Microsoft ROBOTICS STUDIO

Microsoft Visual Simulation 사용자 가이드 및 튜토리얼

Microsoft 시뮬레이션 개발환경 가이드 및 튜토리얼은 아래의 항목으로 구성됩니다.

목차

1. 시작하기

Microsoft Visual Simulation 시작하기

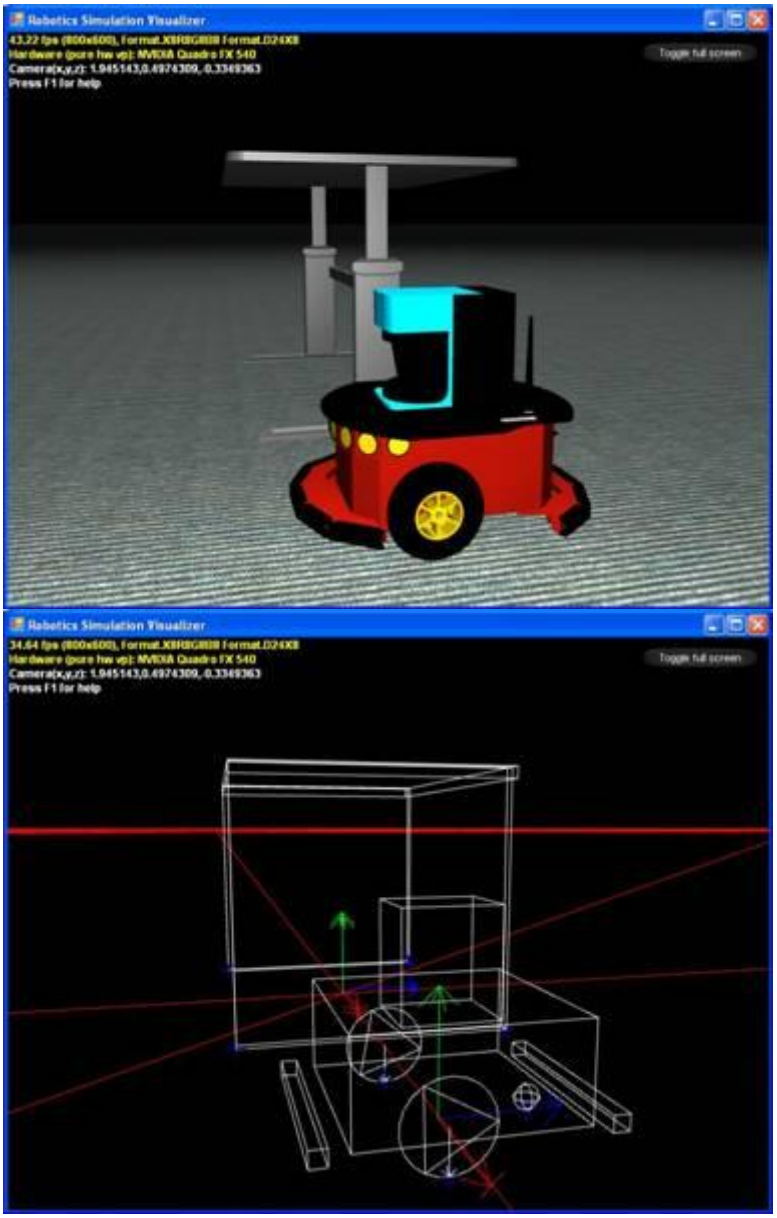
로봇을 개발하는데 있어서 하나의 심각한 문제는 로봇 자체가 고가의 하드웨어이고 대부분 한 대 밖에 없다는 사실입니다. 이러한 문제로 인하여, 개발된 프로그램들에 대해 사전에 충분한 테스트를 수행할 수 없고 많은 기간과 비용이 소요되고 있습니다. MSRS는 이러한 문제를 해결하고 개발자들에게 사전에 충분히 테스트할 수 있고 재현할 수 있는 환경을 제공하기 위해 시뮬레이션 환경을 제공합니다.

MSRS의 시뮬레이터는 물질세계에서의 로봇에 적용되는 중요한 요소들을 모두 시뮬레이션 상황하에서 구현해 낼 수 있습니다. 기본적으로, 공간과 중력, 마찰, 탄성 계수뿐 만이 아니라, 모터와 센서 등의 기본적인 작용 등도 구현이 가능합니다.

MSRS의 시뮬레이션 환경은 Ageia사의 PhysX 엔진과 Microsoft XNA 프레임워크를 기반으로 하며, 하드웨어 로봇과 동일하게 작동되는 소프트웨어 로봇을 시뮬레이터 상에서 개발할 수 있도록 함으로써, 로봇 개발에 있어서 기간과 비용을 대폭 감소시키고, 개발 결과물에 대한 생산성과 코드의 품질을 대폭 향상시킵니다.

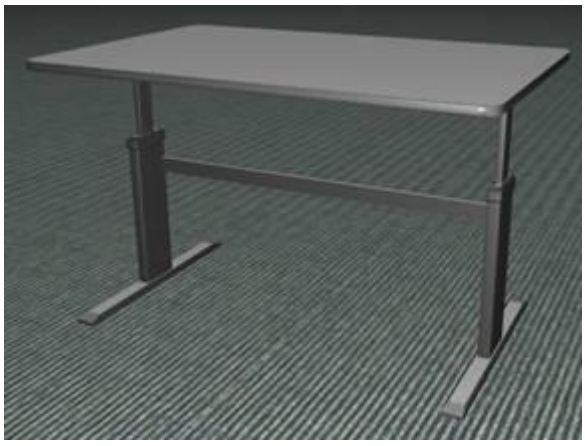
또한 하드웨어 로봇 없이 시뮬레이션 환경 구성만으로 로봇 주행 알고리즘의 연구와 같은 다양한 형태의 전문화된 알고리즘 연구에도 활용될 수 있으므로, 알고리즘 및 지능형 서비스에 특화된 개발이 가능하도록 지원합니다.

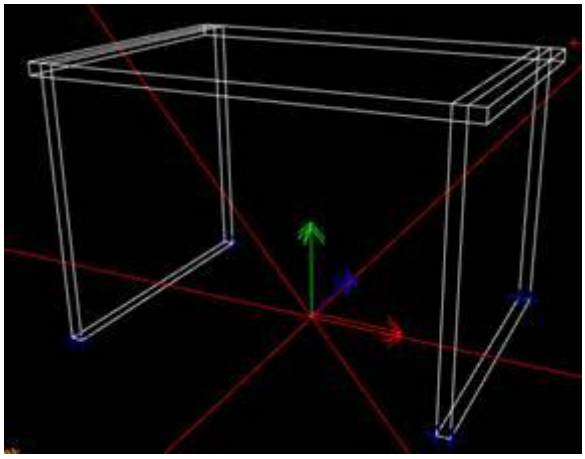
MSRS 시뮬레이션 환경에서는 간단한 물리객체(Entity)의 추가를 통해 로봇에 필요한 모터, 휠, 센서 등의 기능들을 손쉽게 구현해 낼 수 있습니다.



로봇 객체의 구현 화면

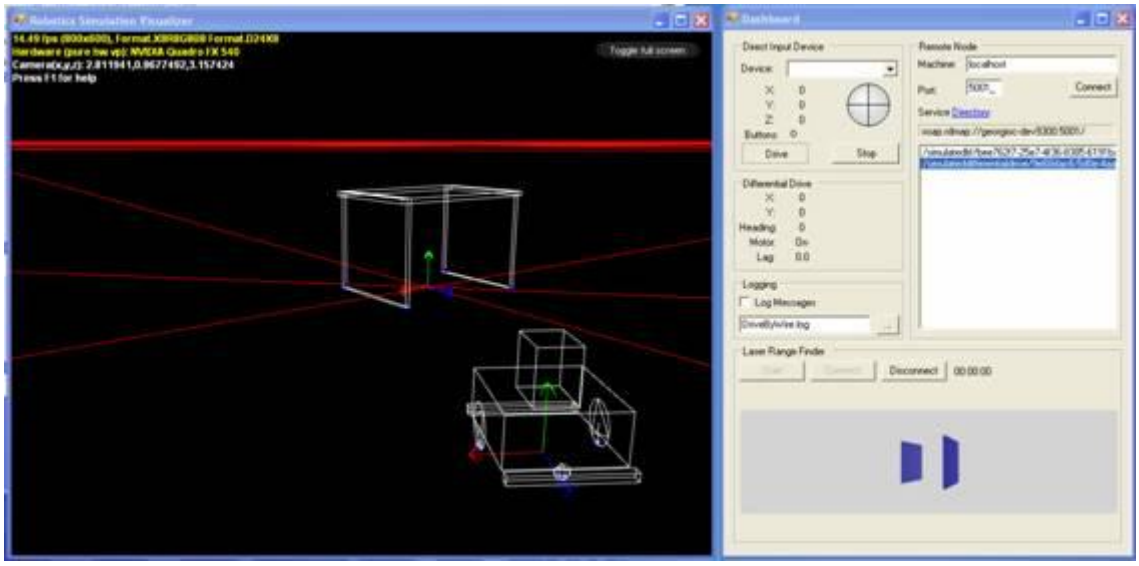
또한 하늘, 산, 테이블, 벽 등의 다양한 사물들과 환경 등을 추가하거나 설정할 수 있습니다.





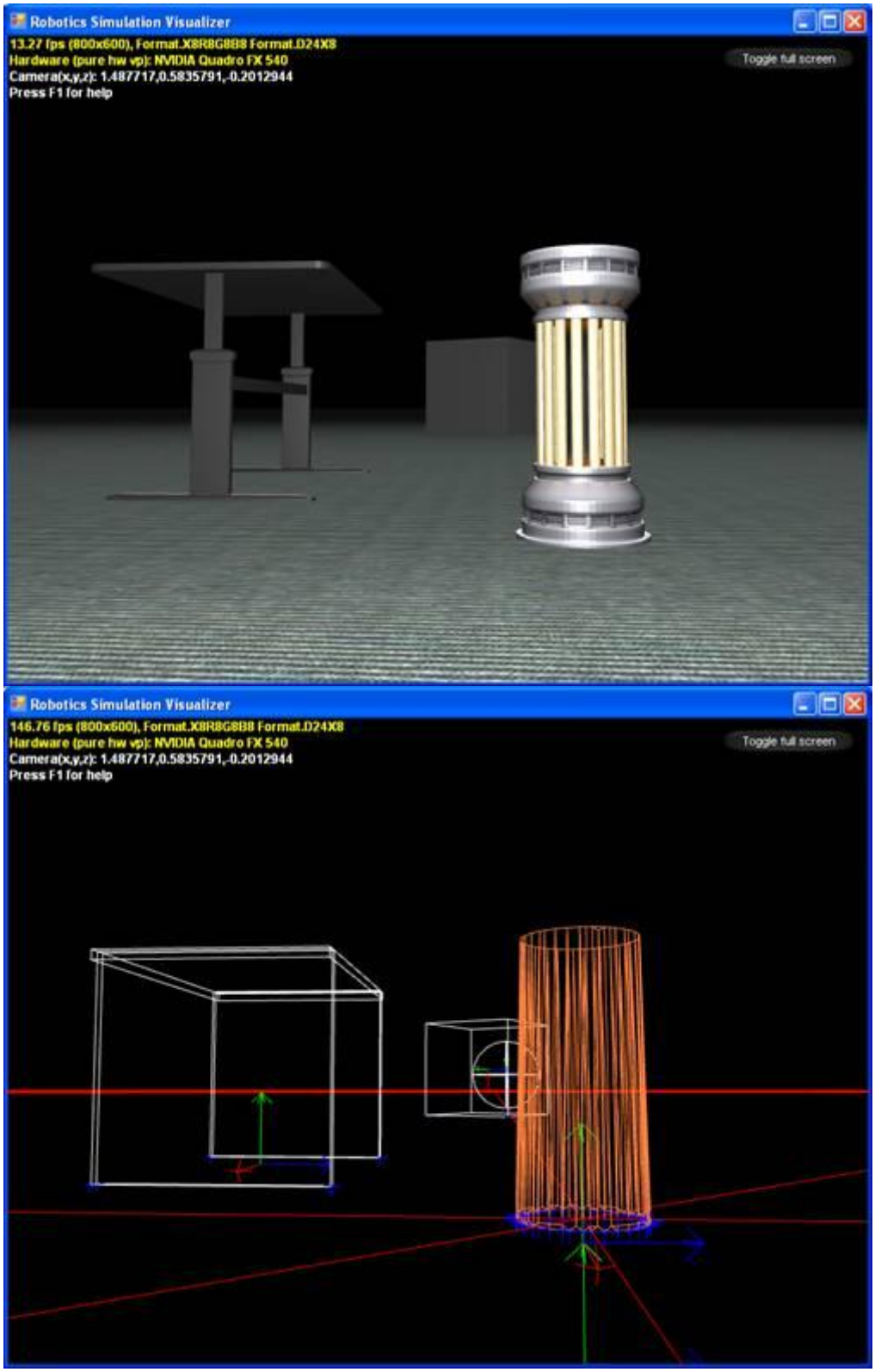
테이블 객체를 추가한 화면

MSRS의 시뮬레이션 환경에서는 기본적으로 제공되는 Simple Dashboard 서비스를 통하여, 시뮬레이터상의 로봇을 제어하거나 레이저 스캐닝 데이터 등을 확인할 수 있습니다.



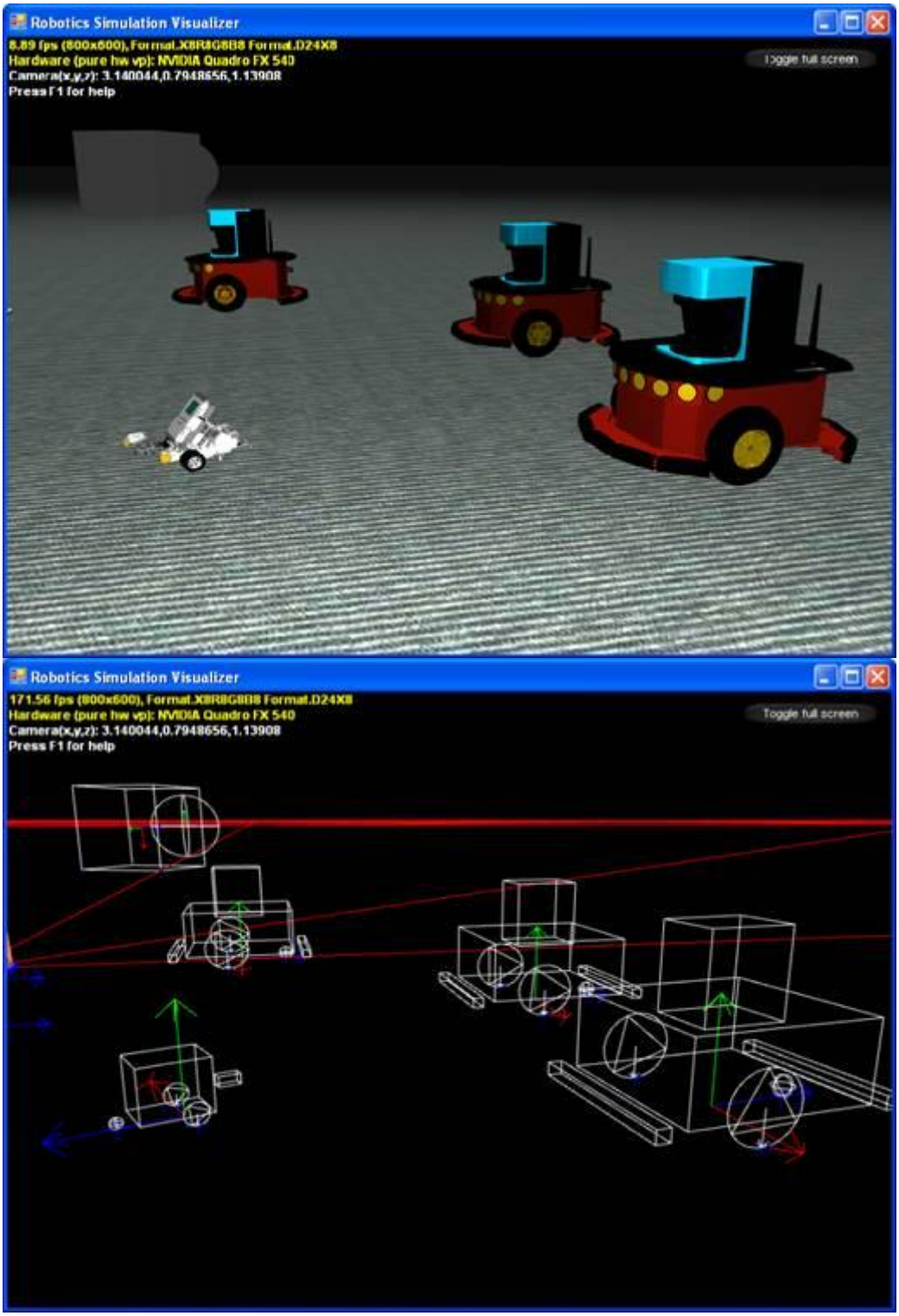
로봇에 연결하여 레이저 스캐닝 센서 데이터를 읽고 있는 화면

MSRS의 시뮬레이션 환경에서는 단순한 형태의 물체뿐 만이 아니라 복잡한 형태의 사물도 설계 데이터를 변환하여 손쉽게 반영할 수 있습니다.



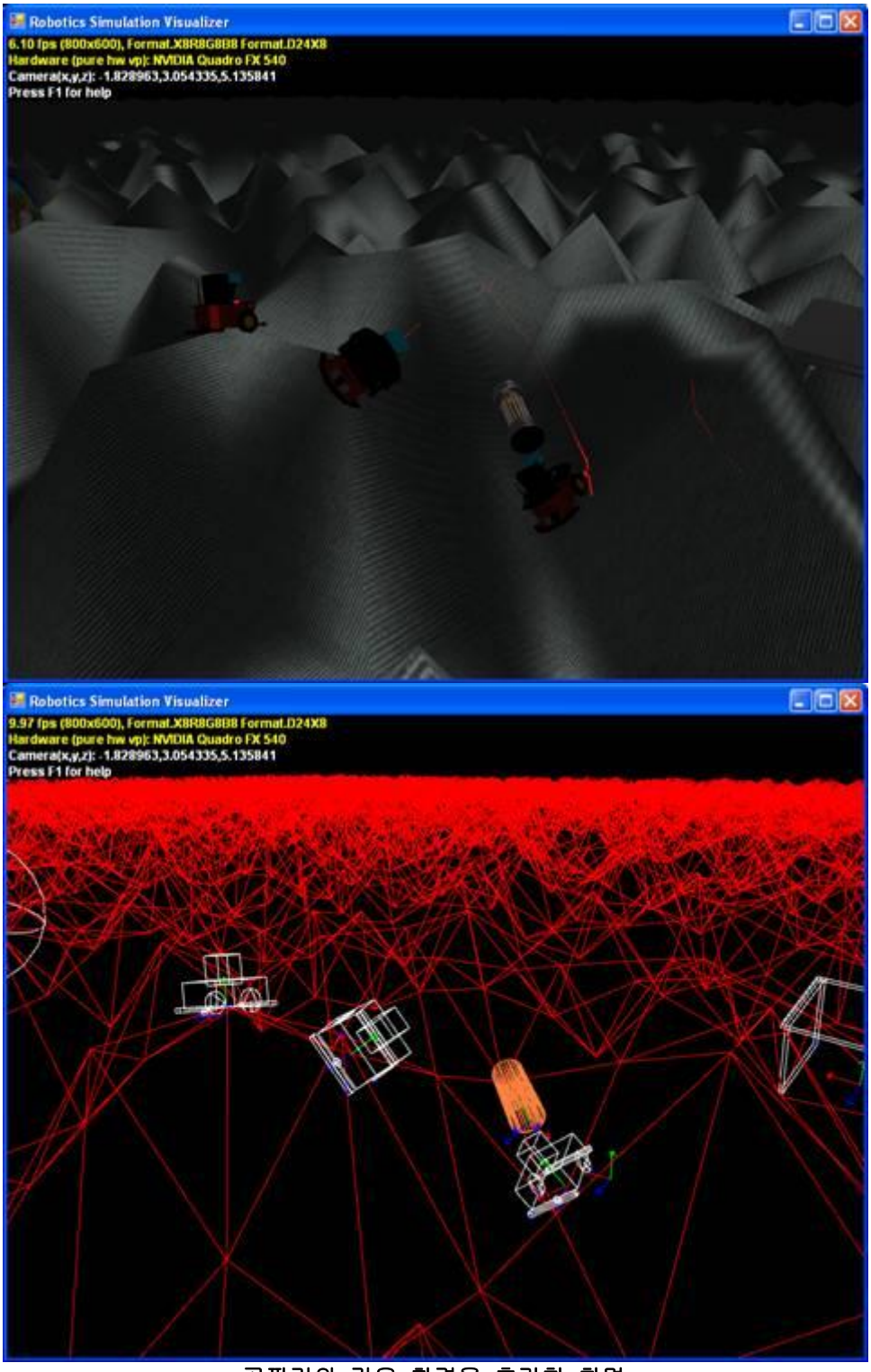
복잡한 사물을 시뮬레이션 환경에서 구현한 화면

MSRS의 시뮬레이션 환경에서는 한 대의 로봇을 시뮬레이션 하는 것뿐 만이 아니라 동시에 여러 대의 로봇을 시뮬레이션 하고 각각의 로봇들을 제어하는 작업들을 손쉽게 진행할 수 있습니다.



4대의 로봇들을 동시에 구현한 화면

MSRS의 시뮬레이션 환경에서는 평지뿐 만이 아니라 산과 같은 다양한 조건의 로봇 이동 환경을 추가할 수 있습니다.



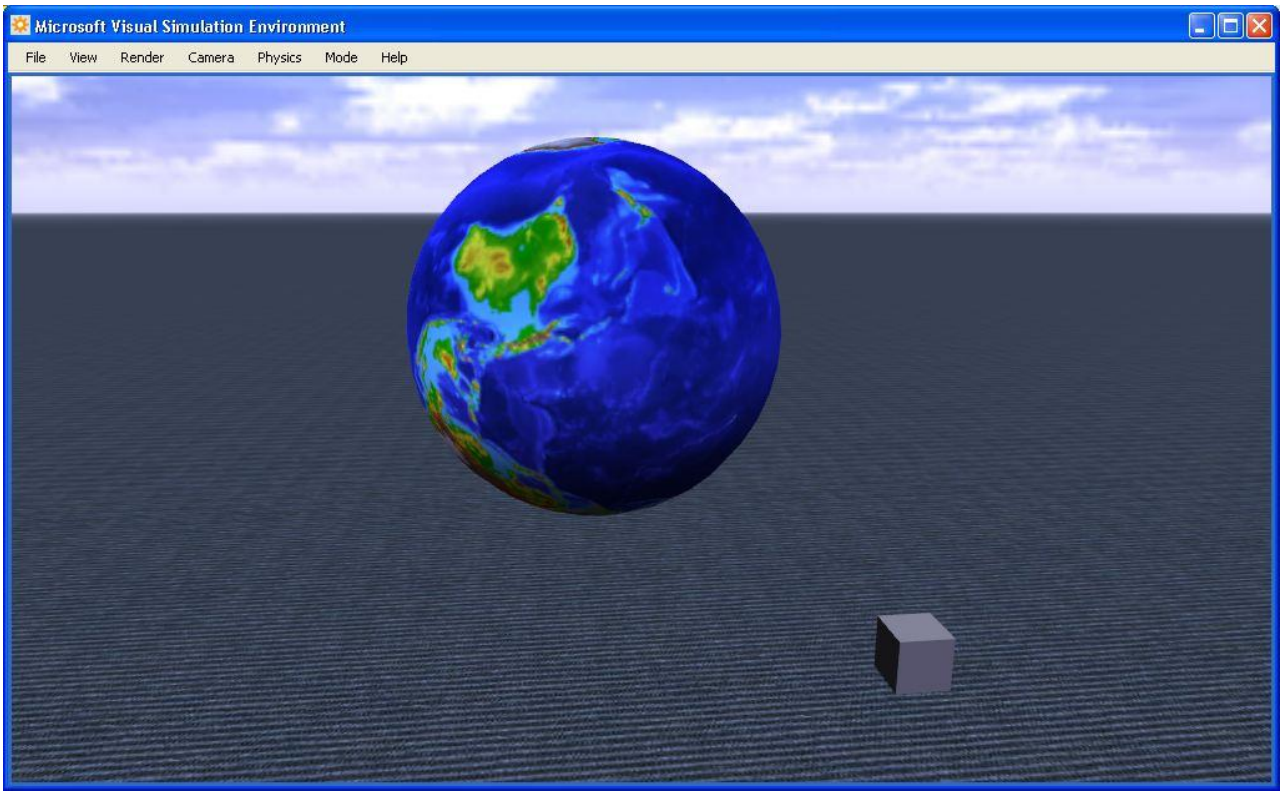
골짜기와 같은 환경을 추가한 화면

2. 사용자 가이드

Microsoft 비주얼 시뮬레이션 환경 시작하기

비주얼 시뮬레이션 환경을 경험해 보기 위해서는 MSRS의 시작 메뉴 중에서 Visual Simulation Environment 메뉴를 클릭한 후 하부 메뉴에 있는 여러 가지의 시뮬레이션 항목 중 하나를 선택하여 실행합니다. MSRS 시작 메뉴에는 시뮬레이션 환경을 처음 접하는 개발자가 시뮬레이션 환경을 이해할 수 있도록 대표적인 시뮬레이션 활용 사례들을 모아 놓았습니다.

