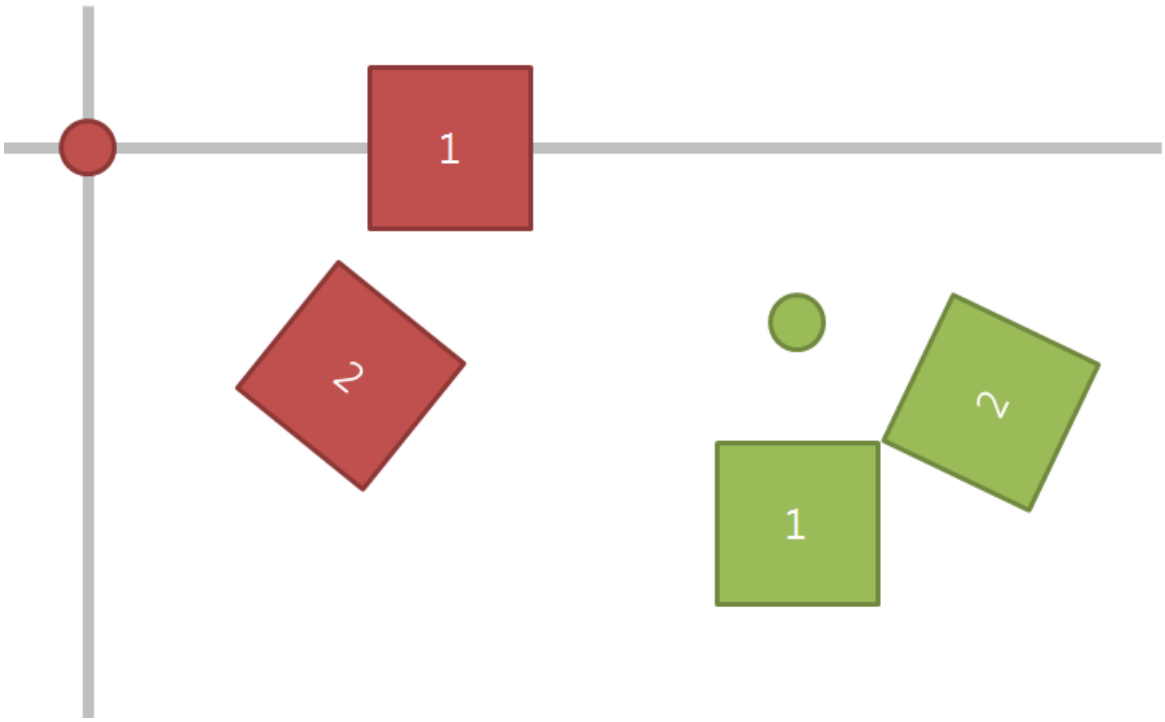

그림 13. 중심점에 따른 회전의 차이

평행이동

```
static Matrix3x2F Matrix3x2F::Translation(
        float x,
        float y
);
```

평행이동은 단순히 x, y 만큼 좌표를 이동시키는 변환이다.