시뮬레이션 환경은 MSRS의 하나의 파트너 서비스로서 개발자가 개발한 시뮬레이션 모듈이 실행될 때 같이 실행되며, 일반적인 서비스를 구동하는 것과 같이 메니페스트 파일을 통해 구동이 될 수 있고, 파트너 서비스 등록을 통해 구동이 되기도 합니다.



시뮬레이터가 실행될 때, 왼쪽 마우스 버튼을 클릭한 채로 포인터를 스크린 상에서 움직여 가면서 카메라의 뷰포인트를 변경시킬 수 있습니다. 이러한 변경은 카메라 자체의 포지션을 변경시키지 않고 단순히 카메라가 바라보는 방향만을 변경시킵니다.

카메라를 위치를 변경시키기 위해서는 아래와 같은 키보드 명령을 사용하여 변경시킬 수 있습니다.

w	앞으로 이동
s	뒤로 이동
a	왼쪽으로 이동
d	오른쪽으로 이동
q	위로 이동
e	아래로 이동

Microsoft 비주얼 시뮬레이션 환경 메뉴

비주얼 시뮬레이션 환경에서는 아래의 메뉴들이 제공됩니다.

File 메뉴

Open Scene - scene 파일을 불러옵니다.

Save Scene As - scene 파일을 저장합니다. Scene 파일이 저장될 때, 시뮬레이터는 scene에 있는 모든 엔터티들의 상태값을 저장하며, 해당 엔터티와 연관되어 있는 서비스와 관련 메니페스트 정보도 같이 저장됩니다.

Open Manifest - 서비스 메니페스트 파일을 불러 옵니다..

Capture Image As - 현재의 시뮬레이션 상의 뷰를 이미지 파일로 저장합니다.

Exit - 시뮬레이터를 종료하고 DSS 노드를 종료합니다.

Entity 메뉴

Cut - 현재 체크된 엔터티들을 잘라냅니다.

Copy - 현재 체크된 엔터티들을 복사합니다.

Paste - 잘라내거나 복사한 엔터티들을 scene 안에 추가합니다.

Paste As Child - 잘라내거나 복사한 엔터티들을 현재 체크되어 있는 엔터티의 하위 엔터티로 추가합니다.

New - 새로운 엔터티를 추가하기 위한 다이얼로그 창을 표시합니다.

Load Entities - 파일로부터 엔터티들을 불러옵니다.

Save Entities As - 현재 체크된 엔터티들을 파일에 저장합니다.

View 메뉴

Status bar - 상태바를 숨기거나 표시합니다. 상태바는 1 초단위의 프레임 비율과 카메라 위치 방향 등을 표시합니다.

Look Along - 특정 축을 기반으로 뷰를 설정합니다.

Render 메뉴

첫번째 4개의 항목들은 시뮬레이션에서의 엔터티들을 어떻게 표시할 것인지에 대해 각각 다른 방법을 보여주며, F2 키를 눌러서 각각의 메뉴를 순차적으로 변경시킬 수 있습니다.

Visual - Full 3D 상태로 표시하며, 각각의 엔터티에 연관된 메시들로 표시되고, 빛과 명암이 표시됩니다.

Wireframe - Scene을 wireframe 뷰로 표시합니다. 이 모드에서는 얼마나 많은 폴리곤이 메시를

구성하는 지, 그리고 폴리곤의 면이 어디에 있는 지에 대한 대략적인 아이디어를 보여줍니다.

Physics - Scene을 physics 아웃라인 형태로 보여줍니다. 이 모드에서는 각각의 엔터티가 물리엔진에서 어떻게 표시되는 지를 알 수 있습니다. 만약 물리엔진이 사용 가능하지 않다면 엔터티들이 표시되지 않을 수 있습니다.

Combined - Full 3D 뷰와 Physics 뷰를 결합하여 표시합니다. 이 모드에서는 비주얼 메시와 물리 인터티가 얼마나 잘 매칭되는 지를 확인할 수 있습니다.

Graphics Settings - Scene이 어떻게 랜더링 되는 지에 대한 컨트롤 값을 설정합니다.

Camera 메뉴

카메라 메뉴는 쉽게 개발자가 카메라를 변경하여 보여지는 Scene을 변경하는 기능을 제공합니다. 사용자는 F8키를 눌러 카메라를 순차적으로 변경할 수 있습니다.

Main Camera - 기본적으로 제공되는 카메라로 설정합니다.

Other cameras - 현재의 메니페스트 파일에서 제공되는 카메라들의 목록을 보여주고 해당 카메라를 선택할 수 있도록 합니다.

Physics 메뉴

Enabled - 시뮬레이션에서 물리 속성을 설정하거나 해제할 수 있습니다.

Settings - 중력값을 설정하고, 기본 카메라를 강체로 활용할 수 있도록 설정할 수 있습니다. 만약 카메라 옵션이 설정되었을 경우, 카메라를 움직여서 Scene 안에 있는 물체들에 충돌시킬 수 있습니다.

Mode 메뉴

이 항목 안에 있는 메뉴들은 실행 또는 편집 모드로 구성되며 F5키를 통해 모드가 상호 변환됩니다.

Run - 기본적인 시뮬레이션 실행 모드로 전환합니다.

Edit - 편집 모드로 전환하여 각각의 엔터티들의 상태를 변경할 수 있도록 합니다. 이 모드에서는 물리 속성이 작용하지 않습니다.

Help 메뉴

Contents - 시뮬레이션 환경에 대한 보다 더 세부적인 정보를 제공하기 위한 웹페이지를 표시합니다.

About - 제품의 버전 정보를 표시합니다.

Microsoft 비주얼 시뮬레이션 환경의 단축키 및 마우스

비주얼 시뮬레이션 환경에서는 단축키와 마우스를 통해서 다양한 형태의 명령들을 수행할 수 있습니다.

단축키 사용

시뮬레이터는 단축키에 의해 카메라의 방향을 변경시킬 수 있으며, 이러한 작동은 해당 시뮬레이터 패널이 포커스를 가지고 있을 경우에만 가능합니다. 해당 패널에 포커스가 가 있을 경우에는 테두리 색갈이 파란색으로 변경됩니다.

w	앞으로 이동
s	뒤로 이동
a	왼쪽으로 이동
d	오른쪽으로 이동
q	위로 이동
e	아래로 이동

만약 Shift 키와 같이 위의 키를 누를 경우 20배 빨리 수행됩니다.

각 모드간 변환키는 아래와 같습니다.

F2	랜더 모드 변경
F3	Physics 엔진 모드 변경
F5	편집 모드와 실행 모드 변경
F8	카메라 변경

편집모드에서 특정 엔터티를 선택하였을 경우, 왼쪽 Ctrl키를 누르면, 해당 엔터티가 밝게 표시됩니다. 또한 Ctrl 키를 누른 상태에서는 다음과 같은 키를 같이 사용할 수 있습니다.

Up 방향키	선택된 객체를 해당 객체의 +Y 방향에서 바라봅니다.
Ship+Up 방향키	선택된 객체를 해당 객체의 -Y 방향에서 바라봅니다.
Left 방향키	선택된 객체를 해당 객체의 +X 방향에서 바라봅니다.
Shift+Left 방향키	선택된 객체를 해당 객체의 -X 방향에서 바라봅니다.
Right 방향키	선택된 객체를 해당 객체의 +Z 방향에서 바라봅니다.
Shift+Right 방향키	선택된 객체를 해당 객체의 -Z 방향에서 바라봅니다.

마우스 사용

실행상태에서 왼쪽 마우스 버튼을 누른 채 마우스를 움직이며, 카메라의 방향이 변경됩니다. 그리고 편집 모드에서는 특정 객체를 선택하고, 좌측의 속성 창에서 Position Vector 항목을 선택한 후 왼쪽 마우스 버튼을 클릭한 채로 마우스를 움직이면 해당 객체를 움직일 수 있습니다. 또한 Rotation Vector 항목을 선택한 후 마찬가지로 마우스를 움직이면 해당 객체를 회전시킬 수 있습니다.

시뮬레이션 좌표 시스템

시뮬레이터는 오른손 좌표계를 사용합니다. +Y 축의 방향은 위쪽을 가리키며, X와 Z축은 바닥과 평행한 부분을 가리킵니다.

3. 시뮬레이션 툴 및 유틸리티

Obj-to-Bos 파일 변환 툴 (Obj2Bos.exe)

“.obj” 파일을 “.bos” 파일로 변환하는 툴로서, 오브젝트 시뮬레이션 메쉬 파일을 최적화된 바이너리 파일로 변환합니다.

```
obj2bos.exe [options]
```

파라메터

Option	단축	설명
/infile:<string>	/i	.obj 파일을 .bos 포멧으로 변환합니다. 와일드카드 문자열은 * 과 ? 가 사용될 수 있습니다.
/level:{None|Default|Maximum}	/l	최적화 수준을 나타냅니다. 높은 최적화 값을 지정할 경우 압축 시간은 늘어나지만 mesh 파일의 크기가 줄어듭니다.
/outfile:<string>	/o	출력파일 이름이며 옵션 사항입니다. 별도로 지정하지 안으면 동일한 파일 이름의 .bos 파일이 생성됩니다.
/verbosity:{Off|Error|Warning|Info|Verbose}	/v	Verbosity 수준을 정의합니다.
@<file>		여러 변환 대상 파일의 이름을 포함하고 있는 텍스트 파일의 이름을 지정합니다.

사용예

```
Obj2Bos.exe /i:"store₩media₩earth.obj"
```

4. 시뮬레이션 튜토리얼

시뮬레이션 튜토리얼 1 (C#) - 시뮬레이션 런타임에 대한 소개

MSRS는 물리엔진에 기반한 다양한 가상 환경에서 로봇 어플리케이션을 개발할 수 있는 환경을 제공합니다. 이번 튜토리얼은 시뮬레이션 엔진 서비스와 렌더링 윈도우를 어떻게 시작하는 지에 대해 보여주며, 두 개의 객체를 가상 공간에 추가하는 방법을 보여줍니다.

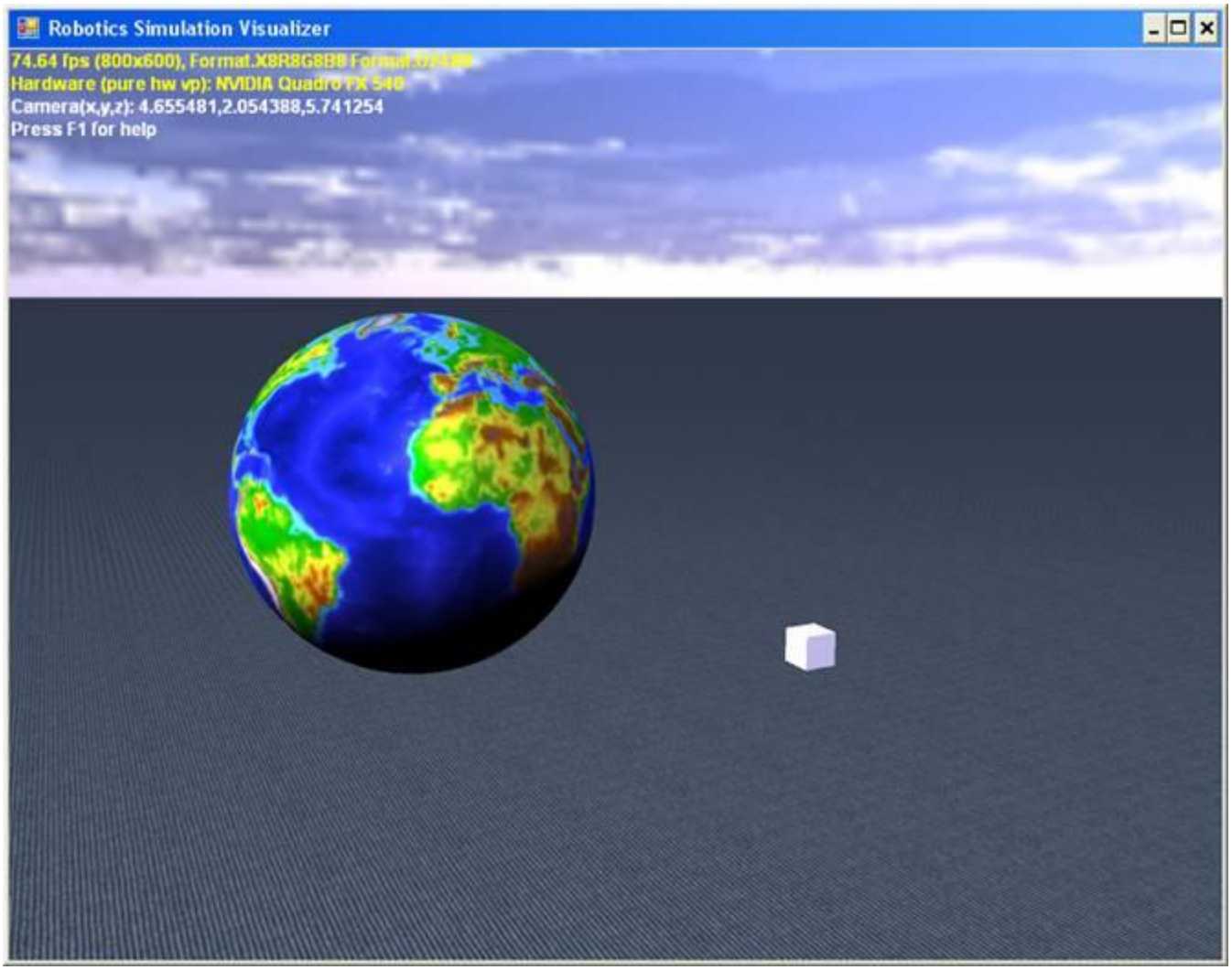


그림 1 - 시뮬레이션 실행 화면

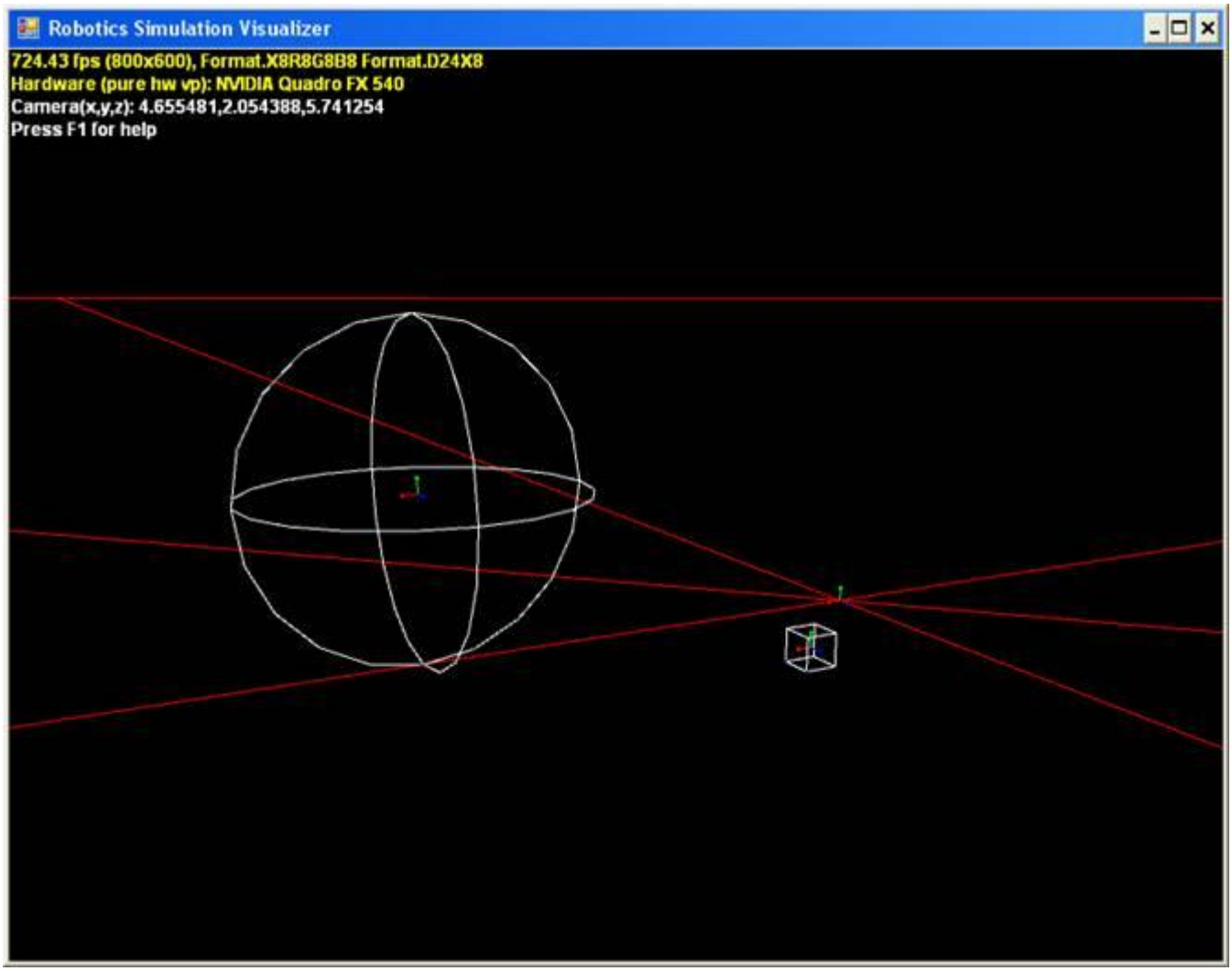


그림 2 - 시뮬레이션의 물리적 표현 모드

시뮬레이션 실행모듈은 다음과 같은 컴포넌트로 구성되어 있습니다.

- 시뮬레이션 엔진 서비스 - 엔터티들의 렌더링과 물리엔진에서의 시뮬레이션 시간을 처리합니다. 시뮬레이션 공간 전체의 상태를 트래킹하고 서비스와 시뮬레이션 앞 단 부분을 처리합니다.
- Managed 물리엔진 랩퍼 - 저수준의 물리엔진의 API에 대해 C# 형태의 인터페이스를 노출합니다.
- AGEIA PhysX 기술 - AGEIA 물리 프로세서를 통한 하드웨어 가속 기능을 제공합니다 (별도의 하드웨어 카드 추가시)
- 엔터티 - 시뮬레이션 공간 상에서의 하드웨어 및 물리 객체를 나타냅니다. 다양한 엔터티들이 미리 정의되어 있으며, 이러한 엔터티들을 조합하여 새로운 형태의 엔터티들을 생성할 수 있고 다양한 형태의 로봇 플랫폼을 개발할 수 있습니다.

사전 요구 사항
하드웨어

시뮬레이션을 구동하기 위해서는 PC의 그래픽 카드가 이를 지원하는 카드이어야 가능합니다. 지원되는 그래픽 카드는 아래의 링크를 통해 확인 가능합니다.

http://channel9.msdn.com/wiki/default.aspx/Channel9.SimulationFAQ

소프트웨어

시뮬레이션 프로그래밍을 위해서는 아래의 툴 중 한가지가 필요합니다.

- Microsoft Visual C# 2005 Express Edition.
- Microsoft Visual Studio Standard Edition.
- Microsoft Visual Studio Professional Edition.
- 또는 Microsoft Visual Studio Team System.

시작하기

이번 튜토리얼의 소스 코드는 아래의 폴더에 미리 작성되어 있습니다.

    SamplesWSimulationTutorialsWTutorial1

Visual Studio에서 SimulationTutorial1.csproj 파일을 로딩함으로써 튜토리얼을 시작합니다. 프로젝트를 빌드한 후 MSRS Commnad 창에서 아래와 같이 실행시켜 봄으로써 결과를 확인해 볼 수 있습니다.

dsshost /port:50000 /manifest:"samplesWconfigWSimulationTutorial1.manifest.xml"

Visual Studio에서 프로젝트를 로딩한 경우, 해당 프로젝트 속성에 위의 실행 명령어가 등록되어 있으므로, 개발 툴에서 F5키를 눌러 직접 실행을 시키면 해당 프로그램을 실행시킬 수 있습니다.

각각의 코드 구성에 대한 설명은 아래와 같습니다.

Step 1: 참조 구성

시뮬레이션을 실행시키기 위해서는 아래의 dll 파일들을 반드시 참조에 추가해 놓아야 하며, 이러한 dll 파일들은 MSRS의 설치 폴더 안에 bin 폴더에 존재합니다.

- RoboticsCommon.DLL - PhysicalModel namespace를 포함하고 있으며, 물리적인 로봇 속성을 모델링하기 위한 공통 값들이 정의되어 있습니다.
- PhysicsEngine.DLL - Native physics engine dll에 대한 Managed C++ 랩퍼입니다.
- SimulationCommon.DLL - 공통 타입들이 정의되어 있습니다.
- SimulationEngine.DLL - 랜더링 엔진, 시뮬레이션 상태 관리와 서비스 앞 단 처리를 수행합니다.
- SimulationEngine.Proxy.DLL - 시뮬레이션 엔진 상태의 Proxy이며 엔진이 파트너 서비스로 호출될 때 사용됩니다.

각 참조항목들은 Copy Local Property가 False로 설정되어 있어야 합니다.

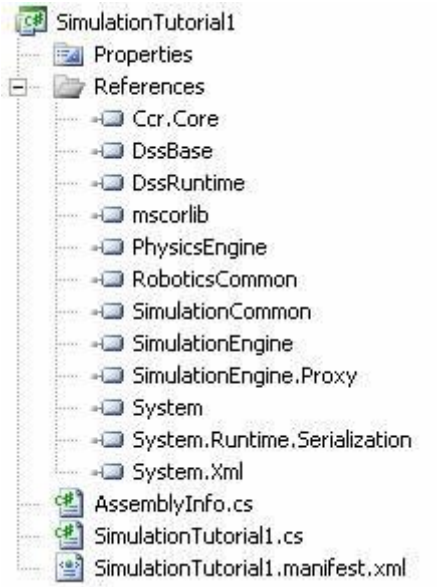


그림 3 - SimulationTutorial1 참조 항목

SimulationTutorial1.cs 파일에는 다음과 같은 Using 구분이 사용됩니다.

```
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;

using System;
using System.Collections.Generic;
using Microsoft.Robotics.Simulation;
using Microsoft.Robotics.Simulation.Engine;
using engineproxy = Microsoft.Robotics.Simulation.Engine.Proxy;
using Microsoft.Robotics.Simulation.Physics;
using Microsoft.Robotics.PhysicalModel;
using System.ComponentModel;
```

Step 2: 시뮬레이션 엔진 시작하기

시뮬레이션 엔진은 시뮬레이션 공간을 랜더링하는 윈도우를 실행시킵니다. 이 모듈이 MSRS의 서비스 형태로 되어 있기 때문에, 이 모듈을 구동하기 위해서는 해당 서비스를 파트너로 등록해 놓아야 합니다.

```
[DisplayName("Simulation Tutorial 1")]
[Description("Simulation Tutorial 1 Service")]
[Contract(Contract.Identifier)]
public class SimulationTutorial1 : DsspServiceBase
{
    [Partner("Engine",
        Contract = engineproxy.Contract.Identifier,
        CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
    private engineproxy.SimulationEnginePort _engineStub =
```

```
        new engineproxy.SimulationEnginePort();

    // Main service port
    [ServicePort("/SimulationTutorial1", AllowMultipleInstances=false)]
    private SimulationTutorial1Operations _mainPort =
        new SimulationTutorial1Operations();

    public SimulationTutorial1(DsspServiceCreationPort creationPort) :
            base(creationPort)
    {

    }
```

이미 해당 엔진이 매니페스트를 통해 실행되는 경우도 있기 때문에 파트너 속성에서 CreationPolicy.UseExistingOrCreate로 지정하였음을 유의하시기 바랍니다. _engineStub 변수는 단순히 파트너를 로딩시키기 위한 목적 이외에는 다른 용도로 사용되지 않습니다. 대신에 튜토리얼에서는 시뮬레이션 엔진에 대한 정적 참조를 사용하며 직접 메시지를 전달합니다. 이러한 통신은 서비스 처리 방식에서 예외적인 부분이며, 성능을 극대화하기 위해 적용된 방식입니다.

Step 3: 엔터티를 환경에 추가하기

Start 메소드에서 SetupCamera와 PopulateWorld 메소드가 호출됩니다. SetupCamera에서는 카메라의 방향과 위치를 지정하며, PopulateWorld에서는 간단한 환경 엔터티들을 추가합니다.

```
protected override void Start()
{
    base.Start();
    // Orient sim camera view point
    SetupCamera();
    // Add objects (entities) in our simulated world
    PopulateWorld();
}
```

SetupCamera는 아래와 같이 정의됩니다.

```
private void SetupCamera()
{
    // Set up initial view
    CameraView view = new CameraView();
    view.EyePosition = new Vector3(-1.65f, 1.63f, -0.29f);
    view.LookAtPoint = new Vector3(-0.68f, 1.38f, -0.18f);
    SimulationEngine.GlobalInstancePort.Update(view);
}
```

Sky 추가하기

Sky 객체는 다음과 같이 추가됩니다.

```
void AddSky()
{
    // Add a sky using a static texture. We will use the sky texture
    // to do per pixel lighting on each simulation visual entity
    SkyDomeEntity sky = new SkyDomeEntity("skydome.dds", "sky_diff.dds");
    SimulationEngine.GlobalInstancePort.Insert(sky);
```

```
    // Add a directional light to simulate the sun.
    LightSourceEntity sun = new LightSourceEntity();
    sun.State.Name = "Sun";
    sun.Type = LightSourceEntityType.Directional;
    sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
    sun.Direction = new Vector3(0.5f, -.75f, 0.5f);
    SimulationEngine.GlobalInstancePort.Insert(sun);
}
```

SkyEntity 클래스는 파라메터로 2개의 파일 이름을 필요로 합니다.

- Sky 비트맵을 위한 텍스쳐 파일 - 세부적인 하늘을 묘사하기 위한 고해상도 이미지
- 단순화된 큐브 맵 형태의 텍스쳐 파일 - 픽셀단위 광원 효과를 위한 파일

LightEntity 클래스는 광원 효과를 시뮬레이션 하기 위해 사용됩니다.

평지 추가

이제 평지를 시뮬레이션 공간에 추가합니다. 평지는 다른 객체를 추가하기 전에 미리 추가되어야 하며, 그렇지 안으면 다른 객체들이 모두 평지 아래로 떨어져 버립니다.

```
void AddGround()
{
    // create a large horizontal plane, at zero elevation.
    HeightFieldEntity ground = new HeightFieldEntity(
        "simple ground", // name
        "03RamieSc.dds", // texture image
        new MaterialProperties("ground",
            0.2f, // restitution
            0.5f, // dynamic friction
            0.5f) // static friction
        );
    SimulationEngine.GlobalInstancePort.Insert(ground);
}
```

Ground 엔터티는 미리 정의되어 있는 HeightFieldEntity를 사용하며, 이 객체는 다양한 Height들의 배열로 구성됩니다. 위의 예제는 평지를 구성한 예를 보여줍니다.

Ground 엔터티를 생성한 다음에는 반드시 Insert 메소드를 통해 시뮬레이션 엔진에 등록시켜 주어야 합니다.

단순한 물체 추가

다음으로는 단순한 박스를 추구하는 코드입니다. 만약 별도의 메쉬 파일이 제공되지 않는 다면, 시뮬레이션 엔진은 기본적인 색상으로 해당 객체를 랜더링 합니다.

```
void AddBox(Vector3 position)
{
    Vector3 dimensions =
        new Vector3(0.2f, 0.2f, 0.2f); // meters

    // create simple movable entity, with a single shape
```

```
    SingleShapeEntity box = new SingleShapeEntity(
        new BoxShape(
            new BoxShapeProperties(
            100, // mass in kilograms.
            new Pose(), // relative pose
            dimensions)), // dimensions
        position);

    box.State.MassDensity.Mass = 0;
    box.State.MassDensity.Density = 0;

    // Name the entity. All entities must have unique names
    box.State.Name = "box";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(box);
}
```

미리 정의되어 있는 SingleShapeEntity는 시뮬레이션 공간에 객체를 추가하는 데 있어서 좋은 출발점이 될 수 있으며, 구, 박스 또는 캡슐 형태의 객체를 추가할 수 있습니다. 이러한 모양의 객체를 추가할 때, 질량과 상대적 위치, 그리고 해당 물체의 크기를 같이 지정합니다.

SingleShapeEntity를 사용함에 있어서 유의할 사항은 아래와 같습니다.

- 질량과 밀도가 정의되지 않은 Shape은 정적이며, 움직일 수 없고 무한의 질량을 가진 것으로 가정됩니다.
- Shape은 물리엔진에서 엔터티로 묘사되며, 만약 메쉬 파일이 제공된다면, 랜더링은 물리적 표현과 별개로 처리됩니다.
- Pose 클래스는 객체의 위치와 방향을 같이 나타냅니다. 이 값은 해당 엔터티 내에서 상대적인 위치 값으로 표시됩니다.
- 모든 엔터티들은 고유의 식별되는 이름을 반드시 가져야 합니다. 만약 이름이 중복되는 경우에는 해당 엔터티가 표시되지 않습니다.

메쉬 기반의 텍스쳐 엔터티

시뮬레이션 엔진은 실제와 유사한 사진 기반의 형태로 시뮬레이션 공간상에서 사물을 쉽게 표현할 수 있습니다. 이렇게 표현하기 위해서는 파일 기반의 DirectX 메쉬 파일이 필요합니다. 아래의 예제에서는 이러한 메쉬 파일에 기반하여 객체를 랜더링 하는 간단한 사례를 보여줍니다.

```
void AddTexturedSphere(Vector3 position)
{
    SingleShapeEntity entity = new SingleShapeEntity(
        new SphereShape(
            new SphereShapeProperties(10, // mass in kg
            new Pose(), // pose of shape within entity
            1)), //default radius
        position);

    entity.State.Assets.Mesh = "earth.obj";
    entity.SphereShape.SphereState.Material = new MaterialProperties("sphereMaterial",
0.5f, 0.4f, 0.5f);
```

```
    // Name the entity
    entity.State.Name = "detailed sphere";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(entity);
}
```

파일 기반의 메쉬 파일을 사용하는 데 있어서 유의할 사항은 아래와 같습니다.

- Asset 멤버에서 정의되어 있는 모든 파일 경로는 storeWMedia 내에 있는 것으로 간주됩니다. 만약 오직 파일이름 만 사용하는 경우에는 경로는 무시되며, 해당 파일을 storeWMedia 폴더 밑에 모두 복사해 두어야 합니다.
- 메쉬를 사용하는 경우, 물리 Shape을 계산하기 위해 TriangleMeshEnvironmentEntity 또는 SimplifiedConvexMeshEnvironmentEntity와 같은 고급 엔터티 기능을 사용할 수 있습니다.

Step 4: 튜토리얼 실행

SimulationTutorial1.csproj 파일을 로드한 후 컴파일 과정을 거쳐 실행시킵니다. 정상적으로 실행된다면, 먼저 비주얼 윈도우 화면이 표시되고, 이 튜토리얼의 앞 부분에 있는 그림과 같은 화면이 표시됩니다.

단순한 시뮬레이션 공간을 이동시켜 보기 위해 단축키를 눌러 화면을 이동시켜 봅니다.

단축키

F1 - 도움말을 숨김
F2 - 랜더링 모드를 변경함

카메라 이동

- A - 왼쪽으로 이동
- D - 오른쪽으로 이동
- W - 앞으로 이동
- S - 뒤로 이동
- Q - Y 축을 따라 위로 이동
- E - Y 축을 따라 아래로 이동

마우스 컨트롤

카메라의 방향을 마우스를 통해 이동시킬 수 있습니다.

Xbox360 게임패드 컨트롤

왼쪽 엄지 손가락의 컨트롤을 통해 카메라의 위치를 이동시킬 수 있습니다. 오른쪽 엄지 손가락의 컨트롤을 통해 카메라의 방향을 이동시킬 수 있습니다.

시뮬레이션 튜토리얼 2 (C#) - 시뮬레이션 서비스를 통한 복합 엔터티

이번 튜토리얼 에서는 일반적인 엔터티를 어떻게 생성하고 모듈화된 시뮬레이션 로봇을 어떻게 생성하는 지에 대해 설명합니다.

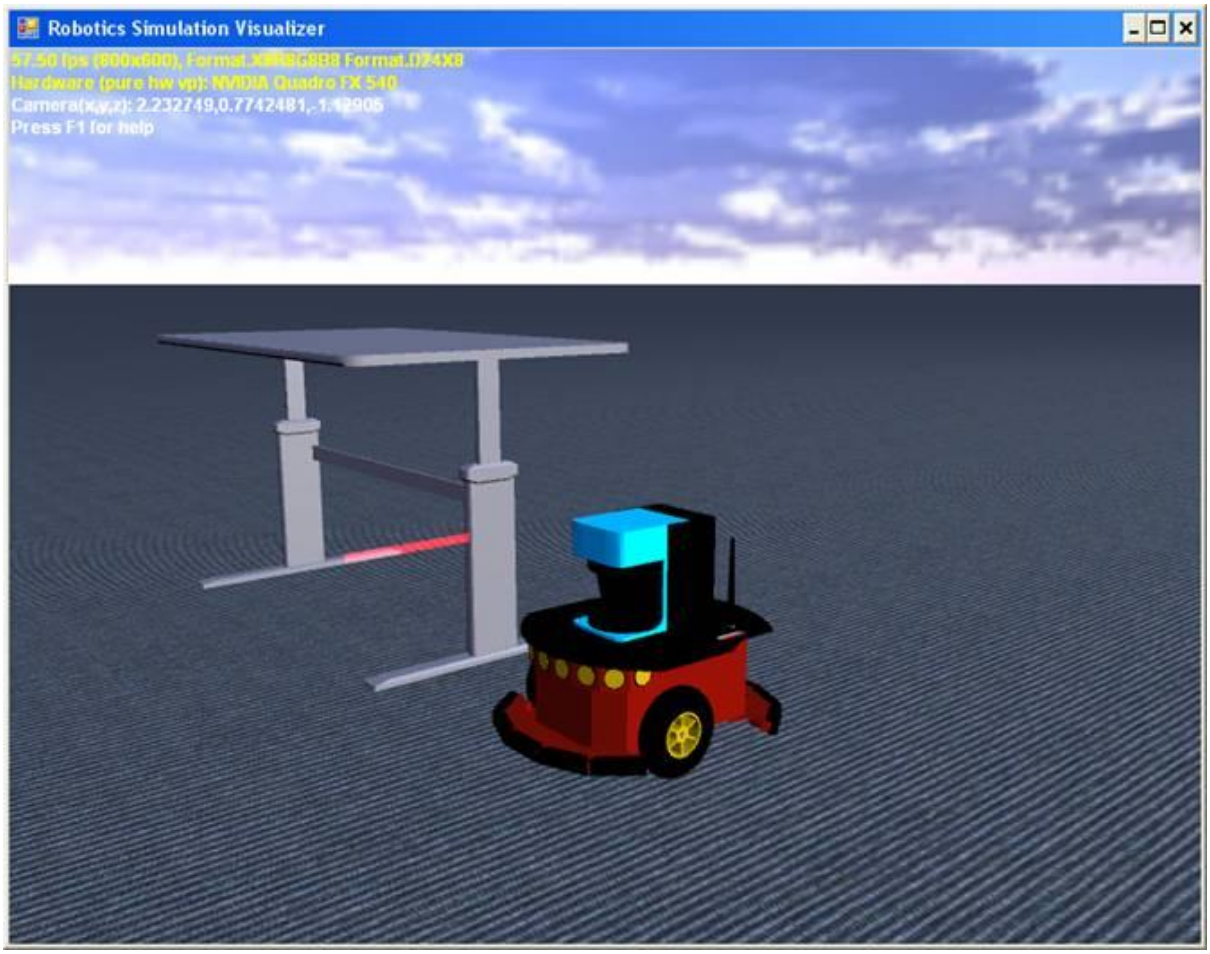


그림 1 - 시뮬레이션 실행 화면

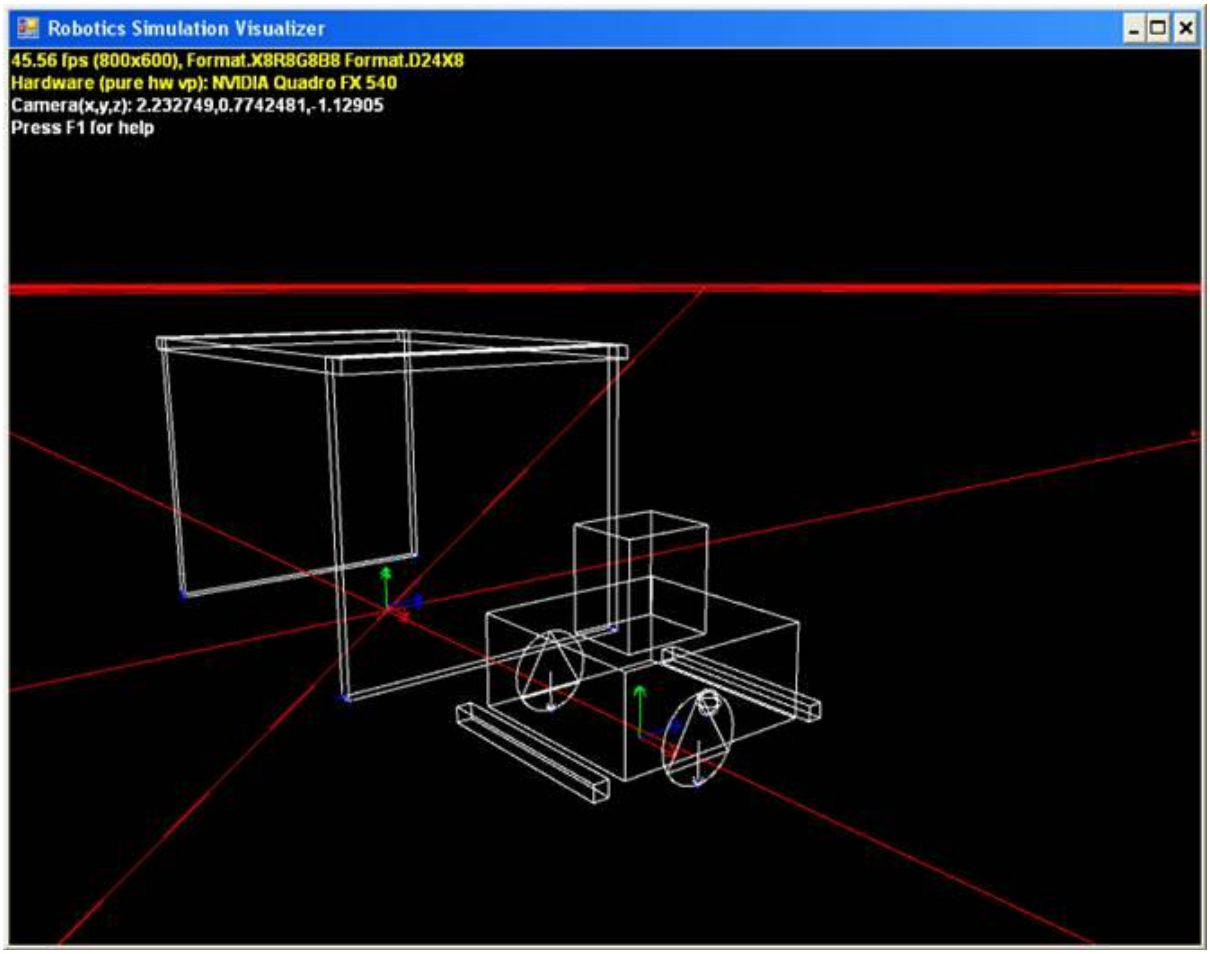


그림 2 - 시뮬레이션의 물리적 표현 화면

시작하기

이번 튜토리얼은 SamplesWSimulationTutorialsWTutorial2 밑에 소스 코드가 해당 폴더에 있는 SimulationTutorial2.csproj 파일을 로드하여 프로젝트를 실행할 수 있습니다.

해당 프로젝트를 실행하기 위해서는 F5 키를 눌러 Visual Studo에서 바로 실행할 수도 있으며, 아래와 같이 MSRS 실행창에서 명령어를 통해 실행할 수도 있습니다.

```
binWdsshost /port:50000 /manifest:"samplesWconfigWSimulationTutorial2.manifest.xml"
```

Step 1: 참조 추가

시뮬레이션을 실행하기 위해서는 아래와 같은 dll 파일을 참조에 추가해 놓아야 합니다.

- SimulatedBumper.Y2006.M05.Proxy
- SimulatedDifferentialDrive.2006.M06.Proxy
- SimulatedLRF.Y2006.M05.Proxy
- SimulatedWebcam.Y2006.M09.Proxy

또한 다음의 Using 문장을 프로그램의 상단에 추가해야 합니다.

```
using drive = Microsoft.Robotics.Services.Simulation.Drive.Proxy;
using lrf = Microsoft.Robotics.Services.Simulation.Sensors.LaserRangeFinder.Proxy;
```

```
using bumper = Microsoft.Robotics.Services.Simulation.Sensors.Bumper.Proxy;
using simwebcam = Microsoft.Robotics.Services.Simulation.Sensors.SimulatedWebcam.Proxy;
```

서비스 시작

튜토리얼 1에서 보다 확장되어 좀 더 복잡한 형태의 엔터티들이 시뮬레이션에 추가됩니다.

```
private void SetupCamera()
{
    // Set up initial view
    CameraView view = new CameraView();
    view.EyePosition = new Vector3(2.491269f, 0.598689f, 1.046625f);
    view.LookAtPoint = new Vector3(1.873792f, 0.40983f, 0.2830455f);
    SimulationEngine.GlobalInstancePort.Update(view);
}

private void PopulateWorld()
{
    AddSky();
    AddGround();
    AddCameras();
    AddTable(new Vector3(1, 0.5f, -2));
    AddPioneer3DXRobot(new Vector3(1, 0.1f, 0));
    AddLegoNxtRobot(new Vector3(2, 0.1f, 0));
    //AddIRobotCreateRobot(new Vector3(2, 0.1f, 0));  // uncomment this to add an iRobot
Create robot
}
```

Table 엔터티는 프로그램에 의해 생성된 다중 Shape 엔터티입니다. 그리고 Pioneer3DX와 LegoNxt는 디퍼런셜 드라이드 기반의 로봇으로서 각각 레이저 파인더와 2개의 범퍼 센서 그리고 한 개의 전방 범퍼 센서를 포함하고 있습니다.

Step 2: 다중 Shape을 통한 환경 엔터티 정의

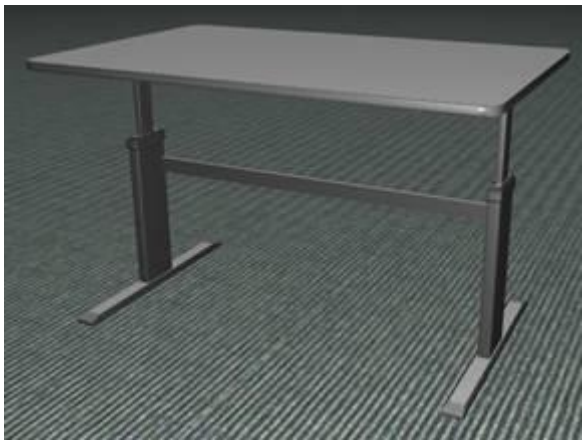


그림 3 - 시뮬레이션에서의 Table 객체 표현

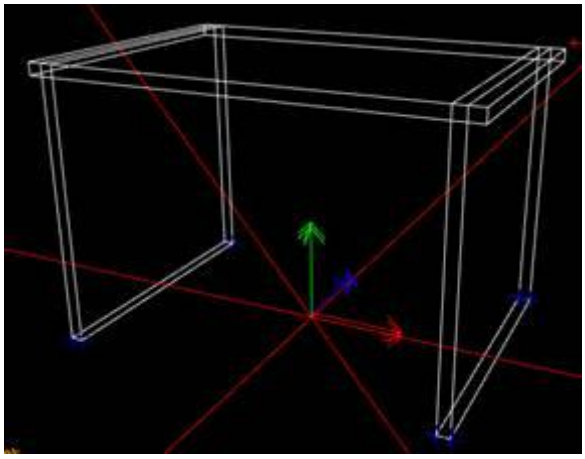


그림 4 - 시뮬레이션에서의 Table의 물리적 표현

복잡한 형태의 물체를 손쉽게 생성하기 위해 기본적인 Shape들의 집합을 사용할 수 있습니다. 위의 Table 예에서는 여러 개의 Box Shape을 결합하여 하나의 엔터티로 생성을 한 예입니다.

```
void AddTable(Vector3 position)
{
    // create an instance of our custom entity
    TableEntity entity = new TableEntity(position);

    // Name the entity
    entity.State.Name = "table:"+Guid.NewGuid().ToString();

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(entity);
}
```

TableEntity 클래스는 MultiShapeEntity 클래스로부터 유도되었으며, 여러 개의 박스를 결합하여 테이블을 생성합니다. 엔터티 정의는 SimulationTutorial2.cs 파일의 아래 부분에 기술되어 있습니다.

```
/// <summary>
/// An entity for approximating a table.
/// </summary>
[DataContract]
public class TableEntity : MultiShapeEntity
{
    /// <summary>
    /// Default constructor.
    /// </summary>
    public TableEntity() { }

    /// <summary>
    /// Custom constructor, programmatically builds physics primitive shapes to describe
    /// a particular table.
    /// </summary>
    /// <param name="position"></param>
    public TableEntity(Vector3 position)
    {
```

```
        State.Pose.Position = position;
        State.Assets.Mesh = "table_01.obj";
        float tableHeight = 0.65f;
        float tableWidth = 1.05f;
        float tableDepth = 0.7f;
        float tableThinkness = 0.03f;
        float legThickness = 0.03f;
        float legOffset = 0.05f;

        // add a shape for the table surface
        BoxShape tableTop = new BoxShape(
            new BoxShapeProperties(30,
            new Pose(new Vector3(0, tableHeight, 0)),
            new Vector3(tableWidth, tableThinkness, tableDepth))
        );

        // add a shape for the left leg
        BoxShape tableLeftLeg = new BoxShape(
            new BoxShapeProperties(10, // mass in kg
            new Pose(
                new Vector3(-tableWidth/2 + legOffset, tableHeight/2, 0)),
            new Vector3(legThickness, tableHeight + tableThinkness, tableDepth))
        );

        BoxShape tableRightLeg = new BoxShape(
            new BoxShapeProperties(10, // mass in kg
            new Pose(
                new Vector3(tableWidth / 2 - legOffset, tableHeight / 2, 0)),
            new Vector3(legThickness, tableHeight + tableThinkness, tableDepth))
        );

        BoxShapes = new List<BoxShape>();
        BoxShapes.Add(tableTop);
        BoxShapes.Add(tableLeftLeg);
        BoxShapes.Add(tableRightLeg);
    }

    public override void Update(FrameUpdate update)
    {
        base.Update(update);
    }
}
```

자체의 엔터티를 생성하는 데 있어서 유의할 점은 아래와 같습니다.