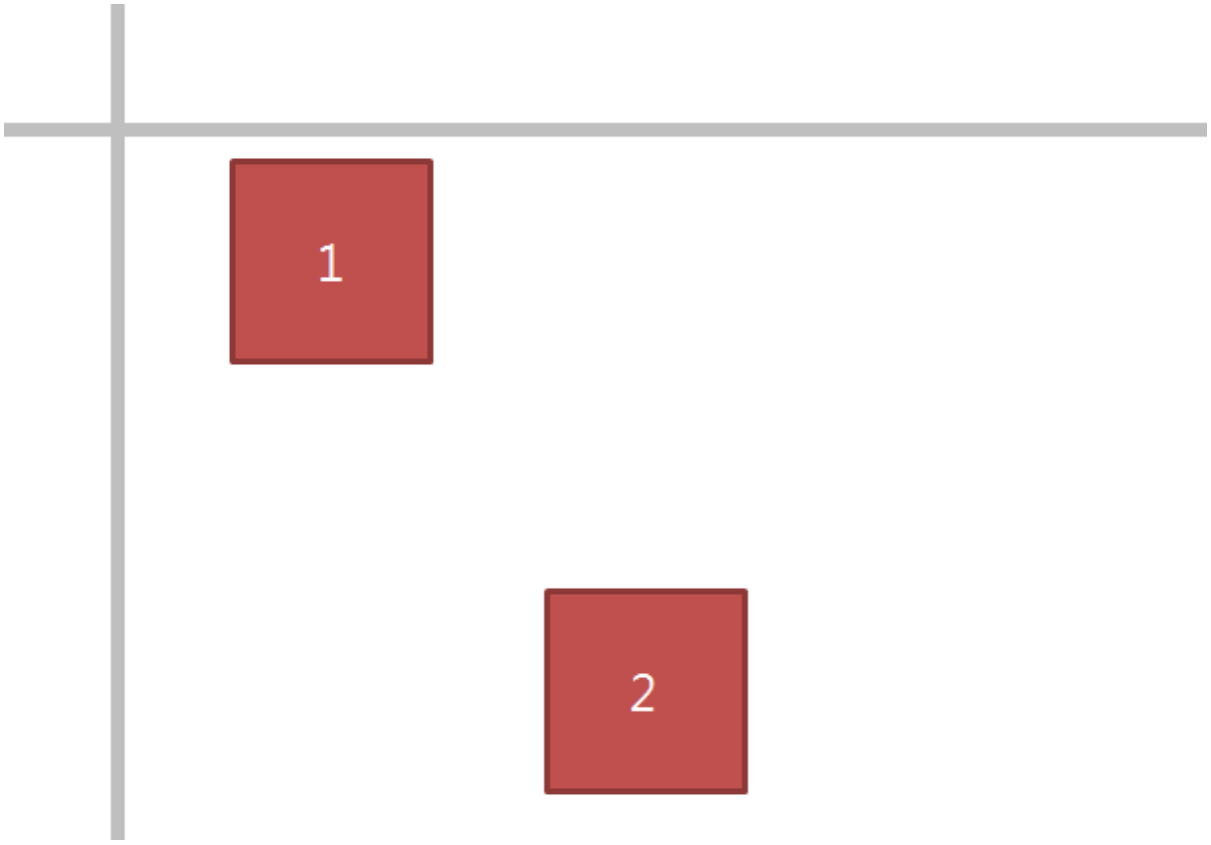

그림 14. X축으로 100, Y축으로 150가량 평행이동한 사각형

척도변환

```
static Matrix3x2F Matrix3x2F::Scale (
         float sx,
         float sy,
         D2D1_POINT_2F centerPoint = D2D1::Point2F()
)
static Matrix3x2F Matrix3x2F::Scale (
         float sx,
         float sy,
         D2D1_POINT_2F *centerPoint
)
```

척도 변환 역시 특정 점을 기준으로 확대(혹은 축소)하는 변환이다. X축으로는 sx만큼, Y축으로는 sy만큼 비례하게 된다. 단순히 크기만 늘어나는 것 뿐만 아니라, 위치 역시 바뀔 수 있다. 단순

히 크기만을 커지게 하고 싶다면, 중심점을 도형의 안으로 지정해야 한다.

아래 그림에서, 빨간 사각형은 중심점을 기준으로 확대하여 위치와 크기가 같이 바뀌었으나, 초록 사각형은 사각형의 내부 점을 중심으로 확대하여 중심점에서 벗어나지 않는다.

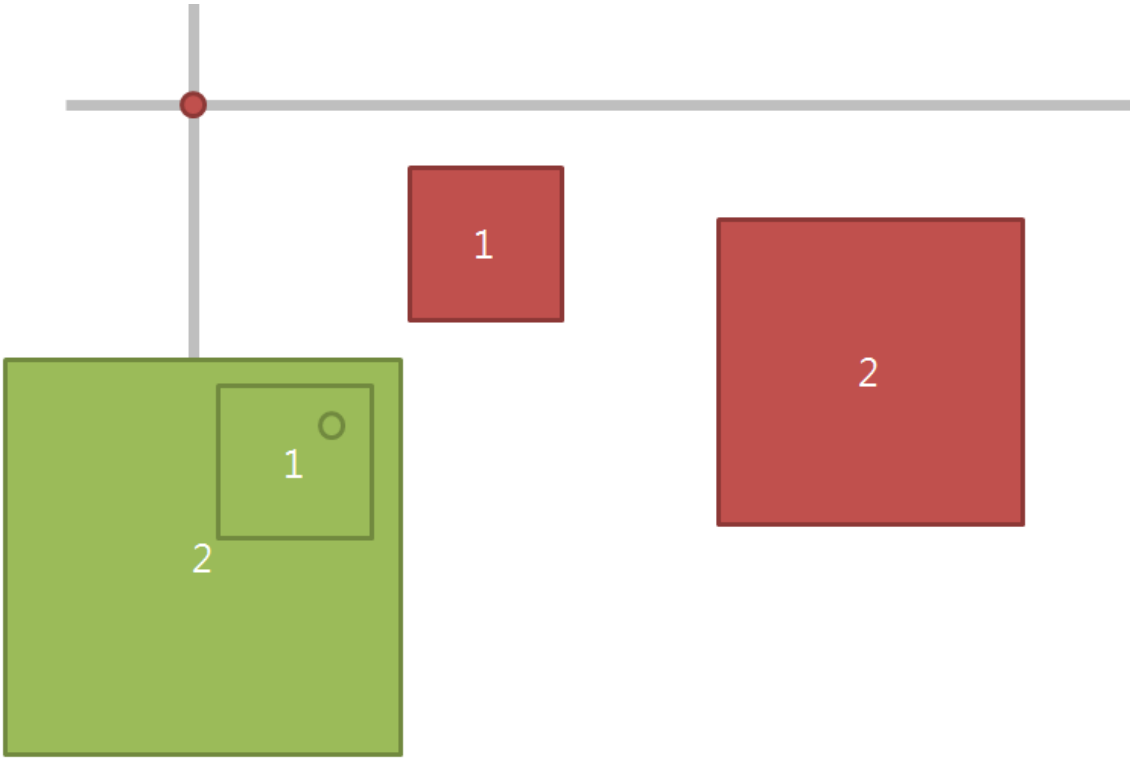

그림 15. 중심점의 위치에 따른 척도변환의 차이

항등 행렬(Identity Matrix)

```
static Matrix3x2F Matrix3x2F::Identity();
```

항등행렬은 아무런 변화도 주지 않는 행렬이다. 만약 그릴 때 변환이 필요하지 않다면 렌더타겟에 항등 행렬을 변환 행렬로 지정해야 한다.

아래 예와 같이 * 연산자를 통해서 여러 개의 변환을 혼합할 수 있다. 예컨데 45도 회전한 뒤 평면좌표에서 (150, 100)만큼 이동하는 변환 행렬을 만들고 싶다면 아래와 같이 하면 된다.

```
Matrix3x2F matrix = D2D1::Matrix3x2F::Rotation(45)
       * D2D1::Matrix3x2F::Translation(150, 100);
```

위 예제는 아래와 같은 결과를 이끌어낸다. Direct2D에서는, 곱의 왼쪽에 있는 변환이 먼저 적용된다. 이러한 것을 전위곱(Pre-multiplication)이라고 한다, 더 자세한 내용은 이후의 장에서 다룬다.