- 물리적 파라메터와 Shape을 기술하기 위해 커스텀 생성자를 사용합니다. 만약 시뮬레이션 상태를 캡쳐하고 상태 파일을 로드할 경우, 기본 생성자가 구동되고 동일한 엔터티가 시리얼라이즈된 값을 이용해 재생성 될 것입니다.
- 복잡한 동작이 필요치 않다면 이미 존재하는 엔터티를 이용해 새로운 엔터티를 생성합니다. MultiShapeEntity는 Box와 Sphere와 같은 다양한 형상의 객체들에 대한 리스트를 지원합니다.

Step 3: 모듈화된 로봇과 시뮬레이션 하드웨어 서비스 생성

그림 5 - Pioneer3DX 로봇의 시뮬레이션 화면

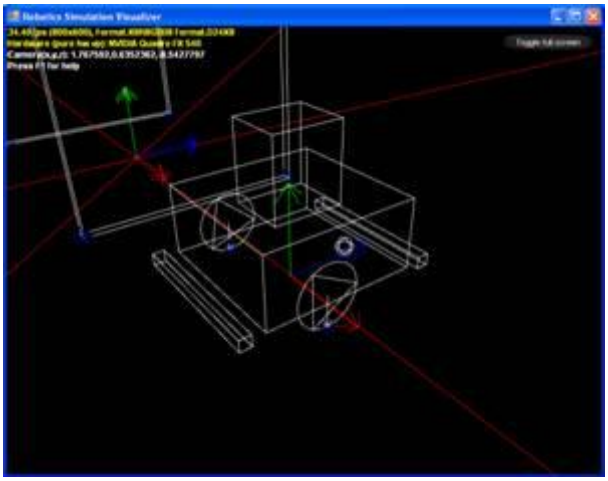


그림 6 - Pioneer3DX 로봇의 물리적 표현 화면

시뮬레이션 런타임은 몇 개의 사전에 정의되어 있는 휠 로봇 기반의 로봇들을 실행시킵니다. 이러한 로봇들은 디퍼런셜 드라이브와 몇 개의 센서들을 포함하고 있습니다. 이번 시뮬레이션 튜토리얼에서는 2개의 로봇들을 사용할 예정이며, 첫 번째로 Pioneer3DX 로봇에 대한 인스턴스를 아래와 같이 생성하여 사용합니다. 이러한 로봇은 기본적으로 DifferentialDriveEntity 를 확장하여 생성한 로봇 엔터티 입니다.

```
void AddPioneer3DXRobot(Vector3 position)
{
    Pioneer3DX robotBaseEntity = CreateMotorBase(ref position);

    // Create Laser entity and start simulated laser service
    LaserRangeFinderEntity laser = CreateLaserRangeFinder();
    // insert laser as child to motor base
    robotBaseEntity.InsertEntity(laser);

    // Create bumper array entity and start simulated bumper service
    BumperArrayEntity bumperArray = CreateBumperArray();
    // insert as child of motor base
    robotBaseEntity.InsertEntity(bumperArray);
```

```
    // create Camera Entity ans start SimulatedWebcam service
    CameraEntity camera = CreateCamera();
    // insert as child of motor base
    robotBaseEntity.InsertEntity(camera);

    // Finally insert the motor base and its two children
    // to the simulation
    SimulationEngine.GlobalInstancePort.Insert(robotBaseEntity);
}
```

모듈화된 시뮬레이션 로봇은 아래와 같이 3 단계를 거쳐 생성됩니다.

1. 디퍼런셜 드라이브 기반의 기본 모터를 생성합니다. Pioneer3DX와 LegoNXT의 디퍼런셜 드라이브는 기본적으로 동일한 클래스를 상속받아 생성되었으며, 두 로봇 사이에는 차이가 없습니다.
2. 가상 하드웨어에 대해 LaserRangeFinderEntity와 레이저 파인더 시뮬레이션 서비스의 인스턴스를 생성합니다.
3. 두개의 범퍼에 대한 BumperArrayEntity와 범퍼 시뮬레이션 서비스의 인스턴스를 생성합니다.
4. CameraEntity와 웹캠 서비스에 대한 시뮬레이션 서비스의 인스턴스를 생성합니다.

위의 과정을 통해 레이저, 범퍼, 카메라를 생성한 후에는 모터 기반 엔터티에 해당 엔터티들을 자식 엔터티로 추가해 주어야 합니다.

```
robotBaseEntity.InsertEntity(bumperArray);
```

디퍼런셜 드라이브 엔터티

부모 엔터티는 Pioneer3DX 로봇의 물리 형상을 가진 디퍼런셜 모터 클래스로부터 생성됩니다. CreateEntityPartner 메소드는 엔터티가 런타임 시에 바인딩될 필요가 있다는 것을 서비스에게 알려주어 서비스 파트너를 생성시킵니다.

```
private Pioneer3DX CreateMotorBase(ref Vector3 position)
{
    // use supplied entity that creates a motor base
    // with 2 active wheels and one caster
    Pioneer3DX robotBaseEntity = new Pioneer3DX(position);

    // specify mesh.
    robotBaseEntity.State.Assets.Mesh = "Pioneer3dx.bos";
    // specify color if no mesh is specified.
    robotBaseEntity.ChassisShape.State.DiffuseColor = new Vector4(0.8f, 0.25f, 0.25f,
1.0f);

    // the name below must match manifest
    robotBaseEntity.State.Name = "P3DXMotorBase";
    // Start simulated arcos motor service
    drive.Contract.CreateService(ConstructorPort,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
            "http://localhost/" + robotBaseEntity.State.Name)
    );
```

```
    return robotBaseEntity;
}
```

Laser Range Finder 엔터티

아래의 코드는 레이저 엔터티와 관련된 서비스가 어떻게 시작되는 지에 대해 간략히 보여 줍니다. 엔터티의 이름과 파트너 이름이 매칭되어야 한다는 것에 유의하시기 바랍니다.

```
private LaserRangeFinderEntity CreateLaserRangeFinder()
{
    // Create a Laser Range Finder Entity .
    // Place it 30cm above base CenterofMass.
    LaserRangeFinderEntity laser = new LaserRangeFinderEntity(
        new Pose(new Vector3(0, 0.30f, 0)));

    laser.State.Name = "P3DXLaserRangeFinder";
    laser.LaserBox.State.DiffuseColor = new Vector4(0.25f, 0.25f, 0.8f, 1.0f);

    // Create LaserRangeFinder simulation service and specify
    // which entity it talks to
    Irf.Contract.CreateService(
        ConstructorPort,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
        "http://localhost/" + laser.State.Name));
    return laser;
}
```

센서 엔터티에 관해 한가지 흥미로운 부분은 상대적인 위치가 모터 본체 위에 항상 위치해야 한다는 것입니다. 모듈화된 로봇의 경우 센서와 추가적인 하드웨어들은 부모 엔터티의 용어로 기술되어야 하며, 결합된 형태가 아닌 독립적인 형태로 기술될 필요가 있습니다.

Bumper Array 엔터티

이번 섹션에서는 모터 본체의 앞 뒤에 위치한 두 개의 Shape을 사용하여 Bumper Array를 생성하는 방법에 대해 보여줍니다. Pioneer 메쉬 파일은 10개의 범포를 사용하는 것으로 보여주고 있지만, 시뮬레이션에서는 단지 2개의 범퍼만 사용합니다. 범퍼 Shpae의 상대적인 위치에 유의하시기 바랍니다.

```
private BumperArrayEntity CreateBumperArray()
{
    // Create a bumper array entity with two bumpers
    BoxShape frontBumper = new BoxShape(
        new BoxShapeProperties("front",
            0.001f,
            new Pose(new Vector3(0, 0.05f, -0.25f)),
            new Vector3(0.40f, 0.03f, 0.03f)
        )
    );
    frontBumper.State.DiffuseColor = new Vector4(0.1f, 0.1f, 0.1f, 1.0f);

    BoxShape rearBumper = new BoxShape(
```

```
        new BoxShapeProperties("rear",
            0.001f,
            new Pose(new Vector3(0, 0.05f, 0.25f)),
            new Vector3(0.40f, 0.03f, 0.03f)
        )
    );
    rearBumper.State.DiffuseColor = new Vector4(0.1f, 0.1f, 0.1f, 1.0f);

    // The physics engine will issue contact notifications only
    // if we enable them per shape
    frontBumper.State.EnableContactNotifications = true;
    rearBumper.State.EnableContactNotifications = true;

    // put some force filtering so we only get notified for significant bumps
    //frontBumper.State.ContactFilter = new ContactNotificationFilter(1,1);
    //rearBumper.State.ContactFilter = new ContactNotificationFilter(1, 1);

    BumperArrayEntity
        bumperArray = new BumperArrayEntity(frontBumper, rearBumper);
    // entity name, must match manifest partner name
    bumperArray.State.Name = "P3DXBumpers";

    // start simulated bumper service

    bumper.Contract.CreateService(
        ConstructorPort,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
        "http://localhost/" + bumperArray.State.Name));
    return bumperArray;
}
```

Bumper Array 엔터티는 물리 공간에서 다른 물체와 충돌을 알리기 위해 ContractNotification 방식을 적용합니다. 범퍼가 충돌을 감지하기 위해서는 아래와 같은 속성이 설정되어야 합니다.

```
frontBumper.State.EnableContactNotifications = true;
```

카메라 엔터티

이본 섹션에서는 실시간을 시뮬레이션 공간상의 영상을 캡쳐해서 전송하는 카메라를 어떻게 추가시키는 지에 설명을 합니다. 카메라는 모터 본체에 부착되어 있으며, 이로 인해 로봇이 움직일 때 같이 움직이도록 되어 있습니다.

시뮬레이션 엔진은 CameraEntity를 생성하는 데 있어서 다중 인스턴스를 지원하며, 아래와 같이 2가지의 모드로 작동될 수 있습니다.

1. 실시간 모드 - CameraEntity는 임의의 카메라로부터 전체 영역의 영상을 실시간으로 랜더링 할 수 있습니다. 이 모드는 웹캠 센서를 모델링 하는 데 유용하게 사용됩니다. 실시간 카메라는 영상의 랜더링을 중복적으로 수행하기 때문에 높은 랜더링 자원을 소모합니다.
2. 비실시간 모드 - CameraEntity는 시뮬레이션 상에서 오직 활성화 될 때만 랜더링을 합니다. 따라서 랜더링 자원이 많오 소요되지 않습니다.

```
private CameraEntity CreateCamera()
{
    // low resolution, wide Field of View
    CameraEntity cam = new CameraEntity(320, 240);
    cam.State.Name = "robocam";
    // just on top of the bot
    cam.State.Pose.Position = new Vector3(0.0f, 0.5f, 0.0f);
    // camera renders in an offline buffer at each frame
    // required for service
    cam.IsRealTimeCamera = true;

    // Start simulated webcam service
    simwebcam.Contract.CreateService(
        ConstructorPort,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
            "http://localhost/" + cam.State.Name)
    );

    return cam;
}
```

Step 4: 시뮬레이션 서비스와 엔터티 파트너쉽

서비스와 하드웨어 시뮬레이션 엔터티와의 파트너링

CreateService 메소드는 런터임시에 해당 서비스 정보의 요소로서 파트너 요소를 제공하는 것과 함께 주어진 Contract에 대한 서비스 인스턴스를 생성하도록 합니다. SimulatedDifferentialDrive 서비스의 구현은 시뮬레이션 서비스를 어떻게 찾고 엔터티 파트너를 어떻게 이용하는 지에 대해 구체적으로 보여줍니다. 아래 코드는 CreateMotorNase 메소드에서 사용된 코드입니다.

```
// Start simulated arcos motor service
drive.Contract.CreateService(ConstructorPort,
    Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
        "http://localhost/" + robotBaseEntity.State.Name)
);
```

시뮬레이션 서비스 디자인 패턴

시뮬레이션 서비스들은 엔터티 생성 과정시 비동기 특성을 확실하게 하기 위해 특별한 시작 패턴을 따라야 합니다. 엔터티가 서비스가 시작된 후에 시뮬레이션에 추가될 수도 있기 때문에, 서비스는 엔터티 Notification이 도착할 때 까지 DSSP 오퍼레이션 프로세싱을 허용하지 않도록 유의해야 합니다. 아래 코드는 SimulatedDifferentialDrive 서비스에서 사용된 코드로서 이러한 패턴의 사례를 보여주고 있습니다.

```
protected override void Start()
{
    // Find our simulation entity that represents the "hardware" or real-world service.
    // To hook up with simulation entities we do the following steps
    // 1) have a manifest or some other service create us, specifying a partner named
SimulationEntity
    // 2) in the simulation service (us) issue a subscribe to the simulation engine
```

```
looking for
    //     an instance of that simulation entity. We use the Entity.State.Name for the
match so it must be
    //    exactly the same. See SimulationTutorial2 for the creation process
    // 3) Listen for a notification telling us the entity is available
    // 4) cache reference to entity and communicate with it issuing low level commands.

    _simEngine = simengine.SimulationEngine.GlobalInstancePort;
    _notificationTarget = new simengine.SimulationEnginePort();

    if (_state == null)
        CreateDefaultState();

    // PartnerType.Service is the entity instance name.
    _simEngine.Subscribe(ServiceInfo.PartnerList, _notificationTarget);

    // dont start listening to DSSP operations, other than drop, until notification of
entity
    Activate(new Interleave(
        new TeardownReceiverGroup
        (
            Arbiter.Receive<simengine.InsertSimulationEntity>(false,  _notificationTarget,
InsertEntityNotificationHandlerFirstTime),
            Arbiter.Receive<DsspDefaultDrop>(false, _mainPort, DefaultDropHandler)
        ),
        new ExclusiveReceiverGroup(),
        new ConcurrentReceiverGroup()
    ));
}

void CreateDefaultState()
{
    _state = new diffdrive.DriveDifferentialTwoWheelState();
    _state.LeftWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
    _state.RightWheel = new Microsoft.Robotics.Services.Motor.Proxy.WheeledMotorState();
    _state.LeftWheel.MotorState                                =                                new
Microsoft.Robotics.Services.Motor.Proxy.MotorState();
    _state.RightWheel.MotorState                                =                                new
Microsoft.Robotics.Services.Motor.Proxy.MotorState();
}

void InsertEntityNotificationHandlerFirstTime(simengine.InsertSimulationEntity ins)
{
    InsertEntityNotificationHandler(ins);

    base.Start();

    // Listen on the main port for requests and call the appropriate handler.
    MainPortInterleave.CombineWith(
        new Interleave(
            new TeardownReceiverGroup(),
            new ExclusiveReceiverGroup(
```

```
                Arbiter.Receive<simengine.InsertSimulationEntity>(true,
_notificationTarget, InsertEntityNotificationHandler),
                Arbiter.Receive<simengine.DeleteSimulationEntity>(true,
_notificationTarget, DeleteEntityNotificationHandler)
            ),
            new ConcurrentReceiverGroup()
        )
    );
}

void InsertEntityNotificationHandler(simengine.InsertSimulationEntity ins)
{
    _entity = (simengine.DifferentialDriveEntity)ins.Body;
    _entity.ServiceContract = Contract.Identifier;

    // create default state based on the physics entity
    if(_entity.ChassisShape != null)
        _state.DistanceBetweenWheels = _entity.ChassisShape.BoxState.Dimensions.X;

    _state.LeftWheel.MotorState.PowerScalingFactor = _entity.MotorTorqueScaling;
    _state.RightWheel.MotorState.PowerScalingFactor = _entity.MotorTorqueScaling;
}
```

초기화 하는데 있어서 핵심적인 코드 영역은 아래와 같습니다.

1. Drop 오퍼레이션 또는 Entity Notification에 대해 한번 기다리도록 간단한 interleave를 활성화합니다. InsertEntityNotificationHandlerFirstTime 핸들러가 단 한번 사용된 것에 유의하시기 바랍니다.
2. Notification 헨들러가 한번 사용됨에 있어서, 엔터티 인스턴스와 서비스 상태를 구성하기 위해서 정상적인 Notification 헨들러를 호출합니다.
3. 베이스 클래스인 Start 메소드가 호출되며 서비스가 해당 오퍼레이션 헨들러를 활성화하고 서비스 디렉토리에 추가합니다.
4. 엔터티들이 동시 삭제 또는 추가되는 동시성 상황으로부터 오퍼레이션을 보호하기 위해, 엔터티 Notification을 수신하는 단순한 Interleave를 메인 포트 Interleave에 결합시킵니다.

Step 5: 튜토리얼 실행

코드를 컴파일 하고 실행시키면 윈도우 창과 “Simple Dashboard” 이름의 윈도우 창이 실행되는 것을 확인하실 수 있습니다.

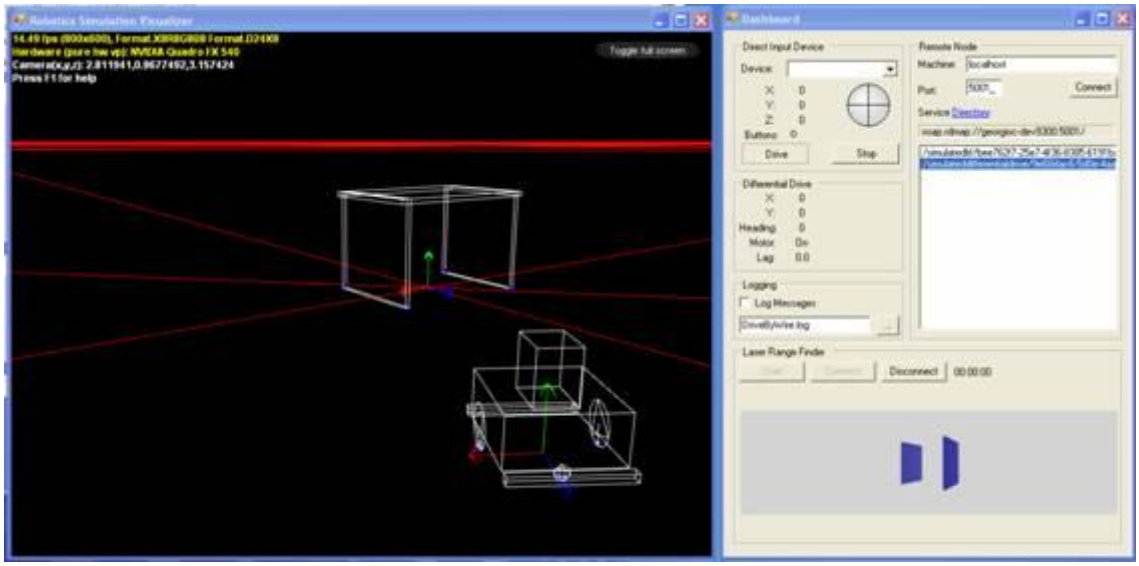


그림 7 - SimpleDashboard에 대한 물리적 표현 화면

Simple Dashboard는 별도의 서비스 프로그램으로서 매니페스트에서 별도의 서비스로 등록되어 있습니다. 따라서 Simple Dashboard 서비스는 별도로 실행 가능한 프로그램이며, 이번 튜토리얼에서는 시뮬레이션 로봇을 제어하기 위해 사용되었습니다. Simple Dashboard 프로그램을 통해 네트워크로 로봇에 접속할 수 있으며, 로봇에 주행 명령을 전송할 수 있습니다. Drive 기반 주행 명령에 대해서는 로보틱스 튜토리얼을 참조하시기 바랍니다. 그리고 Simple Dashboard에서는 레이저 파인더의 값을 읽어 와서 화면에 표시해 주는 기능도 지원합니다.

로봇에 접속하기 위해 Simple Dashboard 창에서 아래의 명령을 수행하시기 바랍니다.

1. Machine 입력창에 localhost를 입력하고 Connect 버튼을 클릭합니다.
2. Connect 버튼 클릭 후에 리스트 박스 창에 2개의 서비스 목록을 확인할 수 있습니다.
3. 화면 하단의 Laser Range Finder 부분에서 Connect 버튼을 클릭합니다.

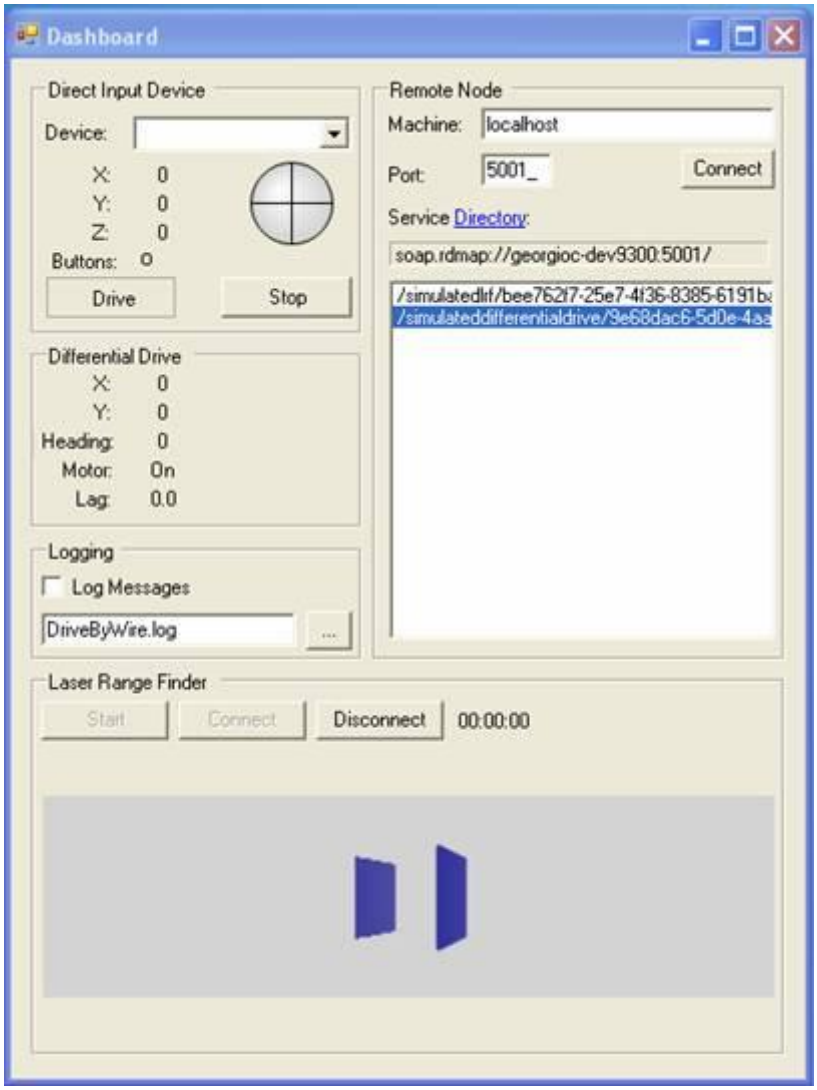


그림 8 - Simple Dashboard

Simple Dashboard를 이용해 로봇을 제어할 수 있으며 아래와 같이 수행합니다.

1. 화면 우측 상단의 서비스 목록중에서 /simulateddifferentialdrive/로 시작하는 항목을 더블 클릭합니다.
2. 좌측의 버튼 중에서 Drive 버튼을 클릭합니다.
3. Drive 버튼 위에 있는 원형 안에 십자 모양이 그려져 있는 부분을 마우스로 클릭하여 움직입니다. 이 도형은 컨트롤러를 대신하여 로봇을 조정하도록 합니다.
4. 만약 조이스틱이나 컨트롤러가 부착되어 있다면, Direct Input Device 항목에 표시되며, 연결된 디바이스를 통해서도 로봇을 제어할 수 있습니다.

시뮬레이션 튜토리얼 3 (C#) - 상태 정보와 매니페스트를 이용한 시뮬레이션 생성

이번 튜토리얼에서는 시뮬레이션 엔진의 상태를 파일로 저장하고 시뮬레이션 구동시 이러한 상태 파일부터 시작정보를 읽어서 초기 값을 설정하는 시나리오를 설명합니다. 이번 시나리오는 별도의 샘플 소스코드를 필요로 하지 않으며, 기존의 튜토리얼을 활용하여 상태 정보를 저장하고 활용하는 과정을 소개합니다.

시작하기

이번 튜토리얼을 시작하기 위해서는 시물레이션 튜토리얼 1번과 2번을 실행한 후 실행중인 시뮬레이션 상태를 파일로 저장하는 것이 필요합니다. 상태를 저장하는 방법은 아래의 Step1을 참고하기 바랍니다.

Step 1: 시뮬레이션 실행 상태를 파일로 저장하기

윈도우 시작 메뉴에서 MSRS Command 창을 실행시키고 아래의 시뮬레이션 튜토리얼 1번을 실행시키는 명령어를 수행합니다.

```
dsshost.exe /port:50000 /tcpport:50001
/manifest:"samplesWconfigWSimulationTutorial1.manifest.xml"
```

웹브라우저를 실행한 후 아래의 주소로 접속하여 서비스가 정상적으로 실행 중 인지를 체크합니다.

http://localhost:50000/simulationengine

연결 후에는 아래의 결과와 같은 페이지를 볼 수 있습니다.

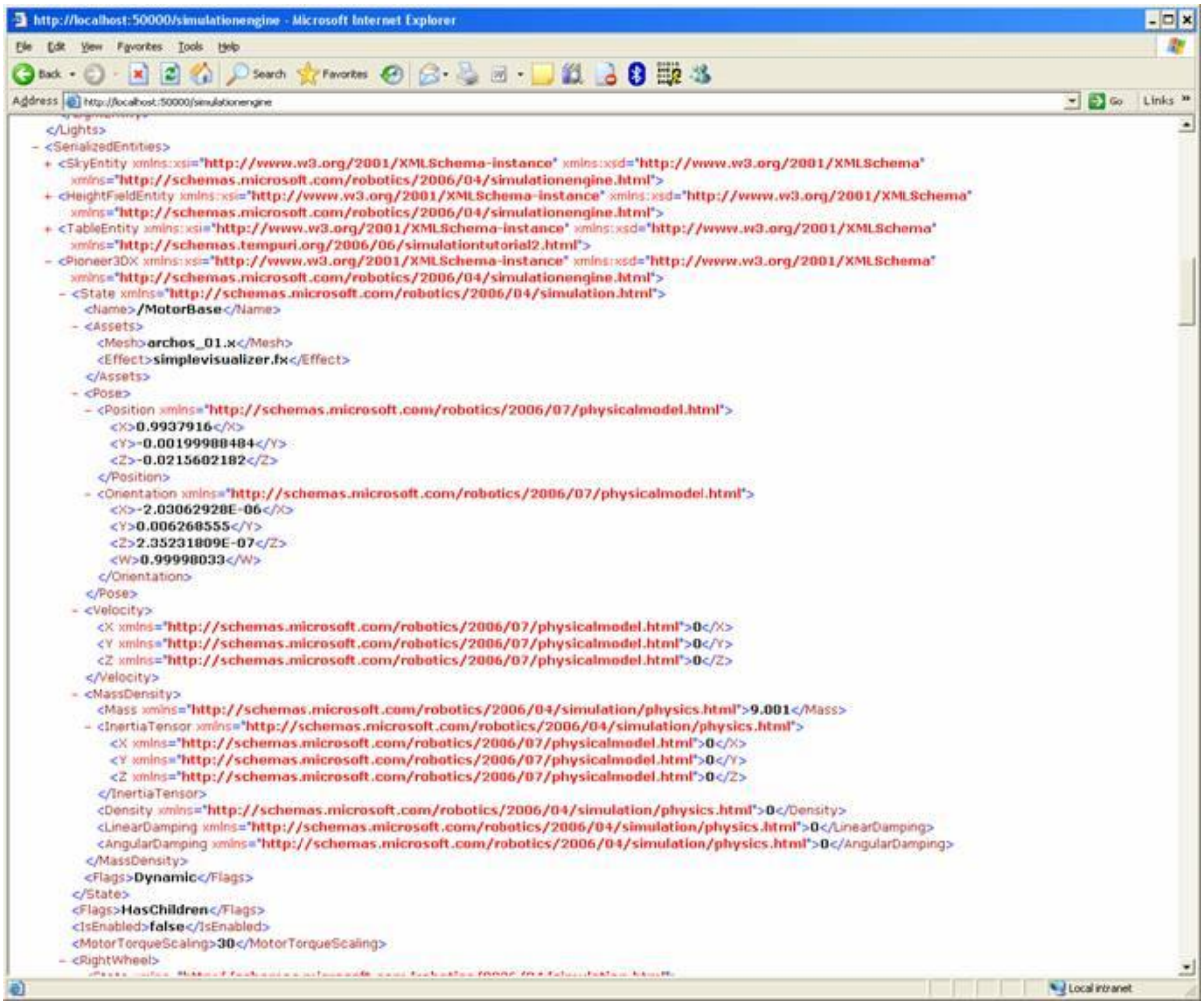


그림 1 - Serialized XML 형태로 표현된 시뮬레이션 상태

Serialized XML 형태로 표현된 시뮬레이션 엔진의 상태는 시뮬레이션 환경에 있는 모든 엔터티들의 현재 속성을 보여줍니다. 스크롤 다운한 후 SerializedEntities XML 노드를 확인해 보면 아래와 같습니다.

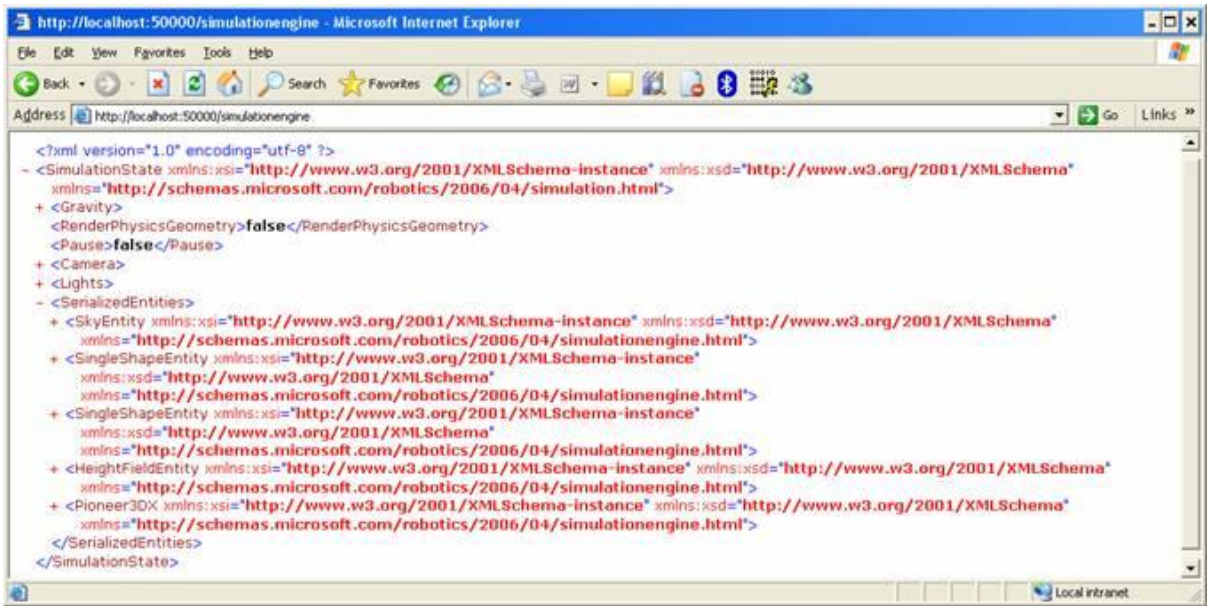


그림 2 - SerializedEntities 노드

첫번째로 이동 가능한 엔터티는 SingleShapeEntity이며, 이 엔터티의 시리얼라이즈된 상태값은 첫번째 시뮬레이션 튜토리얼 (SimulationTutorial1.cs)에 포함되어 있는 아래의 코드 부분에서 설정한 값을 보여줍니다.

```
void AddBox(Vector3 position)
{
    Vector3 dimensions =
        new Vector3(0.2f, 0.2f, 0.2f); // meters

    // create simple movable entity, with a single shape
    SingleShapeEntity box = new SingleShapeEntity(
        new BoxShape(
            new BoxShapeProperties(
            100, // mass in kilograms.
            new Pose(), // relative pose
            dimensions)), // dimensions
        position);

    box.State.MassDensity.Mass = 0;
    box.State.MassDensity.Density = 0;

    // Name the entity. All entities must have unique names
    box.State.Name = "box";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(box);
}
```

시뮬레이션 엔진 서비스의 웹 주소에 대해 HTTP GET 형태로 값을 읽어 보면, 시뮬레이션 엔진 서비스의 내부에 값이 유지되는 상태값을 확인해 볼 수 있습니다.

웹브라우저에 보이는 내용을 MSRS 설치 폴더 밑에 Wstore 폴더 안에 SimulationEngineState.xml 로 저장해 놓습니다.

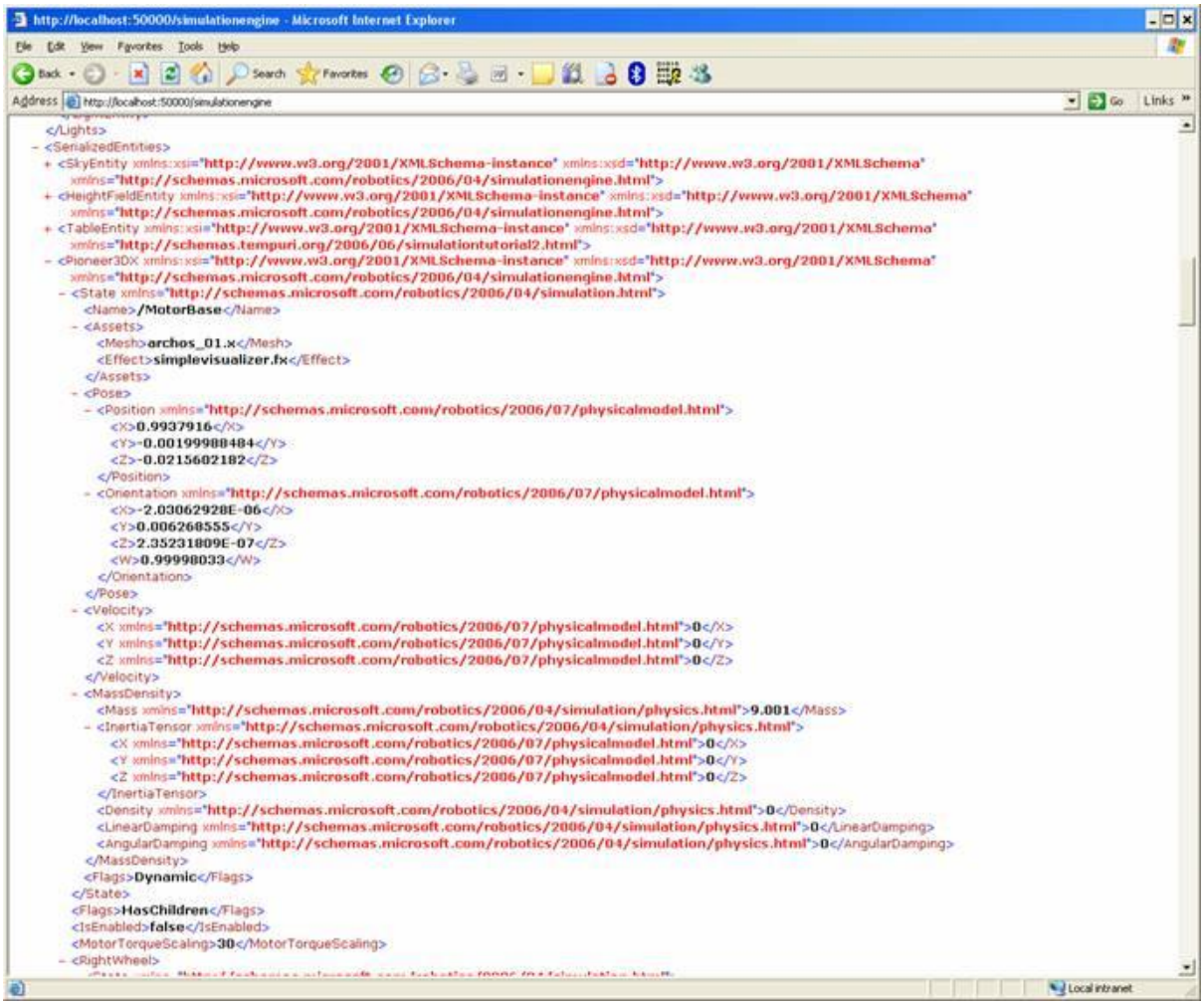


그림 3 - XML 파일로 저장되어 있는 시뮬레이션 상태값

이제는 실행차에서 Ctrl+C를 눌러 노드의 실행을 중지시키고 아래의 명령어를 실행시킵니다.

1. Samples 아래에 새로운 폴더를 생성합니다.

```
cd Samples
md MySimulationTutorial3
  cd MySimulationTutorial3
```

2. 위에서 Store 폴더 안에 저장해 놓은 파일을 생성해 놓은 폴더 안에 복사해 놓습니다.

```
copy ..W..WstoreWSimulationEngineState.xml MySim3.SimulationState.xml
```

Step 2: 시뮬레이션 상태 파일을 결합하기

이제 두번째 시뮬레이션 튜토리얼 예제를 실행합니다. 위의 Step1 과정과 동일하게 웹브라우저로 접속 후 실행상태를 XML 파일로 저장해 놓습니다.

두번째 시뮬레이션 튜토리얼은 아래와 같이 실행시킵니다.

```
dsshost.exe /p:50000 /t:50001 /manifest:"samplesWconfigWSimulationTutorial2.manifest.xml"
```

실행 상태는 웹브라우저를 통해서 아래의 주소로 접속하여 확인입니다.

http://localhost:50000/simulationengine

이제 실행 상태를 중지하고 웹브라우저의 내용을 XML 파일로 저장합니다. 파일은 위에서 만든 폴더에 생성하며, 파일 이름은 MySim3.SimulationState.XML 파일로 저장합니다.

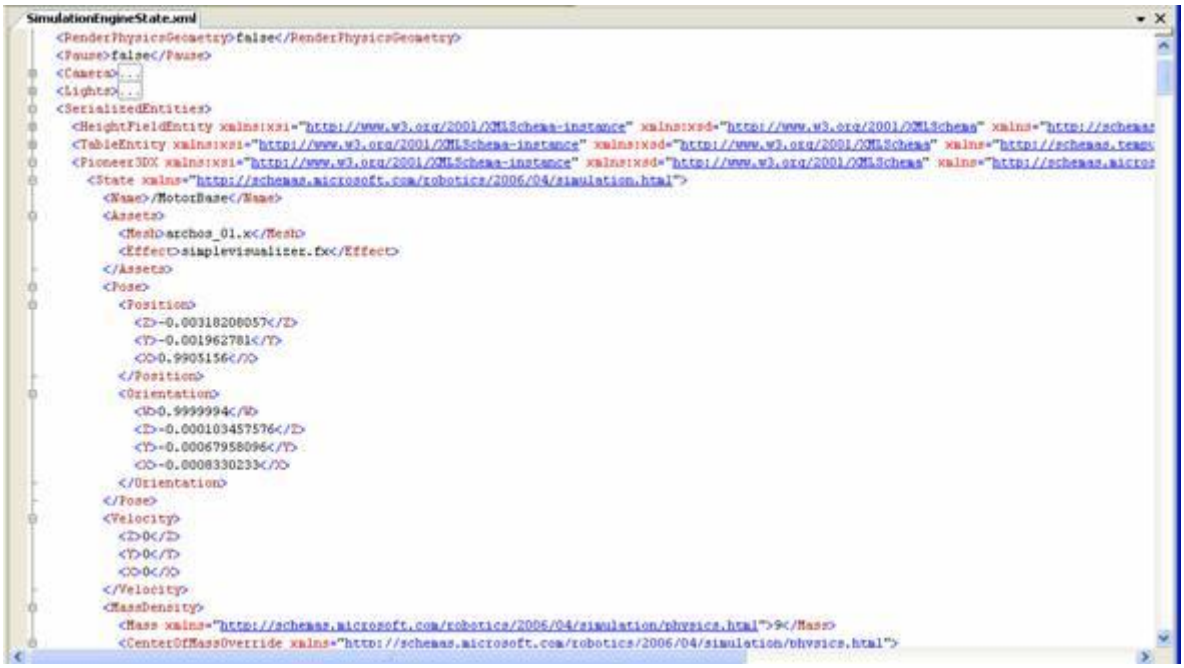


그림 4 - XML 형태로 저장된 파일의 내용 중 Pioneer3DX 부분

이제 나중에 저장된 XML 파일과 Step 1에서 저장된 XML 파일을 Visual Studio를 통해 파일을 엽니다. 그리고 나중에 저장된 파일 중에서 Pioneer3DX와 이에 연관되어 있는 LaserRangeFinderEntity, BumperArrayEntity, CameraEntity 관련 XML 노드들을 복사하여, 첫번째 XML 파일인 SimulationEngineState.XML의 <SerializedEntities> 노드 안에 복사해 놓습니다. Pioneer3DX는 아래와 같이 시작되며, LaserRangeFinderEntity와 나머지 항목들도 복사해 놓고 파일을 저장합니다.

```
<Pioneer3DX                                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.microsoft.com/robotics/2006/04/simulationengine.html">
```

Step 3: 저장된 XML 파일로 서비스 실행하기

이제 SimulationTutorial3.manifest.xml 파일을 새롭게 생성하여, 위의 단계에서 생성한 XML 파일을 이용해서 서비스를 구동합니다.

MSRS 설치 폴더 밑에 SamplesWConfig 폴더에는 SimulationTutorial3.manifest.xml 이름의 파일이 샘플로 이미 생성되어 있으며, 해당 파일의 내용은 아래와 같습니다.

```
<?xml version="1.0" ?>
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns:simcommon="http://schemas.microsoft.com/robotics/2006/04/simulation.html"
    >
  <!--Simulation Tutorial 3 manifest. Uses simulation engine state file as configuration
```

```
for simulation engine -->
  <CreateServiceList>

    <!--Start simulation engine service-->
    <ServiceRecordType>

<dssp:Contract>http://schemas.microsoft.com/robotics/2006/04/simulationengine.html</dssp:C
ontract>
      <dssp:PartnerList>
        <dssp:Partner>
          <dssp:Service>SimulationTutorial3.SimulationEngineState.xml</dssp:Service>
          <dssp:Name>dssp:StateService</dssp:Name>
        </dssp:Partner>
      </dssp:PartnerList>
    </ServiceRecordType>

    <!-- Start simulated motor service-->
    <ServiceRecordType>

<dssp:Contract>http://schemas.microsoft.com/robotics/simulation/services/2006/05/simulated
differentialdrive.html</dssp:Contract>
      <dssp:PartnerList>
        <dssp:Partner>
          <!--The partner name must match the entity name-->
          <dssp:Service>http://localhost/P3DXMotorBase</dssp:Service>
          <dssp:Name>simcommon:Entity</dssp:Name>
        </dssp:Partner>
      </dssp:PartnerList>
    </ServiceRecordType>

    <!-- Start simulated laser range finder service-->
    <ServiceRecordType>

<dssp:Contract>http://schemas.microsoft.com/robotics/simulation/services/2006/05/simulated
lrf.html</dssp:Contract>
      <dssp:PartnerList>
        <dssp:Partner>
          <!--The partner name must match the entity name-->
          <dssp:Service>http://localhost/P3DXLaserRangeFinder</dssp:Service>
          <dssp:Name>simcommon:Entity</dssp:Name>
        </dssp:Partner>
      </dssp:PartnerList>
    </ServiceRecordType>

    <!-- Start simulated bumber service-->
    <ServiceRecordType>

<dssp:Contract>http://schemas.microsoft.com/robotics/simulation/services/2006/05/simulated
bumper.html</dssp:Contract>
      <dssp:PartnerList>
        <dssp:Partner>
          <!--The partner name must match the entity name-->
```

```
          <dssp:Service>http://localhost/P3DXBumpers</dssp:Service>
          <dssp:Name>simcommon:Entity</dssp:Name>
        </dssp:Partner>
      </dssp:PartnerList>
    </ServiceRecordType>
  </CreateServiceList>

</Manifest>
```

위의 예제에서는 동일한 폴더에 존재하는 SimulationEngineState.xml 파일을 참조하도록 되어 있습니다.

Step 4: 실행하기

이제 아래의 명령어를 MSRS 실행창에서 실행시킵니다.

```
Dsshost.exe                    /port:50000                    /tcpport:50001
/manifest:" samplesWconfigWSimulationTutorial3.manifest.xml"
```

또한 SimpleDashboard 서비스를 아래와 같이 실행시킨 후 SimpleDashboard에서 해당 Pioneer3DX 로봇이 정상적으로 제어되는지를 확인합니다.

```
dsshost.exe /p:51000 /t:51001 /m:"samplesWconfigWSimpleDashboard.manifest.xml"
```

시뮬레이션 튜토리얼 4 (C#) - 조인트와 다관절 Arm 구현

이번 튜토리얼에서는 시뮬레이션 상에서 6자유도를 가지는 조인트(Joint)와 다관절 Arm (Articulated Arms)을 구현하는 예제를 소개합니다.