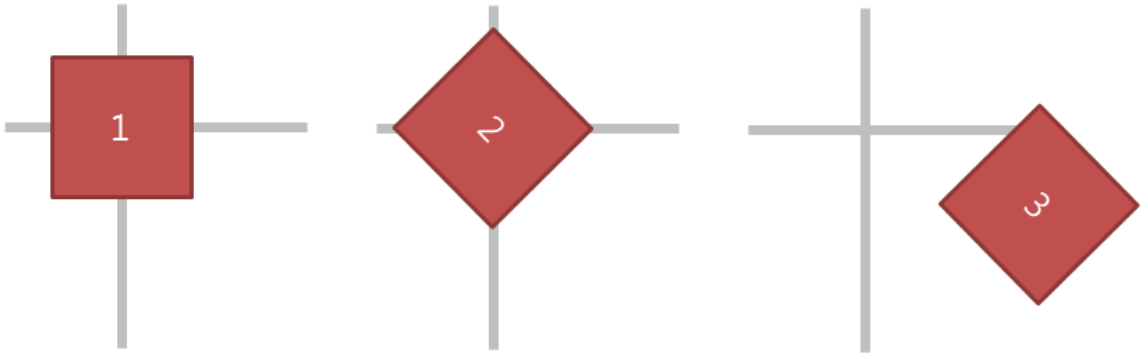

그림 16. 여러 개의 변환이 섞임(회전 → 평행이동)

아래 예제는 다양한 도형들을 SetTransform() 메서드를 통해서 적용한 뒤 그린다. OnDraw() 함수 내에서 `rt->SetTransform(D2D1::Matrix3x2F::Identity());` 이 부분을 보시길!

```
HWND hwnd;
ID2D1Factory *factory2d = 0;          // 팩토리
ID2D1HwndRenderTarget *rt = 0;        // 윈도우 렌더타겟

// 단색 브러시
ID2D1SolidColorBrush *solidBrush = 0;

// 도형 객체 3개
ID2D1Geometry *geom[3];

// 색상 3개
D2D1::ColorF color[] = {
        D2D1::ColorF(D2D1::ColorF::IndianRed),
        D2D1::ColorF(D2D1::ColorF::YellowGreen),
        D2D1::ColorF(D2D1::ColorF::Firebrick)
};

// 변환행렬 3개
D2D1::Matrix3x2F matrices[3];

// 화면을 그릴 때 호출되는 함수
void OnDraw()
{
        rt->BeginDraw();
        // 하얀색으로 지우기
        rt->Clear(D2D1::ColorF(1, 1, 1, 1));

        // 변환 행렬 초기화
        rt->SetTransform(D2D1::Matrix3x2F::Identity());

        // 원래 상태로 그리기
```

```
        for(int i = 0; i < 3; ++i)
        {
                // 브러시 색상 변경
                solidBrush->SetColor(color[i]);
                rt->DrawGeometry(geom[i], solidBrush);
        }

        for(int i = 0; i < 3; ++i)
        {
                // 브러시 색상 변경
                solidBrush->SetColor(color[i]);

                // 변환 지정
                rt->SetTransform(matrices[i]);

                // 도형 그리기!
                // 굵기는 4.5
                rt->DrawGeometry(geom[i], solidBrush, 4.5f);
        }

        rt->EndDraw();
}

// 리소스들을 생성하는 함수
bool Setup()
{
        // 브러시를 생성한다.
        rt->CreateSolidColorBrush (D2D1::ColorF(0, 0, 1), &solidBrush);

        // 도형 객체를 생성한다.
        ID2D1RectangleGeometry *rect;
        factory2d->CreateRectangleGeometry (
                D2D1::RectF(30, 30, 500, 150),
                &rect );

        ID2D1RoundedRectangleGeometry *roundedRect;
        factory2d->CreateRoundedRectangleGeometry (
                D2D1::RoundedRect (
                        D2D1::RectF(100, 100, 200, 200),
                        20,
                        20 ),
                &roundedRect );

        ID2D1EllipseGeometry *ellipse;
        factory2d->CreateEllipseGeometry (
                D2D1::Ellipse (
                        D2D1::Point2F(300, 300),
                        100, 50),
```

```
                &ellipse);

        // 배열에 저장
        geom[0] = rect;
        geom[1] = roundedRect;
        geom[2] = ellipse;

        // 행렬을 만들어서 배열에 저장
        matrices[0] = D2D1::Matrix3x2F::Rotation(45);
        matrices[1] = D2D1::Matrix3x2F::Rotation(45)
                * D2D1::Matrix3x2F::Translation(150, 100);
        matrices[2] = D2D1::Matrix3x2F::Scale(1.5f, 1.5f);

        return true;
}

// 리소스들을 제거하는 함수
void CleanUp()
{
        // 도형들 제거하기
        for(int i = 0; i < 3; ++i)
                d2d::SafeRelease(geom[i]);

        d2d::SafeRelease(solidBrush);
        d2d::SafeRelease(rt);
        d2d::SafeRelease(factory2d);
}
```

예제 2. 변환하기

글씨 그리기

DirectWrite 는 Windows 7 부터 추가된 DirectX 의 글꼴(Font) API 이다. DirectWrite 는 다양하고 훌륭한 기능들을 내장하고 있어서 개발자가 활용하기에 매우 편리하다. 특히나 Direct2D 와의 연동 역시 잘 되어 있기 때문에 손쉽게 Direct2D 앱에 적용할 수 있다. 필자가 후배들에게 OpenGL 과 DirectX 에 대한 질문들을 받았을 때에도 항상 말했던 것은 OpenGL 은 글꼴에 대한 지원이 아쉽다는 것이었다. DirectWrite 는 2D 게임을 만드는 데 있어서 더할 나위 없이 훌륭한 기능들을 제공한다. 한번 알아 보자!

DirectWrite API 는 dwrite.h 안에 정의되어 있으며, dwrite.lib 라이브러리를 링크해야 한다. DirectWrite 역시 WIC 나 Direct2D 처럼 팩토리를 생성하고, 팩토리를 이용해서 다른 객체들을 생성하게 된다. DirectWrite 팩토리를 생성하는 함수는 DWriteCreateFactory() 함수이다. 이후 팩토리로 IDWriteTextFormat 객체를 생성한 뒤 렌더타겟의 DrawText() 메서드에 인자로 주면 끝이다. 매우 간단하다. 한번 예제를 보자.

D2dutils.h 추가된 내용

```
#include <dwrite.h>
#pragma comment(lib, "dwrite.lib")
```

TextDrawing.cpp

```
IDWriteFactory *factoryWrite = 0;
IDWriteTextFormat *formatGoongSeo = 0;
ID2D1SolidColorBrush *solidBrush = 0;

// 화면을 그릴 때 호출되는 함수
void OnDraw()
{
	rt->BeginDraw();
	rt->Clear(D2D1::ColorF(1, 1, 1, 1));

	const wchar_t *str = L"Korean Fanstasy\n 미 눅 +喜奎!";
	rt->DrawTextW(str, wcslen(str),
		formatGoongSeo, D2D1::RectF (
			0, 0, width, height
		), solidBrush);

	rt->EndDraw();
}

// 리소스들을 생성하는 함수
bool Setup()
{
	rt->CreateSolidColorBrush (
			D2D1::ColorF(0, 0, 0),
			&solidBrush );

	DWriteCreateFactory (
		DWRITE_FACTORY_TYPE_SHARED,
		__uuidof(IDWriteFactory),
		(IUnknown**)&factoryWrite);


	factoryWrite->CreateTextFormat (
				L"궁서",
				nullptr,
				DWRITE_FONT_WEIGHT_NORMAL,
				DWRITE_FONT_STYLE_NORMAL,
				DWRITE_FONT_STRETCH_NORMAL,
```

```
                                32.0f,
                                L"ko-KR",
                                &formatGoongSeo );

        return true;
}

// 리소스들을 제거하는 함수
void CleanUp()
{
        d2d::SafeRelease(formatGoongSeo);
        d2d::SafeRelease(factoryWrite);
        d2d::SafeRelease(solidBrush);
        d2d::SafeRelease(rt);
        d2d::SafeRelease(factory2d);
        CoUninitialize();
}
```