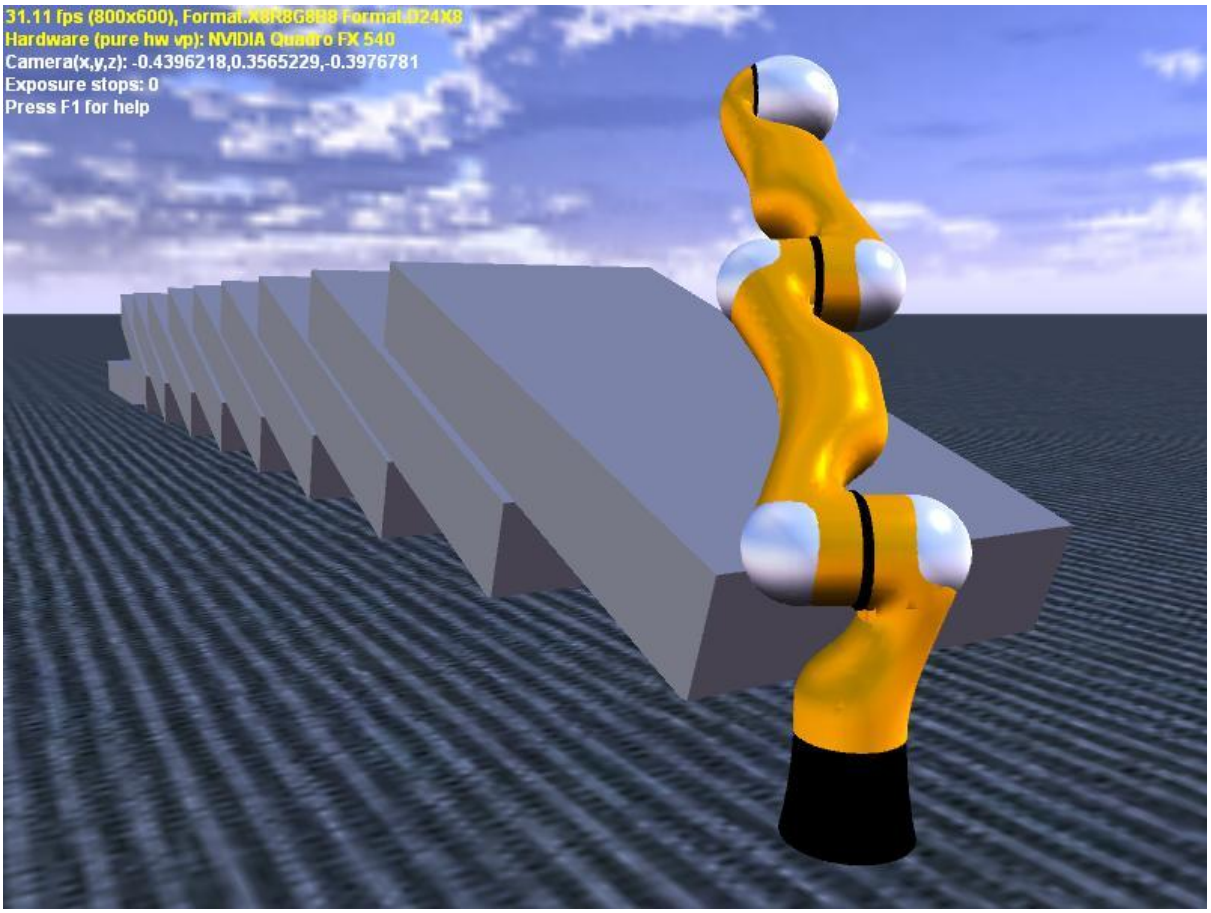


그림 1 - 다관절 Arm을 시뮬레이션에서 구현한 예제

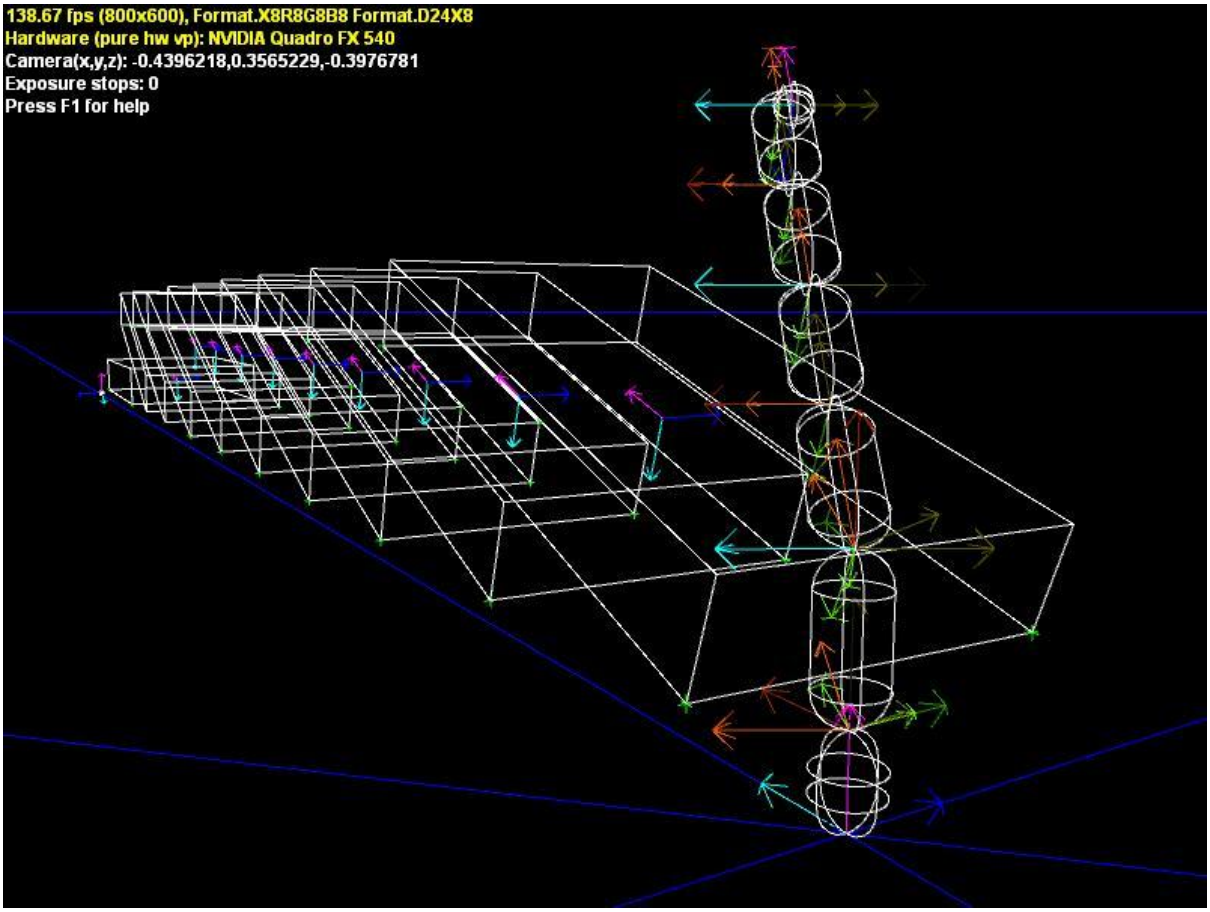


그림 2 - 다관절 Arm의 물리적 랜더링 예제

시작하기

이번 시뮬레이션 튜토리얼은 해당 샘플 코드가 이미 존재하며 samples₩SimulationTutorials₩Tutorial4 폴더를 참조합니다. Simulationtutorial4.csproj 파일을 비주얼 스튜디오를 이용해서 로드합니다.

프로젝트 참조

이번 시물레이션 프로그램이 실행되기 위해서는 두번째 시뮬레이션 프로젝트에 추가적으로 아래와 같이 3개의 Proxy dll 파일을 참조에 추가해야 합니다.

- RoboticsCommon.Proxy - 다관절 Arm과 같은 서비스에 대한 서비스 인터페이스 정의
- SimulatedLBR3Arm.2006.M07.Proxy - 시뮬레이션 LBR3 Arm 서비스에 대한 프록시 라이브러리
- SimulationCommon.Proxy - 공통 시뮬레이션 타입에 대한 프록시 라이브러리. 결과값을 시뮬레이션 Arm에 보내기 위해 사용됩니다.

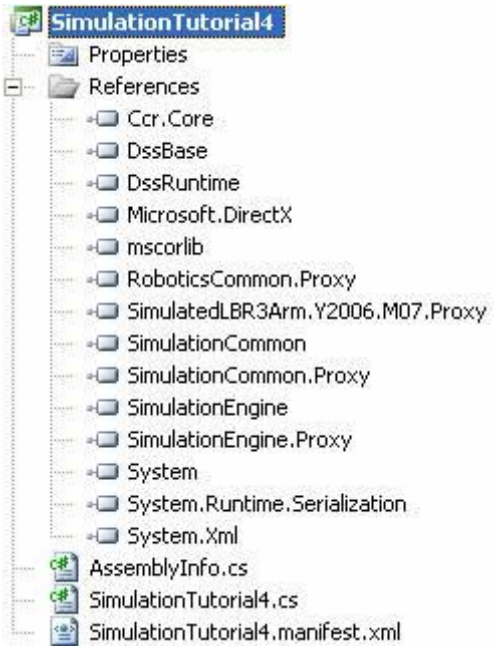


그림 3 - 튜토리얼4의 참조 구성

아래의 using문이 파일 상단에 추가되어야 합니다.

```
using arm = Microsoft.Robotics.Services.ArticulatedArm.Proxy;

using Microsoft.Robotics.PhysicalModel;
using physicalmodelproxy = Microsoft.Robotics.PhysicalModel.Proxy;
```

PhysicalModel 네임스페이스의 타입 적용

Microsoft.Robotics.PhysicalModel 네임스페이스에서는 로봇의 물리 특성을 기술하는 데 필요한 타입들이 정의됩니다. 이러한 타입들은 시뮬레이션에서 엔터티와 시뮬레이션 서비스들을 기술하기 위해 사용되기 때문에, Microsoft.Robotics.Simulation.Physics의 네임스페이스와 타입들이 PhysicalModel 네임스페이스로 단일화 되었습니다. 하지만, Vector3와 Quaternion과 같은 공통 타입들이 정의되는 DirectX에 대한 런타임에서의 의존성을 피하기 위하여, 새로운 기본 타입들이 새로운 네임스페이스에 이러한 이름으로 정의되었습니다.

이러한 새로운 모델 타입을 사용하는 시뮬레이션 서비스 또는 실제 서비스에 연결되는 서비스들에 있어서는 추가적으로 복잡한 작업이 요구되지는 않습니다. 단지 일반적으로 메시지를 전달하드시 원격 서비스에 대한 프록시에 링크를 거는 수준입니다. Quaternion이 포함되는 연산을 수행하는 서비스나 시뮬레이션 서비스들에 있어서는 이러한 작업들이 큰 문제가 되지는 않으며, 아래와 같은 주된 이유를 가집니다.

프록시에서는 정적 루틴을 넘어서 변경이 안되기 때문에, Vector와 Quaternion 연산에 대해서 정적 Helper 메소드가 physicalmodelproxy 네임스페이스에서는 더 이상 유효하지가 않습니다. 이것은 개발자가 RoboticsCommon.dll과 RoboticsCommonProxy.dll 파일 둘 다를 참조에 포함해야 한다는 것을 의미합니다. 이름들이 같은 프록시 네임스페이스에 대해서 이름의 혼동을 피하기 위하여 Using 문장을 기술할 때, 별칭(Alias)을 사용하여 축약된 형태를 사용하는 것을 권장합니다.

이러한 이슈를 다룬 좋은 예제는 SimulatedLBR3Arm 서비스이며 SamplesWSimulationWArticulatedArmsWLBR3 폴더 안에 소스 코드가 있습니다.

Step 1: 서비스를 생성하고 시작하기

서비스 시작

이번 튜토리얼은 두번째 튜토리얼과 매우 유사한 방식으로 시작되며, PopulateWorld 메소드를 호출하고 몇 개의 엔터티를 추가합니다.

```
protected override void Start()
{
    base.Start();
    // Orient sim camera view point
    SetupCamera();
    // Add objects (entities) in our simulated world
    PopulateWorld();
}

private void SetupCamera()
{
    // Set up initial view
    CameraView view = new CameraView();
    view.EyePosition = new Vector3(1.8f, 0.598689f, -1.28f);
    view.LookAtPoint = new Vector3(0, 0, 0.2830455f);
    SimulationEngine.GlobalInstancePort.Update(view);
}

private void PopulateWorld()
{
    AddSky();
    AddGround();
    AddDomino(new Vector3(0.3f, 0, 0.3f), 10);
    SpawnIterator(AddArticulatedArm);
}
```

시뮬레이션 Arm 서비스와 엔터티

AddArticulatedArm 메소드는 산업용 다관절 로봇 Arm에 대한 시뮬레이션 엔터티들과 엔터티들을 제어하기 위한 관련 서비스들을 생성합니다. 해당 메소드가 다중의 비동기 연산을 순차적으로 수행해야 하기 때문에 Iterater 방식이 적용되었습니다.

```
IEnumerator<ITask> AddArticulatedArm()
{
    Vector3 position = new Vector3(0, 0, 0);

    // Create an instance of our custom arm entity.
    // Source code for this entity can be found under
    // SamplesWSimulationWEntitiesWEntities.cs
    KukaLBR3Entity entity = new KukaLBR3Entity(position);

    // Name the entity
    entity.State.Name = "LBR3Arm";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(entity);
```

```
    // create simulated arm service
    DsspResponsePort<CreateResponse> resultPort = CreateService(
        Microsoft.Robotics.Services.Simulation.LBR3Arm.Proxy.Contract.Identifier,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner("http://localhost/"   +
entity.State.Name));

    // asynchronously handle service creation result.
    yield return Arbiter.Choice(resultPort,
        delegate(CreateResponse rsp)
        {
            _armServicePort = ServiceForwarder<arm.ArticulatedArmOperations>(rsp.Service);
        },
        delegate(Fault fault)
        {
            LogError(fault);
        });

    if (_armServicePort == null)
        yield break;

    // we re-issue the get until we get a response with a fully initialized state
    do
    {
        yield return Arbiter.Receive(false, TimeoutPort(1000), delegate(DateTime signal)
{ });

        yield return Arbiter.Choice(
            _armServicePort.Get(new GetRequestType()),
            delegate(arm.ArticulatedArmState state)
            {
                _cachedArmState = state;
            },
            delegate(Fault f)
            {
                LogError(f);
            });

        // exit on error
        if (_cachedArmState == null)
            yield break;
    } while (_cachedArmState.Joints == null);

    // Start a timer that will move the arm in a loop
    // Comment out the line below if you want to control
    // the arm through SimpleDashboard
    //Spawn<DateTime>(DateTime.Now, MoveArm);
}
```

두번째 튜토리얼에서의 AddModularRobot과 유사하게 엔터티가 생성되고 시뮬레이션 엔진과 엔터티에 대한 서비스에 더해져서 추가됩니다. 그리고 아래의 작업들이 추가적으로 적용됩니다.

1. CreateService 메소드에 대한 결과가 비동기적으로 처리되며, 신규 서비스와의 통신을 위해 포트가 생성됩니다.
2. GET DSSP 연산이 해당 상태를 조회합니다. 다관절 Arm의 상태 정의는 각각의 Arm들을 연결하는 조인트들을 기술하는 조인트 인스턴스의 목록을 포함합니다.
3. MoveArm 메소드가 증식되면서 호출됩니다.

MoveArm 메소드는 시뮬레이션 Arm 서비스에 Arm들을 연결하는 다양한 조인트를 제어하기 위한 요청 데이터를 생성합니다. 시뮬레이션 서비스는 다관절 Arm 서비스의 추상화된 인터페이스를 사용하기 때문에, 이러한 메소드는 아래와 같은 서비스 구현으로 동작합니다.

```
void MoveArm(DateTime signal)
{
    _angleInRadians += 0.005f;
    float angle = (float)Math.Sin(_angleInRadians);

    // Create set pose operation. Notice we have specified
    // null for the response port to eliminate a roundtrip
    arm.SetJointTargetPose setPose = new arm.SetJointTargetPose(
        new arm.SetJointTargetPoseRequest(),
        null);

    // specify by name, which joint to orient
    setPose.Body.JointName = _cachedArmState.Joints[0].State.Name;
    setPose.Body.TargetOrientation = new physicalmodelproxy.AxisAngle(
        new physicalmodelproxy.Vector3(1, 0, 0), angle);

    // issue request to arm service for joint0 move.
    _armServicePort.Post(setPose);

    // now move other joints.
    setPose.Body.JointName = _cachedArmState.Joints[1].State.Name;
    _armServicePort.Post(setPose);
    setPose.Body.JointName = _cachedArmState.Joints[2].State.Name;
    _armServicePort.Post(setPose);
    setPose.Body.JointName = _cachedArmState.Joints[3].State.Name;
    _armServicePort.Post(setPose);
    setPose.Body.JointName = _cachedArmState.Joints[4].State.Name;
    _armServicePort.Post(setPose);

    // re- issue timer request so we wake up again
    Activate(Arbiter.Receive(false, TimeoutPort(15), MoveArm));
}
```

메소드는 SetJointTargetPose 데이터를 15 밀리초 단위로 생성하며, 점진적으로 조인트를 해당 축 주위로 이동시킵니다. 시뮬레이션 LBR3 로봇 Arm에 대해서는 모든 조인트들이 단일 자유도를 가지도록 설정됩니다.

Step 2: 조인트 구성

다음 단계로 진행하기 전에, 조인트들이 물리엔진에서 시뮬레이션 상으로 어떻게 기술되는지에 대해 살펴 보도록 합니다. 6자유도와 같은 세부적인 기술자료는 AGEIA PhysX의 기술 문서를 참고하길 바라며, 조인트 속성의 데이터 타입은 SamplesWCommonWPhysicalModel.cs 파일에 정의되어

있습니다.

```
/// <summary>
/// Joint properties
/// </summary>
[DataContract]
public class JointProperties
{
    /// <summary>
    /// Joint name. Must be unique for all joints between an entity pair
    /// </summary>
    [DataMember]
    [Description("The descriptive identifier for the joint.")]
    public string Name;

    /// <summary>
    /// Pair of entities joined through this joint
    /// </summary>
    [DataMember]
    [Description("The pair of entities connected through this joint.")]
    public EntityJointConnector[] Connectors = new EntityJointConnector[2];

    /// <summary>
    /// Maximum force supported by the joint
    /// </summary>
    [DataMember]
    [Description("The maximum force supported by the joint.")]
    public float MaximumForce;

    /// <summary>
    /// Maximum torque supported by the joint
    /// </summary>
    [DataMember]
    [Description("The maximum torque supported by the joint.")]
    public float MaximumTorque;

    /// <summary>
    /// Enables collision modelling between entities coupled by the joint
    /// </summary>
    [DataMember]
    [Description("Enables collision between entities couples by the joint.")]
    public bool EnableCollisions;

    /// <summary>
    /// Underlying physics mechanism to compensate joint simulation errors
    /// </summary>
    [DataMember]
    [Description("Underlying physics mechanism to compensate joint simulation errors.")]
    public JointProjectionProperties Projection;

    /// <summary>
    /// If set, defines a joint with translation/linear position drives
```

```
    /// </summary>
    [DataMember]
    [Description("Identifies if the joint supports translation/linear position drives.")]
    public JointLinearProperties Linear;

    /// <summary>
    /// If set, defines a joint with angular drives.
    /// </summary>
    [DataMember]
    [Description("Identifies if the joint supports angular drives.")]
    public JointAngularProperties Angular;


    // ... Constructors omitted for clarity
}
```

시뮬레이션 엔진은 오진 한가지 타입의 조인트만을 노출시키며, 이러하 조인트는 6자유도 조인트로 알려져 있습니다. 조인트는 2개의 물리 엔터티를 연결시키며, 상대 엔터티에 대한 동작을 제한시킬 수 있습니다. 조인트의 기본 요소는 아래와 같습니다.

- 자유도는 Unlock, Lock 또는 Limited 상태를 가집니다.
- 엔터티들은 로컬 포인트를 기준으로 연결됩니다.
- 각각의 엔터티에 대해 기준 축이 정의됩니다.
- 조인트 법선 (normal)은 해당 조인트가 향하는 방향을 보여줍니다.
- Unlock 자유도에 대해 옵션사항으로 드라이브와 스프링 메커니즘을 적용할 수 있습니다.

구형 조인트 예제

구형 조인트를 구성하기 위해서는 선형 자유도를 고정시키고 (또는 JointProperties.Linear 관련 코드를 전혀 기술하지 않음) 각 자유도 (Angular degree of freedom) 항목을 해제시킵니다.

```
void MoveArm(DateTime signal)
{
    _angleInRadians += 0.005f;
    float angle = (float)Math.Sin(_angleInRadians);

    // Create set pose operation. Notice we have specified
    // null for the response port to eliminate a roundtrip
    arm.SetJointTargetPose setPose = new arm.SetJointTargetPose(
        new arm.SetJointTargetPoseRequest(),
        null);

    // specify by name, which joint to orient
    setPose.Body.JointName = _cachedArmState.Joints[0].State.Name;
    setPose.Body.TargetOrientation = new physicalmodelproxy.AxisAngle(
        new physicalmodelproxy.Vector3(1, 0, 0), angle);

    // issue request to arm service for joint0 move.
    _armServicePort.Post(setPose);

    // now move other joints.
    setPose.Body.JointName = _cachedArmState.Joints[1].State.Name;
```

```
    _armServicePort.Post(setPose);
    setPose.Body.JointName = _cachedArmState.Joints[2].State.Name;
    _armServicePort.Post(setPose);
    setPose.Body.JointName = _cachedArmState.Joints[3].State.Name;
    _armServicePort.Post(setPose);
    setPose.Body.JointName = _cachedArmState.Joints[4].State.Name;
    _armServicePort.Post(setPose);

    // re- issue timer request so we wake up again
    Activate(Arbiter.Receive(false, TimeoutPort(15), MoveArm));
}

JointAngularProperties angular = new JointAngularProperties();
angular.TwistMode = JointDoFMode.Free;
angular.Swing1Mode = JointDoFMode.Free;
angular.Swing2Mode = JointDoFMode.Free;

angular.TwistDrive = new JointDriveProperties(JointDriveMode.Position,
    new SpringProperties(500, 10, 0), 1000);
angular.SwingDrive = new JointDriveProperties(JointDriveMode.Position,
new SpringProperties(500, 10, 0), 1000);


JointProperties sphericalJoint = new JointProperties(angular, null, null);
```

위의 코드에서는 각 자유도 항목들이 해제되었으며, Twist 자유도에 대해서만 드라이브 메커니즘에 적용되었습니다. 이제 조인트는 MoveArm에서 보여지는 SetJointtargetPose를 통해서 제어됩니다. SetJointTargetVelocity를 이용해 조인트 드라이브를 제어하기 위해서는 다음과 같이 드라이브 모드를 설정해야 합니다.

```
angular.TwistDrive = new JointDriveProperties(JointDriveMode.Velocity,
    new SpringProperties(500, 10, 0), 1000);
```

Step 3: 엔터티 구현에 대한 이해와 조인트 정의

MSRS에서 지원되는 모든 엔터티에 대해서는 에제의 형태로 소스 코드가 제공됩니다. samplesWsimulationWentitiesWentities.cs 에는 내장되어 있는 엔터티의 구현 사례가 제공되어 있으며, 컴파일 되어 SimulationEngine.dll 안에 포함되어 있습니다. 이번 튜토리얼은 KukaLBR3Entity에 초점을 맞추고 있으며, 7개의 Arm 링크와 6개의 조인트로 구성되어 있습니다. 아래의 코드는 전체 코드중의 일부이며, 조인트 리스트와 ArticulatedArmEntity 인스턴스 항목을 참조하기 바랍니다.

```
/// <summary>
/// Models KUKA LBR3 robotic arm
/// </summary>
[DataContract]
[CLSCompliant(true)]
public class KukaLBR3Entity : VisualEntity
{
    const float ARM_MASS = 1f;
    const float ARM_THICKNESS = 0.03f;
    const float ARM_LENGTH = 0.075f;
```

```
    const float DELTA = 0.000f;

    // approximation of the Lbr3 arms for arms 1->5
    // base is considered arm/link 0, and the end effectors are 6 and 7
    float ARM_LENGTH2 = ARM_LENGTH + ARM_THICKNESS * 2;
    float MIDPOINT_Y = ARM_LENGTH / 2f + ARM_THICKNESS;

    const int _jointCount = 7;

    /// <summary>
    /// Number of joints
    /// </summary>
    [DataMember]
    public int JointCount
    {
        get { return _jointCount; }
    }

    List<Joint> _joints = new List<Joint>();
    /// <summary>
    /// Joints
    /// </summary>
    [DataMember]
    public List<Joint> Joints
    {
        get { return _joints; }
        set { _joints = value; }
    }

    List<ArmLinkEntity> _links = new List<ArmLinkEntity>();

    /// <summary>
    /// Default constructor
    /// </summary>
    public KukaLBR3Entity()
    {
    }

    /// <summary>
    /// Initialization constructor
    /// </summary>
    /// <param name="position"></param>
    public KukaLBR3Entity(Vector3 position)
    {
        State.Pose.Position = position;
    }
```

엔터티 초기화

다른 모든 엔터티와 유사하게, KukaLBR3 엔터티는 초기화 루틴 안에서 데이터 구조를 맨 먼저 구성합니다. 그리고 그 다음으로 해당 강체에 대한 물리 인스턴스를 생성하는 Initalized 메소드를

호출합니다. 초기화 루틴은 조인트가 모든 링크 엔터티가 생성되고 나서 조인트들이 생성되어야 하기 때문에 약간 복잡합니다. 조인트들은 사용되기 위해서 인스턴스를 필요로 하기 때문에, 물리엔진에 추가되고 활성화 되어져야 합니다.

엔터티의 초기화 메소드는 엔터티가 어떻게 생성되었는 지에 따라서 두가지 방식으로 구성됩니다.

1. 프로그램에 의한 생성 - 초기화 생성자를 통해 엔터티가 생성되었다면, 메소드는 모든 자식 링크 엔터티들을 자동으로 생성하고 모든 조인트 파라메터를 설정합니다.
2. Deserialization - 만약 엔터티가 Descerialization 되어 있다면, 모든 파라메터는 설정되지만, 초기화 과정은 적당한 물리 객체를 생성하기 위해 호출되어 져야 합니다.