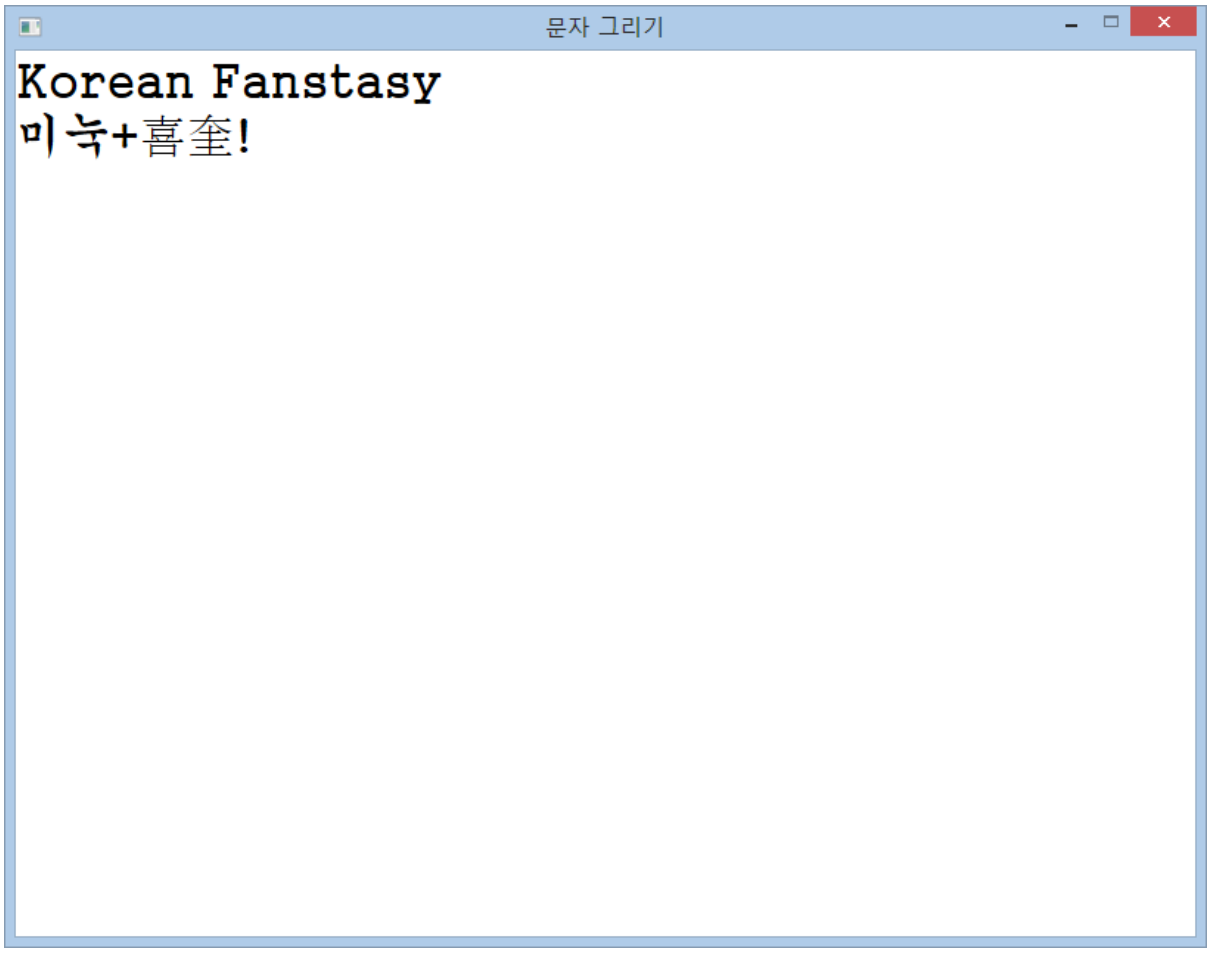

그림 17. TextDrawing.cpp의 결과 화면

DWriteCreateFactory() 함수는 지정된 UUID 의 DirectWrite 팩토리 객체를 생성한다. 첫번째 인자는 생성할 DirectWrite 팩토리의 형식이다. 두 번째 인자는 만들 DirectWrite 팩토리 인터페이스의 UUID 인데, Visual Studio 의 비표준 키워드인 __uuidof 를 이용해서 DirectWrite 객체의 UUID 를 얻어올 수 있다. Setup() 함수에서 DirectWrite 팩토리를 생성하고, 팩토리의 CreateTextFormat() 함수를 통해서 문자 형식을 생성할 수 있다. 이를 이용하여 글자를 그릴 수 있다. 하지만 DirectWrite 팩토리의 기능은 이 외에도 무수히 많기 때문에 DirectWrite 에 잘 안다면 Windows 프로그래밍을 할 때 강력한 도구로 사용할 수 있을 것이다.

```
HRESULT IDWriteFactory::CreateTextFormat (
         const WCHAR * fontFamilyName,
         IDWriteFontCollection * fontCollection,
         DWRITE_FONT_WEIGHT  fontWeight,
         DWRITE_FONT_STYLE  fontStyle,
         DWRITE_FONT_STRETCH  fontStretch,
         FLOAT  fontSize,
         const WCHAR * localeName,
         IDWriteTextFormat ** textFormat
)
```

1. fontFamilyname

   생성할 문자형식의 종류의 패밀리 이름이다. 패밀리 이름이란, 글꼴의 총체적인 이름을 의미한다. 예를 들어 Arial 글꼴은 Arial Black, Arial Bold 등의 다양한 글꼴들을 지니고 있다. 그것들을 합쳐서 Arial 이라는 이름의 글꼴로 만든 것이다. 이 곳에 글꼴의 이름을 넣으면 Windows 에 설치된 글꼴 파일에서 검색하여 객체를 생성하게 된다.

2. fontCollection

   글꼴 패밀리의 세부적인 글꼴들을 지정하는 인자인데, NULL 을 주면 패밀리의 기본 글꼴을 사용하게 된다.

3. fontWeight

   글꼴의 두께(Weight)를 지정한다. 대부분은 무슨 인자를 주던지 별 차이를 느끼기 힘들다.

4. fontStyle

글꼴의 스타일을 지정한다. **볼드체**(DWRITE) 나 *이텔릭체*(Italic)등을 지정할 수 있다.

5. fontStretch

   자간(글자 간격)을 지정하는 인자이다. DWRITE_FONT_STRETCH_* 열거형 값들을 넣으면 된다. 역시나 차이를 느끼기가 힘들 수도 있다.

6. fontSize

   DIP(Device-Independent-Pixels) 단위로 글꼴의 크기를 지정한다. 1DIP 는 1/96 인치를 의미한다.

7. localeName

   로케일은 사용자의 언어를 정의하는 집합을 의미한다. 한 언어에도 여러 개의 로케일이 존재할 수 있다. 아래는 로케일들의 예이다. 한국어는 ko_KR 이다.

국가	로케일
영어	en_US
한국어	ko_KR
프랑스어	fr_FR
중국어 – 간체	zh_CN
중국어 – 번체	zh_TW

8. textFormat

   생성한 문자 형식 객체를 담을 더블 포인터이다.

렌더타겟이 글꼴을 그릴 때는 DrawText 메서드를 사용한다. DrawText 메서드의 프로토타입은 아래와 같다.

```
void DrawText(
         WCHAR *string,
         UINT stringLength,
         IDWriteTextFormat *textFormat,
         const D2D1_RECT_F &layoutRect,
         ID2D1Brush *defaultForegroundBrush,
         D2D1_DRAW_TEXT_OPTIONS options =
D2D1_DRAW_TEXT_OPTIONS_NONE,
```

```
        DWRITE_MEASURING_MODE measuringMode =
DWRITE_MEASURING_MODE_NATURAL
);
```

1. string
   그릴 문자열이다. wchar_t 타입이다.

2. stringLength
   그릴 문자열의 길이다.

3. textFormat
   그릴 때 사용할 문자 형식이다.

4. layoutRect
   그릴 영역을 지정하는 D2D1_RECT_F 구조체이다.

5. defaultForegroundBrush
   그릴 때 칠할 브러시이다. 예제에선 검정색의 단색 브러시를 만들었다.

6. options
   그려지는 문자열들의 크기가 그려질 영역보다 클 때 대처하는 방법을 지정하는 값이다.

   - DWRITE_DRAW_TEXT_OPTIONS_NO_SNAP
     문자가 영역을 넘어서도 자르거나 줄을 바꾸지 않는다.

   - DWRITE_DRAW_TEXT_OPTIONS_CLIP
     넘어선 문자는 잘라내고 그리지 않는다.

   - DWRITE_DRAW_TEXT_OPTIONS_NO_NONE
     기본 값으로, 가로 영역을 넘어서면 자동으로 띄어쓰기를 한다.

7. measuringMode
   글자의 크기를 측정하는 방식을 지정하는 옵션인데, 아직까지 지원되는 값은 DWRITE_MEASURING_MODE_NATURAL 밖에 없다. 이 값은 문자 형식에서 값을 얻어와서 크기를 측정한다. 추후 다른 값들을 추가할 예정인 듯 하다.

문자 레이아웃

DrawText 메서드는 내부적으로 인자로 온 문자열과 문자 형식을 가지고 문자 레이아웃을 만든다. 문자 레이아웃은 문자열이 그려지는 방식이나 정렬에 대한 상태 값들을 갖고 있다. 문자 레이아웃은 IDWriteTextLayout 객체로, DirectWrite 팩토리의 CreateTextLayout 메서드를 통해서 생성할 수 있다. 렌더타겟으로 글씨를 그릴 때, 매번 문자 레이아웃을 생성하는 DrawText 보단,

문자 레이아웃 객체를 만들어 두었다가 DrawTextLayout 메서드로 그려내는 것이 성능에 이점이 있다.

TextLayout.cpp

```
HWND hwnd;
ID2D1Factory *factory2d = 0;            // 팩토리
ID2D1HwndRenderTarget *rt = 0;   // 윈도우 렌더타겟

IDWriteFactory *factoryWrite = 0;
IDWriteTextFormat *formatGoongSeo = 0;  // 문자 형식
IDWriteTextLayout *textLayout = 0;            // 문자 레이아웃
ID2D1SolidColorBrush *solidBrush = 0;

// 화면을 그릴 때 호출되는 함수
void OnDraw()
{
	rt->BeginDraw();
	rt->Clear(D2D1::ColorF(1, 1, 1, 1));

	// 텍스트가 그려지는 경계 영역에 사각형 표시
	rt->DrawRectangle (
		D2D1::RectF(200, 100, 600, 500),
		solidBrush );

	// (200, 100)위치에 문자 레이아웃을 그린다.
	rt->DrawTextLayout(
		D2D1::Point2F(200, 100),
		textLayout,
		solidBrush
		);


	rt->EndDraw();
}

// 문자의 범위를 나타내는 DWRITE_TEXT_RANGE 객체를 생성한다.
DWRITE_TEXT_RANGE MakeRange(int start, int length)
{
	DWRITE_TEXT_RANGE range = {0};
	range.startPosition = start;     // 시작 위치(0 부터)
	range.length = length;                  // 문자 개수
	return range;
}
```

```
// 리소스들을 생성하는 함수
bool Setup()
{
	// 브러시 생성
	rt->CreateSolidColorBrush (
			D2D1::ColorF(0, 0, 0),
			&solidBrush );

	// 팩토리 생성
	DWriteCreateFactory (
		DWRITE_FACTORY_TYPE_SHARED,
		__uuidof(IDWriteFactory),
		(IUnknown**)&factoryWrite);

	// 글꼴 형식 생성
	factoryWrite->CreateTextFormat (
				L"궁서",
				nullptr,
				DWRITE_FONT_WEIGHT_NORMAL,
				DWRITE_FONT_STYLE_NORMAL,
				DWRITE_FONT_STRETCH_NORMAL,
				32.0f,
				L"ko-KR",
				&formatGoongSeo );

	// 그릴 문자열
	const wchar_t *str = L"Korean Fanstasy\n"
				L"미눅미눅+喜奎!\n"
				L"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

	// 텍스트 레이아웃 생성
	factoryWrite->CreateTextLayout (
				str,		// 문자열
				wcslen(str), // 문자열의 길이
				formatGoongSeo,	// 글꼴 형식
				400,		// 최대 가로 크기
				400,		// 최대 세로 크기
				&textLayout);// 생성된 글꼴 레이아웃

	// 단락 정렬 설정(수직 정렬)
	textLayout->SetParagraphAlignment(
DWRITE_PARAGRAPH_ALIGNMENT_CENTER);

	// 문자 정렬 설정(수평 정렬)
	textLayout->SetTextAlignment(DWRITE_TEXT_ALIGNMENT_TRAILING);
```

```
	// 0번째부터 6개의 문자인 'Korean' 을 이탤릭체로
	textLayout->SetFontStyle(
		DWRITE_FONT_STYLE_ITALIC,
		MakeRange(0,6));

	// 앞의 '미눅'을 두껍게
	textLayout->SetFontWeight(
		DWRITE_FONT_WEIGHT_HEAVY,
		MakeRange(16,2));

	// 뒤의 '미눅'을 얇게
	textLayout->SetFontWeight(
		DWRITE_FONT_WEIGHT_THIN,
		MakeRange(18,2));

	// 뒤의 '미눅'에 밑줄
	textLayout->SetUnderline(
		DWRITE_FONT_WEIGHT_THIN,
		MakeRange(18,2));

	return true;
}

// 리소스들을 제거하는 함수
void CleanUp()
{
	d2d::SafeRelease(textLayout);
	d2d::SafeRelease(formatGoongSeo);
	d2d::SafeRelease(factoryWrite);
	d2d::SafeRelease(solidBrush);
	d2d::SafeRelease(rt);
	d2d::SafeRelease(factory2d);
	CoUninitialize();
}
```