Initialized 메소드는 아래와 같습니다.

```
/// <summary>
/// Initialization
/// </summary>
/// <param name="device"></param>
/// <param name="physicsEngine"></param>
public override void Initialize(xnagrfx.GraphicsDevice device, PhysicsEngine physicsEngine)
{
    try
    {
        InitError = string.Empty;


        // all joints will be essentially of the same type:
        // A single DoF free, around the Joint Axis. A drive will be associated with
        // that DoF. Each joint will have a different joint axis and joint normal
        for (int i = 0; i < _jointCount; i++)
        {
            PhysicsJoint jointInstance = null;
            if (_joints.Count < i + 1)
            {
                JointAngularProperties commonAngular = new JointAngularProperties();
                commonAngular.TwistMode = JointDoFMode.Free;

                commonAngular.TwistDrive = new JointDriveProperties(
                    JointDriveMode.Position,
                    new SpringProperties(500000, 100000, 0),
                    1000000);

                jointInstance = PhysicsJoint.Create(new JointProperties(commonAngular, null, null));
                // joints must be names
                jointInstance.State.Name = "Joint" + i.ToString();
                Joints.Add(jointInstance);
            }
            else
            {
                jointInstance = PhysicsJoint.Create(_joints[i].State);
                _joints[i] = jointInstance;
```

```
            }


        }

        if (_baseLink == null)
        {
            // programmatically build articulated arm
            CreateBase(_joints[0]);
            CreateArm1(_joints[0]);
            CreateArm2(_joints[1]);
            CreateArm3(_joints[2]);
            CreateArm4(_joints[3]);
            CreateArm5(_joints[4]);
            CreateArm6(_joints[5]);
            CreateArm7(_joints[6]);
        }

        // if we were deserialized, _links will be null
        // Rebuilt it from the individual named fields

        if (_links == null || _links.Count == 0)
        {
            _links = new List<ArmLinkEntity>();
            _links.Add(_baseLink);
            _links.Add(_arm1Link);
            _links.Add(_arm2Link);
            _links.Add(_arm3Link);
            _links.Add(_arm4Link);
            _links.Add(_arm5Link);
            _links.Add(_arm6Link);
            _links.Add(_arm7Link);
        }

        // set the physics shape on each child entity
        for (int i = 1; i < (_links.Count - 1); i++)
        {
            // Links 1 to 6 use the same capsule shape to approximate the detailed
            // geometry of the LBR3 arm links. Notice that in the local pose for the
            // shape, we translate it up by half its total height since, by default
            // physics engine sets the center of the shape to be its center of mass

            if (_links[i].CapsuleShape == null)
            {
                _links[i].CapsuleShape = new CapsuleShape(
                    new CapsuleShapeProperties(ARM_MASS,
                    new Pose(new Vector3(0, MIDPOINT_Y, 0)),
                    ARM_THICKNESS,
                    ARM_LENGTH));
            }
        }
```

```
        // initialize will load the graphics mesh
        base.Initialize(device, physicsEngine);

        // initialize children
        foreach (VisualEntity c in _links)
        {
            c.Parent = this;
            c.Initialize(device, physicsEngine);
        }

        // fix the base link so its immovable. Remove this if you plan to have the arm
        // attached through yet a different joint, to some other entity
        _baseLink.PhysicsEntity.IsKinematic = true;

        // for each child arm, set the lower and upper joint
        for (int i = 0; i < _links.Count; i++)
        {
            ArmLinkEntity link = _links[i] as ArmLinkEntity;
            if (i < _jointCount)
            {
                link.UpperJoint = _joints[i];
            }

            // assume children.Count == _joints.Count + 1
            if (i > 0)
            {
                link.LowerJoint = _joints[i - 1];
            }

            // set the entity name in upper connector
            if (link.UpperJoint != null)
                link.UpperJoint.State.Connectors[0].Entity = link;
            if (link.LowerJoint != null)
                link.LowerJoint.State.Connectors[1].Entity = link;
        }

        // second pass inserts joints into the simulation, which initializes and activates
them
        // We have to do this after the joint connectors are all attached to entities

        for (int i = 0; i < _jointCount; i++)
        {
            physicsEngine.InsertJoint((PhysicsJoint)_joints[i]);
        }
    }
    catch (Exception ex)
    {
        HasBeenInitialized = false;
        InitError = ex.ToString();
    }
}
```

조인트들은 PhysicsJoint.Create 문장을 통해 물리엔진에서 명확히 초기화 되어져야 합니다.

시각적인 표현을 위한 3D 설계 데이터의 활용

모든 Arm 형태는 기반 Arm을 제외하고는 동일한 캡슐 모양을 가집니다. 시각화를 위해 3개의 메쉬 파일이 사용되었습니다.

- Lbr3_j0.obj - 기반 Arm을 위해 사용되는 메쉬파일
- Lbr3_j1.obj - link1 부터 link4까지의 Arm을 위해 사용되는 메쉬파일
- Lbr3_j5.obj - link5를 위해 사용되는 메쉬파일
- Lbr3_j6.obj - 구형 link6을 위해 사용되는 메쉬파일

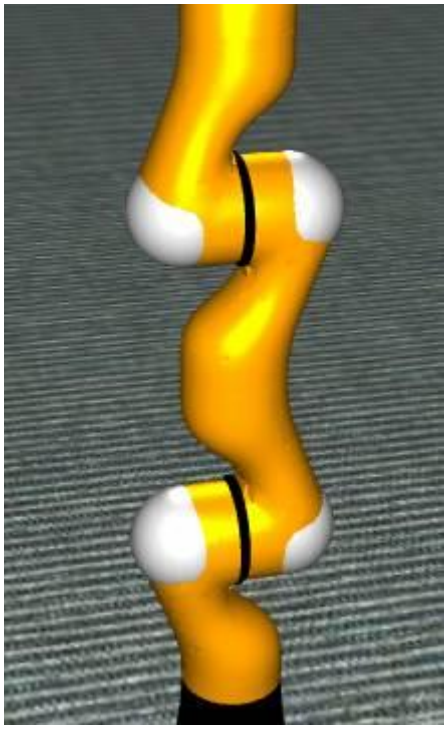


그림 4 - 캡슐 모양

Lbr3_j1.obj 파일은 유사한 링크들의 시각적 표현을 위해 회전되고 변환되어 사용됩니다.

Arm의 프로그램화된 구현

아래 단계에서 보여지는 메소드는 맨 처음의 기반 Arm과 첫번째 Arm 두개를 연결시키는 조인트에 필요한 메소드입니다.

```
void CreateBase(Joint joint)
{
    PhysicsJoint commonJoint = (PhysicsJoint)joint;
    float mass = 1;
    float radius = 0.03f;

    CapsuleShape baseShape = new CapsuleShape(new CapsuleShapeProperties(mass,
        new Pose(new Vector3(0, radius + 0.015f / 2, 0)),
        radius, 0.015f));
```

```
    // create connect point for base. Top and center of its shape.
    commonJoint.State.Connectors[0] = new EntityJointConnector(null,
        new Vector3(0, 0, 1),
        new Vector3(0, 1, 0), // joint rotates around the Y(vertical axis)
        new Vector3(0, baseShape.CapsuleState.Radius * 2 +
baseShape.CapsuleState.Dimensions.Y, 0));

    ArmLinkEntity link = new ArmLinkEntity("base",
       "Ibr3_j0.obj",
       new Pose(new Vector3(0, -radius - 0.005f, 0),
       TypeConversion.FromXNA(xna.Quaternion.CreateFromAxisAngle(new xna.Vector3(0, 1, 0),
0))));

    link.State.Pose.Position = State.Pose.Position;
    link.UpperJoint = commonJoint;
    _baseLink = link;
    _baseLink.CapsuleShape = baseShape;
    _links.Add(link);
}

ArmLinkEntity _arm1Link;

/// <summary>
/// Link 1
/// </summary>
[DataMember]
public ArmLinkEntity Arm1Link
{
    get { return _arm1Link; }
    set { _arm1Link = value; }
}

void CreateArm1(Joint joint)
{
    PhysicsJoint commonJoint = (PhysicsJoint)joint;
    commonJoint.State.Connectors[1] = new EntityJointConnector(null,
        new Vector3(0, 0, 1),
        new Vector3(0, 1, 0),
        new Vector3(0, 0, 0));

    ArmLinkEntity link = new ArmLinkEntity("arm1",
       "Ibr3_j1.obj",
       new Pose(new Vector3(0, -MIDPOINT_Y, 0),
       TypeConversion.FromXNA(xna.Quaternion.CreateFromAxisAngle(new xna.Vector3(0, 1, 0),
0))));

    link.State.Pose.Position = State.Pose.Position;
    _arm1Link = link;
    _links.Add(link);
}
```

아래 그림은 기반 부분과 첫번째 Arm의 연결 상태를 물리적 관점에서 표현한 그림입니다. 둘 다

모두 조인트 축이 빨간색 화살표로 표시됩니다.

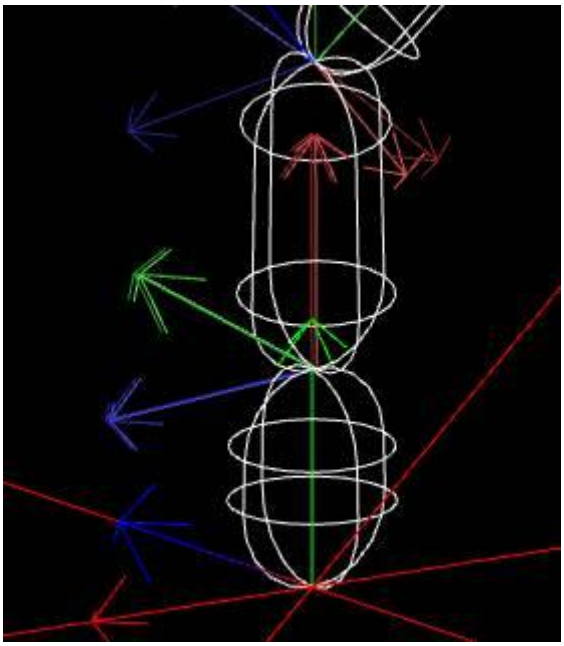


그림 5 - 기반 부분과 첫번째 Arm 연결

```
void CreateArm2(Joint joint)
{
    PhysicsJoint commonJoint = (PhysicsJoint)joint;
    commonJoint.State.Connectors[0] = new EntityJointConnector(
        null,
        new Vector3(0, 1, 0),
        new Vector3(0, 0, -1),
        new Vector3(0, ARM_LENGTH2, 0));

    commonJoint.State.Connectors[1] = new EntityJointConnector(
        null,
        new Vector3(0, 1, 0),
        new Vector3(0, 0, -1),
        new Vector3(0, 0, 0));

    ArmLinkEntity link = new ArmLinkEntity(
       "arm2",
       "lbr3_j1.obj",
       new Pose(new Vector3(0, MIDPOINT_Y, 0),
       Quaternion.RotationAxis(new Vector3(1, 0, 0),
       (float)Math.PI)));

    link.State.Pose.Position = State.Pose.Position;
    _arm2Link = link;
    _links.Add(link);
}
```

위의 코드에서 두번의 new EntityJointConnector 호출은 정상적인 엔터티 연결을 위해 매우 중요한 부분입니다. 해당 객체 내에서 두번째와 세번째 파라메터로 사용된 값들은 조인트들의 방향을 나타내며, 각각 법선 방향과 로컬 축 방향을 나타냅니다.

Step 4: 튜토리얼 실행

튜토리얼 예제는 아래와 같이 실행될 수 있습니다. MSRS 실행창에서 아래의 명령어를 실행하시기 바랍니다.

```
binWdsshost /port:50000 /tcpport:50001 /manifest:"samplesWconfigWSimulationTutorial4.manifest.xml"
```

또한 웹브라우저를 실행하여 아래의 연결 주소를 통해 실행 상태를 확인할 수 있습니다.

http://localhost:50000/directory

그림 6 - 디렉토리 서비스

위의 서비스 목록중에서 /simulatedlbr3 항목을 클릭하면, 다음과 같은 개별 조인트들의 상태값을 확인할 수 있습니다.

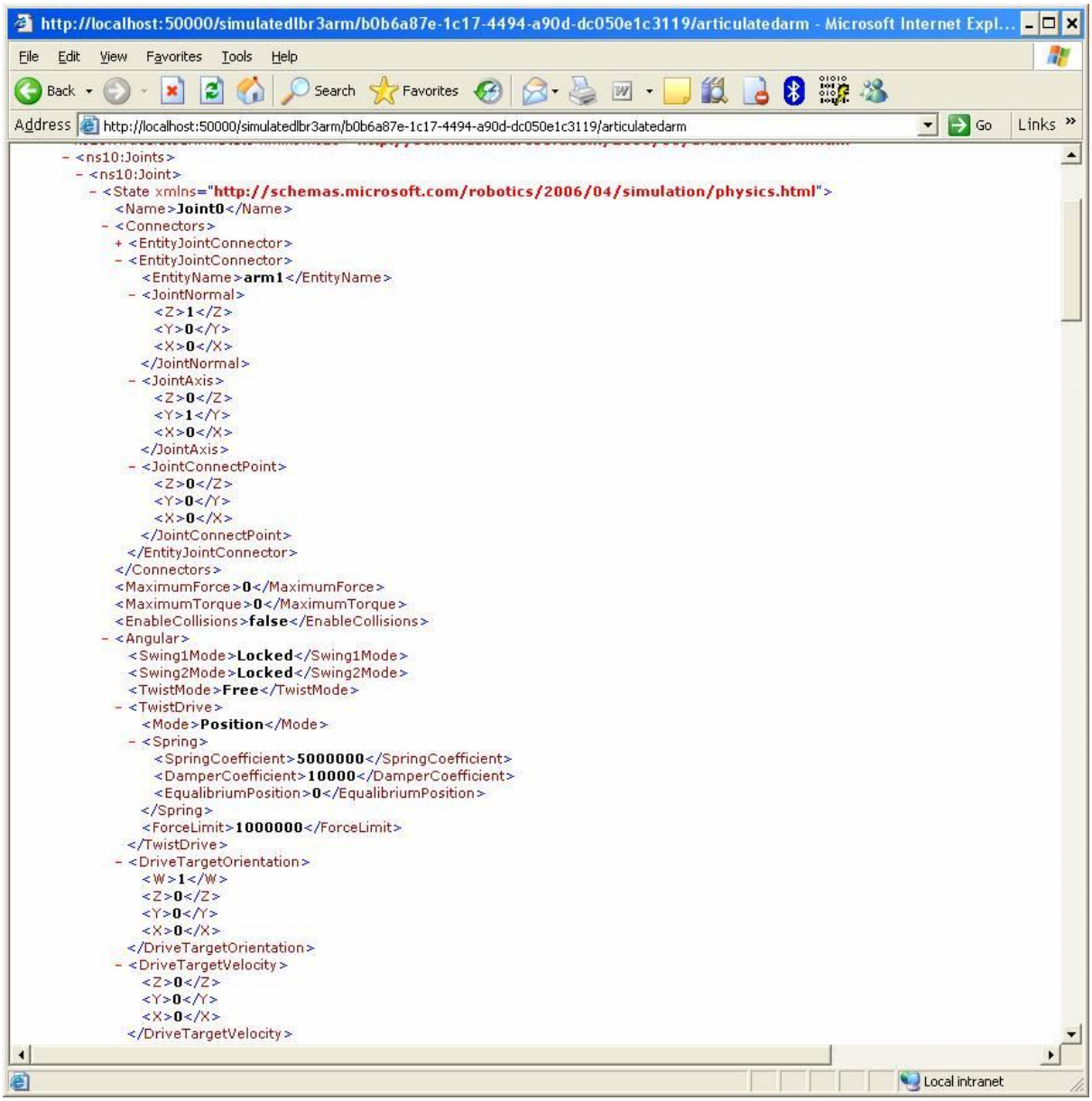


그림 7 - 시뮬레이션 Arm의 상태

초기 Obj 파일에 대한 변환

시뮬레이션 파일은 자동으로 Obj 파일을 이진 파일 형태로 변환합니다. 따라서 맨 처음 실행될 때 변환에 따른 시간 지연이 발생할 수 있습니다.

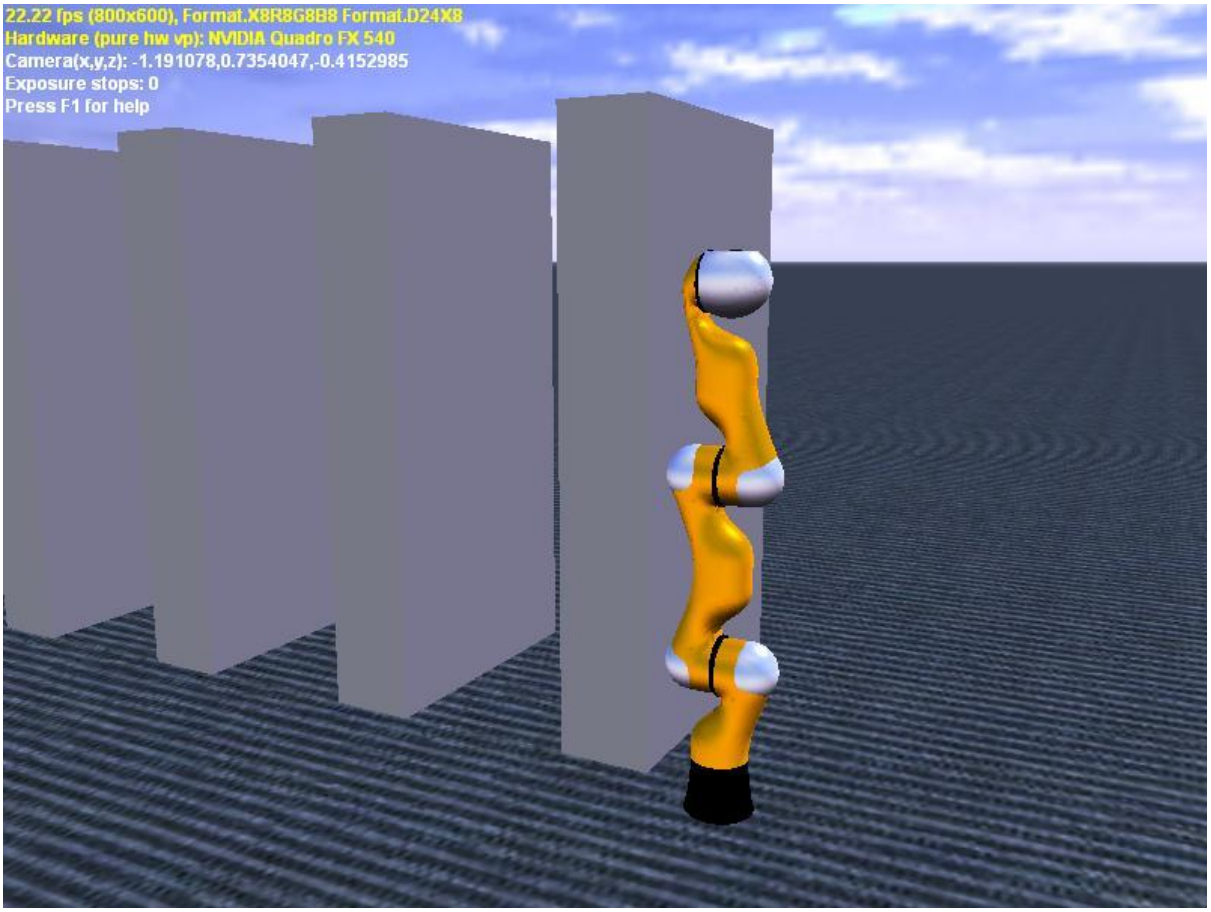


그림 8 - 시뮬레이션으로 표현된 Arm

이번 튜토리얼을 실행하면 자동으로 SimpleDashboard 서비스가 같이 실행됩니다. SimpleDashboard 실행 후 서버명에 localhost를 입력하고 Connect 버튼을 클릭합니다.

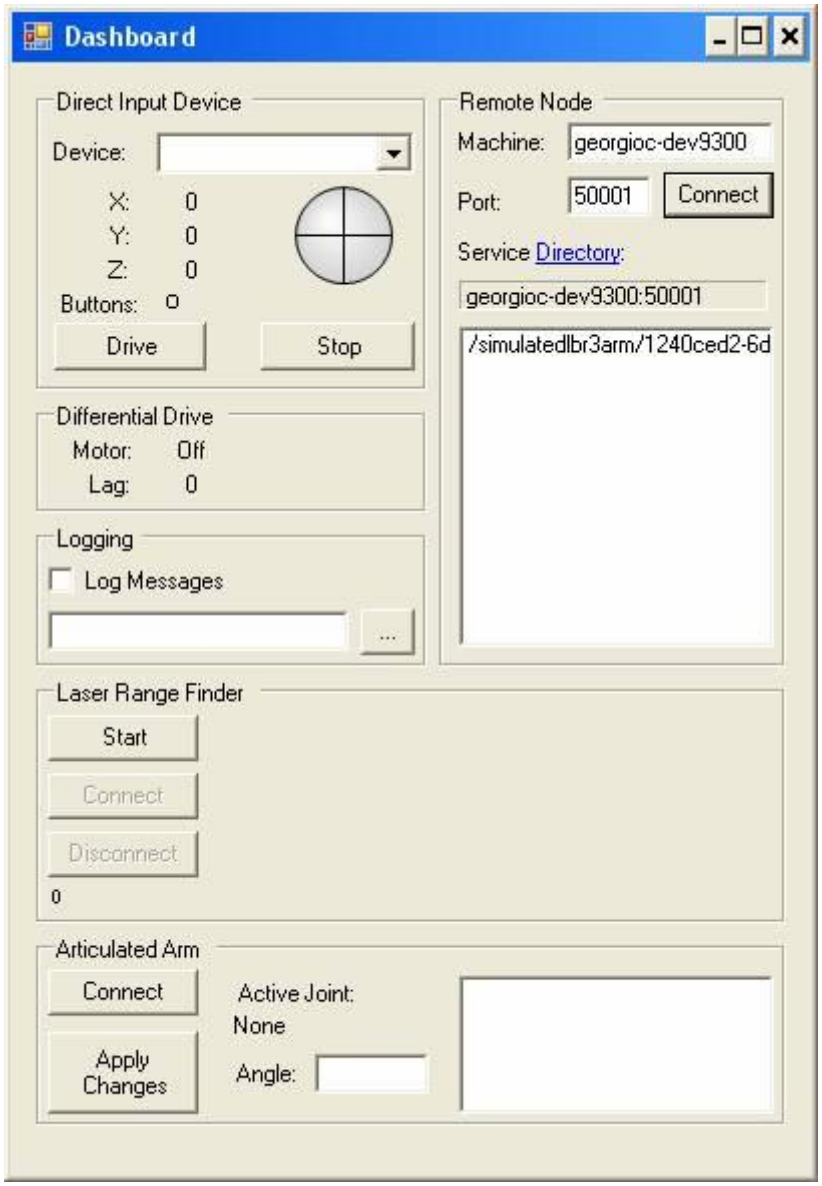


그림 9 - SimpleDashboard를 실행한 화면

화면 하단에서 Articulated Arm 영역에 있는 Connect 버튼을 클릭하면, 우측에 관련된 조인트 목록이 표시됩니다.

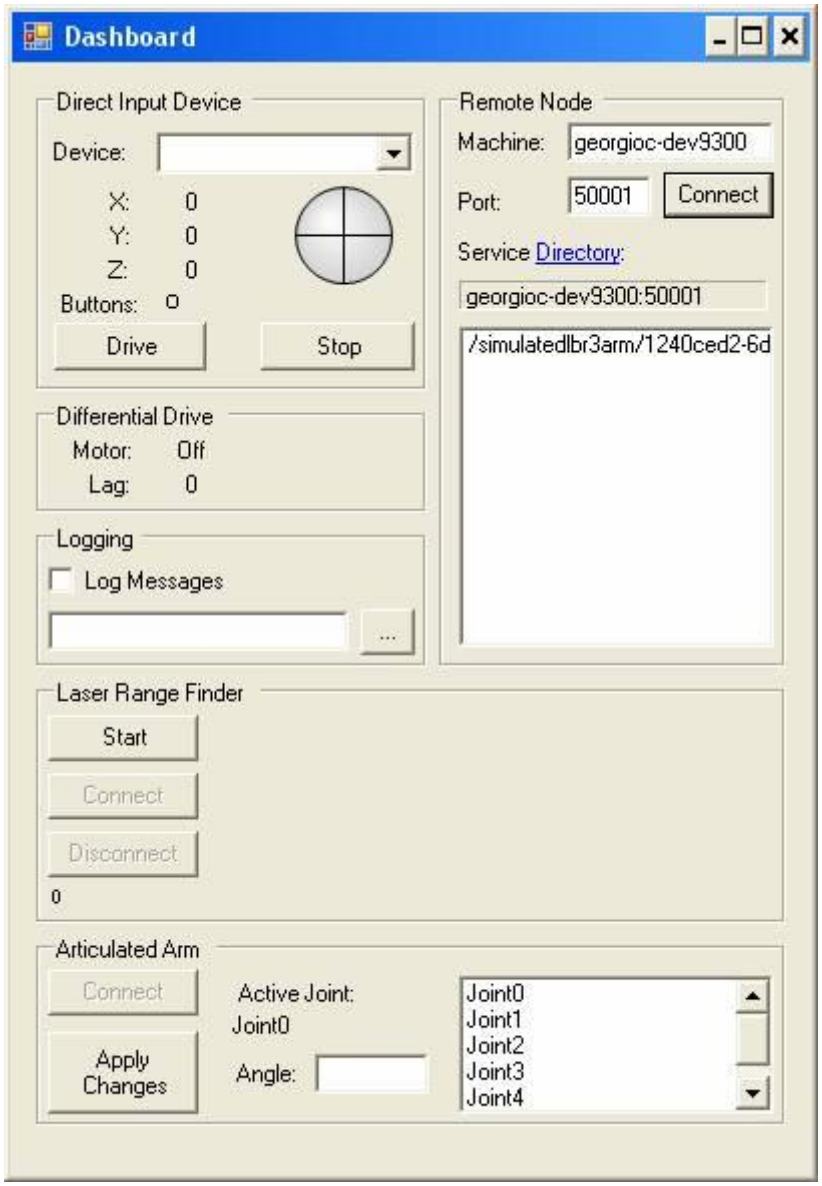


그림 10 - SimpleDashboard에서 조인트 목록을 가져온 결과

목록에 표시되는 Joint 항목들을 선택한 후 Angle 부분에 각도 값 (-90 ~ 90)을 입력합니다. 각도 값 입력 후 Apply Changes 버튼을 클릭합니다.