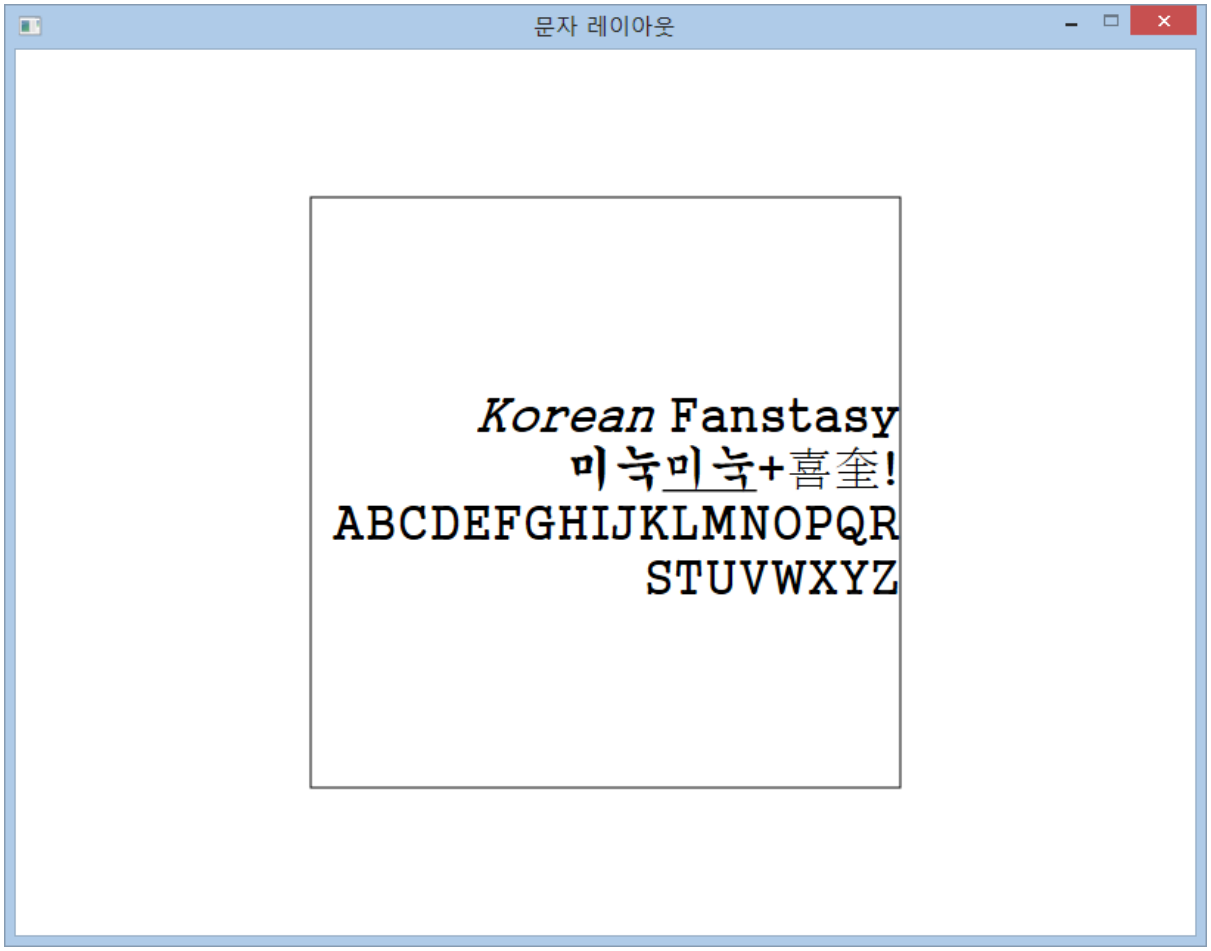

그림 18. TextLayout.cpp

문자 레이아웃의 기능에는 여러가지가 있다, 첫 번째는 문자를 정렬하는 것으로, 수직 수평 정렬을 지정할 수 있다. 수평 정렬은 SetTextAlignment() 메서드를 통해 지정하며, DWRITE_TEXT_ALIGNMENT 열거형의 값을 인자로 준다. 종류로는 아래 3 가지가 있고, 간략하게 썼을 뿐 모두 DWRITE_TEXT_ALIGNMENT_ 로 시작한다.

LEADING	CENTER	TRAILING
안녕	안녕	안녕

수평 정렬에서 주의해야 할 점은 위 표의 예는 왼쪽에서 오른쪽 읽기 상태에서만 나타나기 때문이다. 만약 오른쪽에서 왼쪽 읽기 상태일 경우라면 LEADING 과 TRAILING 은 반대가 된다. 아랍권이나 과거 조선이나 중국에서는 글자를 오른쪽에서 왼쪽으로, 위에서 아래로 쓰지 않았는가? 읽기 방향은 SetReadingDirection() 메서드를 통해 지정할 수 있다. 그러나 이 값은 로케일이나 문자 형식에 영향을 받는 듯 하다. DirectWrite 는 수직 그리기 역시 지원한다. 그러나 그 정도 범주의 고급 기능들은 여기선 다루지 않는다.

수직 정렬은 SetParagraphAlignment() 메서드로 지정하며, 인자는 DWRITE_PARAGRAPH_ALIGNMNET 열거형의 값 중 하나를 준다. 값으로는 아래의 종류들이 있다. 역시나 DWRITE_PARAGRAPH_ 로 시작한다.

NEAR	CENTER	FAR
안녕	안녕	안녕

NEAR 는 그려지는 영역의 윗부분에, CENTER 는 중앙, FAR 는 아랫면에 달라붙어 그려지게 한다.

적중 검사

마지막으로 설명할 DirectWrite 의 멋진 기능은 적중 검사이다. 적중 검사란 문자 레이아웃의 특정 위치에 있는 글자를 알아내거나, 특정 글자가 위치하는 좌표를 알아낸다. 우리가 해 볼 적중 검사 전자로서 문자 레이아웃의 HitTestPoint() 메서드로 할 수 있다. 이전의 예제에서 OnDraw() 함수만 바뀌었다.

HitTest.cpp

```
void OnDraw()
{
	rt->BeginDraw();
	rt->Clear(D2D1::ColorF(1, 1, 1, 1));

	POINT cursor;
	GetCursorPos(&cursor);
	ScreenToClient(hwnd, &cursor);

	DWRITE_HIT_TEST_METRICS metrics = {0};
	BOOL isTrailingHit, isInside;

	// HitTestPoint() 메서드로 적중 검사를 수행한다.
	// 문자 레이아웃을 (200, 100)에 그리기 때문에
	// X좌표에 200을 빼고 Y좌표에 100을 뺀다.
	HRESULT hr = textLayout->HitTestPoint (
			cursor.x - 200,		// X 좌표
			cursor.y - 100,		// Y 좌표
			&isTrailingHit,		// 왼쪽 혹은 오른쪽에 붙어 있는가
			&isInside,		// 영역 안인가?
			&metrics		// 정보들
			);
```

```
	// 기존의 밑줄은 없앤다
	textLayout->SetUnderline(false, range);

	/*
	isTrailingHit는 X좌표가 그려지는 범위 내에 있을 때,
	가장 가까운 줄에 글자가 존재하는지 여부를 의미하며,
	isInside는 점의 위치가 문자 사이에 있는지를 나타내는 값이다
	설명이 어렵다면 아래 조건문을 바꿔 보면서 결과를 확인해
	보길 바란다.
	*/
	// 맞았네
	if(SUCCEEDED(hr) && isTrailingHit && isInside)
	{
		// 적중한 문자의 시작 위치와 길이를 저장하고
		range.startPosition = metrics.textPosition;
		range.length = metrics.length;

		// 적중한 문자에 밑줄을 친다.
		textLayout->SetUnderline(true, range);

		const wchar_t* success = L"성공!";
		rt->DrawText(success, wcslen(success), formatGoongSeo,
			D2D1::RectF(0, 0, 50, 0), solidBrush );
	}

	// (200, 100)위치에 문자 레이아웃을 그린다.
	rt->DrawTextLayout(
		D2D1::Point2F(200, 100),
		textLayout,
		solidBrush
		);


	rt->EndDraw();
}
```

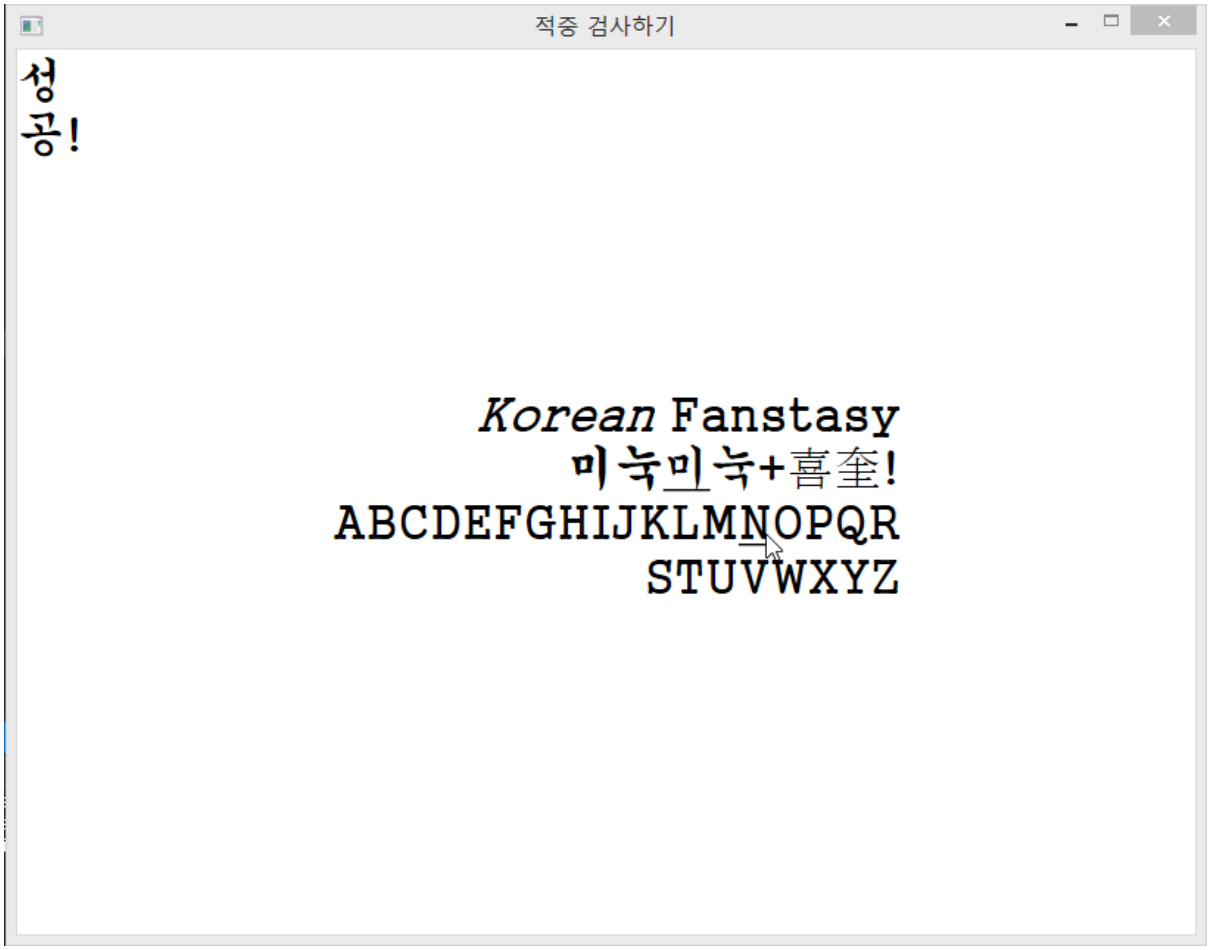

그림 19. HitTest.cpp의 결과 화면

Direct2D, DirectWrite, WIC 에는 수많은 훌륭한 기능들이 존재해서 윈도우 프로그래머들을 도와준다. Windows 는 비스타와 7 을 겪으면서 많은 변화를 거쳤고, 모바일 시장을 겨냥한 Windows 8 의 등장으로 다시 한번 대격변을 겪고 있다. 앞서 설명한 라이브러리들은 새로운 윈도우 운영체제의 핵심 라이브러리가 되어서 더 많은 기능이 추가되고 발전해 나가고 있다. 내가 설명한 내용들이 앞으로 배울 더 많은 기능들의 발판이 되길 바란다.