1. Joint1에 대해 50 값을 입력 후 Apply Changes 버튼을 클릭합니다.
2. Joiny0에 대해 -90도 값을 입력 후 Apply Changes 버튼을 클릭합니다.
3. 아래와 같이 Arm이 이동 후 블록이 차례로 넘어지는 결과를 확인할 수 있습니다.

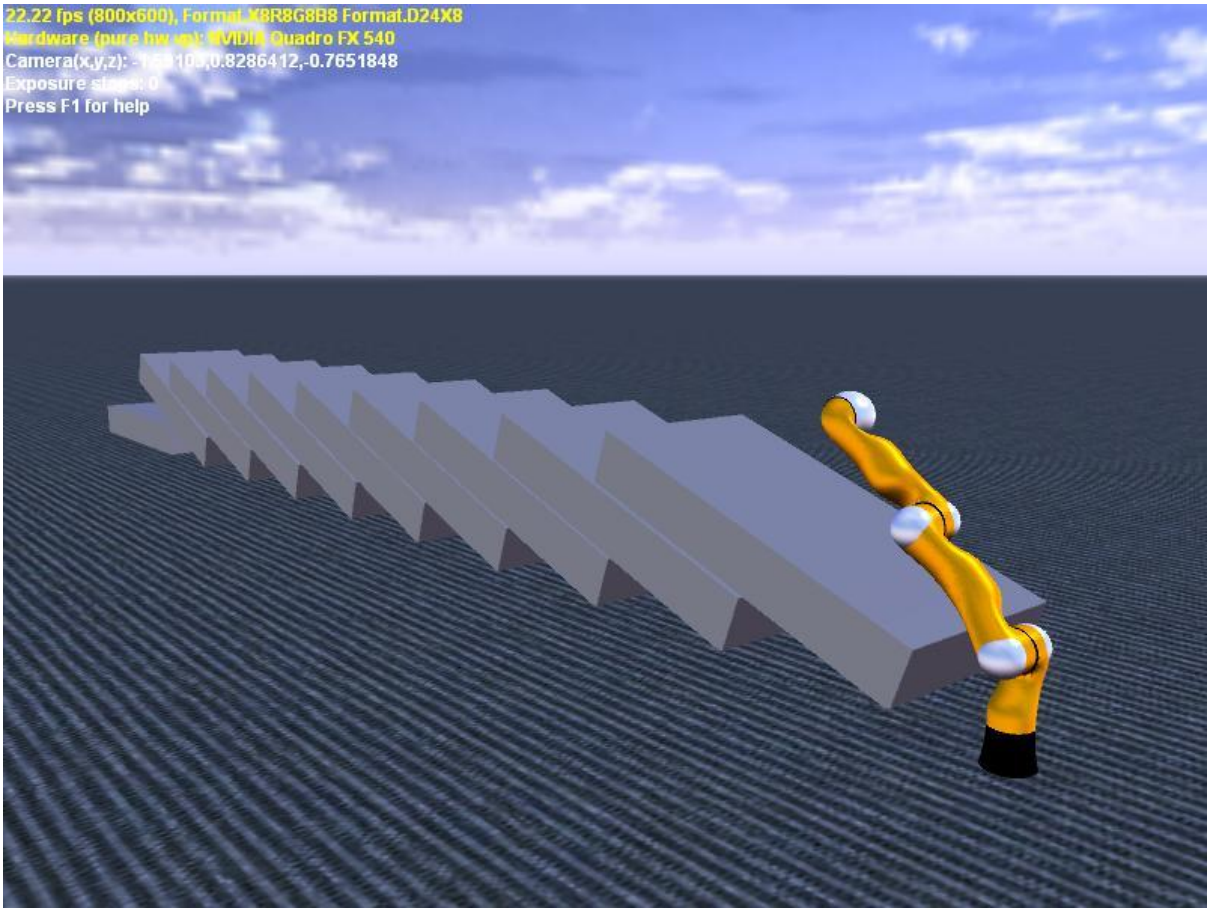


그림 11 - 시뮬레이션 Arm의 실행 결과

시뮬레이션 튜토리얼 5 (C#) - 기하학적 엔터티 생성

이번 튜토리얼에서는 간단한 기하학적 엔터티 생성에 대한 예제와 이러한 방법을 이용하여 복잡한 엔터티를 구현하는 방법에 대해 소개합니다.

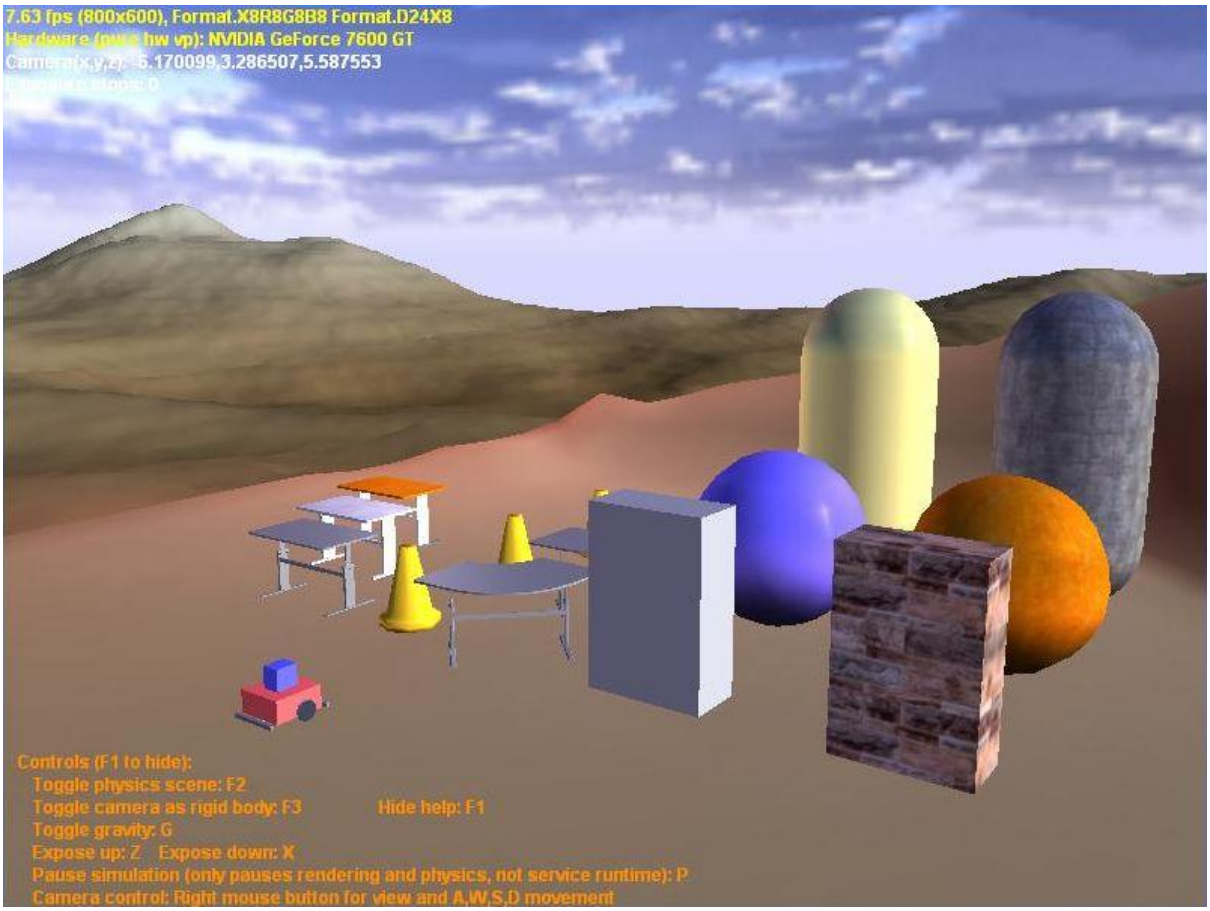


그림 1 - 렌더링된 엔터티들

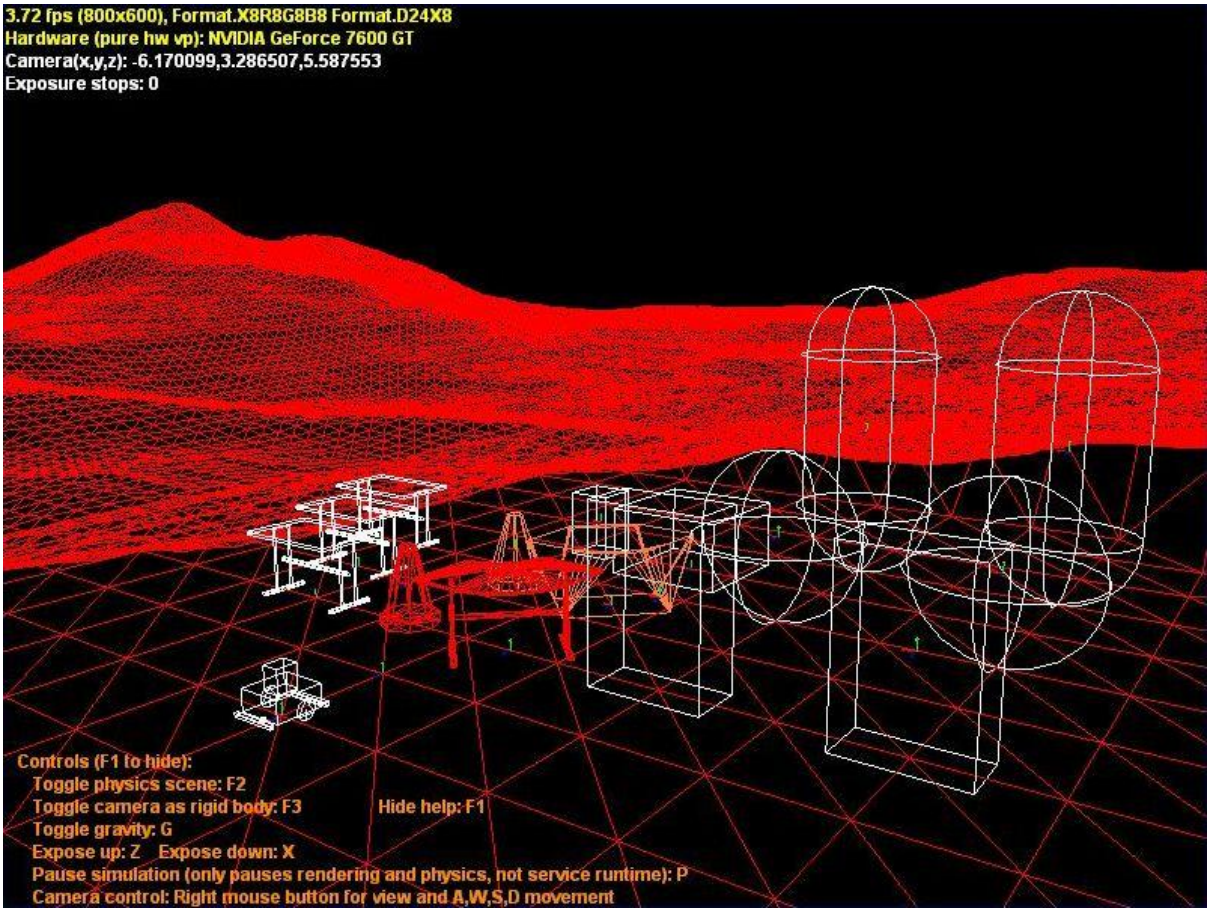


그림 2 - 엔터티들의 물리적 화면 표시

시작하기

이번 튜토리얼은 samplesWSimulationTutorialsWTutorial5 폴더에 소스 코드가 존재하며, SimulationTutorial5.csproj 파일을 Visual Studio에서 로드하여 시작할 수 있습니다.

Step 1: 서비스 생성과 시작

이번 튜토리얼은 이전의 튜토리얼에서와 같이 PopulateWorld를 호출하고 여러 개의 엔터티를 시뮬레이션 공간상에 추가하는 방식으로 진행됩니다. 이번 튜토리얼의 목적은 기하학적 엔터티를 생성하는 과정을 분석하는 것이기 때문에, 대부분의 작업은 생성과 초기화 하는 과정에서 진행됩니다.

```
private void PopulateWorld()
{
    // skydome and sun
    AddSky();

    // basic ground
    AddGround();

    // add simple 3D models with basic physics
    AddSimpleModels();

    // single shapes from physics (with color or texture)
```

```
    AddColoredShapes();
    AddTexturedShapes();

    // multi-part entities (with or without 3D model)
    AddFurniture();

    // geometric Pioneer robot
    AddModularRobot(new Vector3(0.0f, 0.5f, 1.0f));

    // convex shape from 3D model
    AddConvexMesh("table_02.obj", new Vector3(-0.5f, 1f, -2.5f));
    AddConvexMesh("street_cone.obj", new Vector3(0.7f, 1f, -2.5f));

    // triangle shape from 3D model
    AddTriangleMesh("table_02.obj", new Vector3(-0.5f, 0f, -1f));
    AddTriangleMesh("street_cone.obj", new Vector3(0.7f, 0.37f, -1f));
}
```

엔터티

시뮬레이터를 효과적으로 사용하기 위해서는 다양한 엔터티들이 어떻게 생성되는 지에 대해 이해하는 것이 중요합니다. 이번 튜토리얼에서는 다양한 엔터티들이 아래의 정의에 따라 언급될 예정입니다.

- 물리 엔터티: 시뮬레이션에 대한 물리 엔진에 의해 사용되는 물리 객체의 기본 표현, 코드에서는 Ageia 물리엔진에 의해 와이어프레임 모양으로 나타내어 집니다 (Sphere, Capsule, Box, HeightField, ConvexMesh, TriangleMesh)
- 렌더링 엔터티: 비주얼 피드백을 위해 사용되는 3D 삼각 모델. 코드에서는 VisualEntityMesh 클래스의 인스턴스입니다.
- 시뮬레이션 엔터티: 시뮬레이터에서의 기본 엔터티. 단일 시뮬레이션 엔터티는 물리 엔터티를 가지지 않거나 또는 여러 개의 물리 엔터티를 가질 수 있고, 또한 렌더링 엔터티도 가지지 않거나 또는 여러 개의 렌더링 엔터리를 가질 수 있습니다. 코드에서는 VisualEntity 클래스 (또는 이 클래스로부터 파생된 클래스)의 일반적인 인스턴스입니다.

Step 2: Sky, Light 및 Lighting 엔터티

Sky

SkyEntity는 시뮬레이션 공간상에서 하늘 형상을 표현하는 시뮬레이션 엔터티입니다. 이 엔터티는 기본적으로 렌더링만을 위해서 사용되기 때문에 물리적으로 대응되는 대상을 가지지 않습니다. SkyEntity는 Sun과 Sky Dome의 복합으로 구성됩니다. Sun은 직접적인 빛이며 벡터는 빛이 들어오는 방향을 나타내고, 빛의 강도와 색의 농담을 위한 색상이 지정됩니다. Sky Dome은 두 개의 Cubemap 이미지 파일을 사용하여 렌더링된 전체적인 공간을 포함하는 큰 구를 표현합니다. AddSkey() 메소드에서는 아래와 같이 구현됩니다.

```
// Add a sky using a static texture. We will use the sky texture
// to do per pixel lighting on each simulation visual entity
SkyDomeEntity sky = new SkyDomeEntity("skydome.dds", "sky_diff.dds");
SimulationEngine.GlobalInstancePort.Insert(sky);

// Add a directional light to simulate the sun.
```

```
LightSourceEntity sun = new LightSourceEntity();
sun.State.Name = "Sun";
sun.Type = LightSourceEntityType.Directional;
sun.Color = new Vector4(1.0f, 1.0f, 1.0f, 1f);
sun.Direction = new Vector3(1.0f, -0.6f, 0.1f);
SimulationEngine.GlobalInstancePort.Insert(sun);
```

Cubemap은 6개의 이미지로 구성된 텍스쳐입니다. 큐브의 각 면을 위해 하나의 이미지씩 사용합니다. 이러한 맵의 목적은 환경을 둘러싸고 있는 것들에 대한 시각적인 표현을 나타내며, 배경을 표현하기 위해 사용됩니다. 또한 이 맵들은 보다 복잡한 lighting의 계산을 위해서도 사용됩니다. 이러한 맵에는 다양한 형태의 포멧이 있습니다: spherical, double ellipsoid, meridian-parallel. Cubemap은 비디오 카드에 의해 하드웨어 적으로 지원되는 단순한 포멧이며, Microsoft DirectX texture 파일 (.dds)에 직접 저장될 수 있습니다.

SkyEntity는 두 개의 CubeMap을 사용합니다. 첫 번째 것은 하늘을 시각적으로 표현하기 위해 사용되며 아주 세부적이고 Sky Dome을 시각화 한 것입니다. 두 번째 것은 시뮬레이터 엔터티들의 이미지 기반 lighting 환경을 수행하기 위해 사용됩니다. 표면의 각 점 들은 해당 법선(normal) 방향에 있는 Sky의 영역에 해당되는 부분으로부터 빛을 받습니다. 이러한 종류의 lighting 표현은 확산 효과를 나타냅니다. 환경 맵은 무한히 먼 환경을 표현합니다. 즉, 두 엔터티의 상대적인 위치는 하늘로부터 빛을 표현하는 데 있어서 중요하지 않습니다. 만약 두개의 표면이 동일한 방향을 가진다면, 그들은 상대적인 위치에 관계없이 동일한 빛을 받을 것입니다. 대 낮의 하늘과 밤의 별 들은 관측자가 이동하더라도 변하지 안는 것을 볼 수 있기 때문에 무한한 거리의 개념을 적용하는 있어서 좋은 예입니다.

Cubemap을 다루기 위한 툴은 ATI 개발자 웹사이트 (http://www.ati.com/developer/cubemapgen/index.html)에서 찾을 수 있습니다. 이 툴은 또한 확산되는 빛의 계산에 있어서 하위의 cubemap을 만들어 내기 위해 사용될 수 있습니다. Sky dome으로 사용가능한 Cubemap은 인터넷에서 쉽게 구할 수 있습니다. (Codemonsters 웹사이트 : http://www.codemonsters.de/html/textures_cubemaps.html). Prof. Paul Debevec의 홈페이지 (http://www.debevec.org/)에서는 Cubemap과 HDR sky probe(다른 포멧의 Cubemap)들을 구할 수 있습니다.

Light

Light는 아래와 같이 정의됩니다.

- Entity를 정의하는 하나의 이름 (String)
- 빛의 행위를 정의하는 타입 (enum LightEntityType). Directional과 Omni 타입을 지원합니다.
- 빛의 강도와 농도를 나타내는 색상 (Vector4)
- 빛의 위치와 방향을 정의하는 Pose (Pose)

Directional Lights느 항상 동일한 방향을 가지고 빛을 비추는 것을 나타내며 태양을 좋은 예제로 볼 수 있습니다.

Omni light는 해당 위치에서 모든 방향으로 빛을 뿌리는 것을 나타냅니다. 전구와 양초가 좋은 예가 될 수 있습니다.

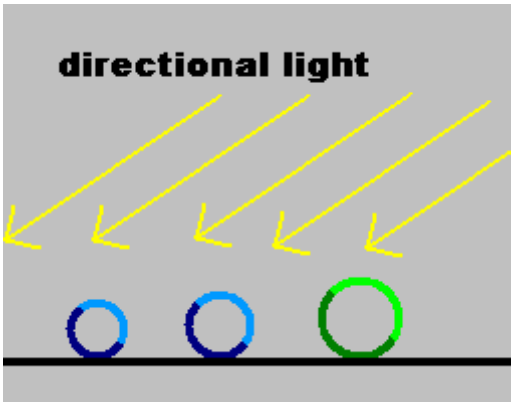


그림 3 – *Directional* Light

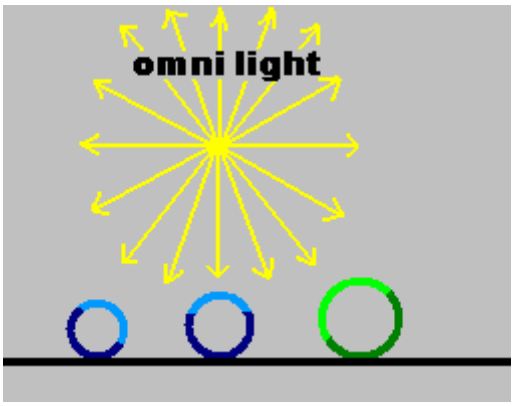


그림 4 – *Omni* Light

Light를 시뮬레이션에 추가하기 위해서는 LightEntity를 생성한 후에 시뮬레이션 엔진에 추가해야 합니다. Light 속성은 수정되거나 삭제될 수 있으며, 이름으로 엑세스할 수 있습니다.

```
// adding a new omni light
LightSourceEntity mylight = new LightSourceEntity();
mylight.State.Name = "RedLight";
mylight.Color = new Vector4(1f, 0.5f, 0.5f, 1f);
mylight.State.Pose.Position = new Vector3(10, 6, -10);
mylight.Type = LightSourceEntityType.Omni;
SimulationEngine.GlobalInstancePort.Insert(mylight);
```

SkyEntity는 light를 포함하고 있으며, 이러한 light는 Sky가 시뮬레이션 엔진에 추가되자 마자 Active light들의 리스트에 추가된 다는 것에 유의하시기 바랍니다.

Lighting 개요

PC가 표현하는 색상은 사람이 인지하는 대역에 비해 매우 제한되어 있습니다. 색상들은 기본적으로 red-green-blue 값들의 조합으로 표시됩니다. Vector4가 색상을 표현하기 위해 사용되며, 이 중 앞의 3개의 벡터값은 0.0과 1.0 사이의 값으로 red, green, blue값을 표현하기 위해 사용되고 32비트 실수형으로 표시됩니다. Black은 [0,0,0,1]로 표시되고, white는 [1,1,1,1]로 표시됩니다. 마지막 항목은 투명도를 결정하는 alpha channel 값을 나타냅니다. 해당 값을 1로 설정하면 툴투명하게 처리됩니다.

빛이 공간에 추가되고, 색상을 통해 빛의 강도와 농도가 결정되면, 객체의 색상은 해당 객체 표면의 색상과 해당 객체가 light로부터 받는 빛과 곱해져서 표현됩니다.

Step 3: 파일로부터 3D Mesh 파일 사용하기

3D 모델 파일이 가능하다면, 해당 파일 이름을 지정하여 3D 모델을 렌더링 엔터티로 사용하여 시뮬레이션 엔터리를 생성하는 것이 가능합니다. 현재 지원되는 포멧은 Wavefront Obj (.obj) 포멧입니다. 아래의 경우에서는 메쉬 파일을 사용하는 예를 보여줍니다.

```
private void AddSimpleModels()
{
    SingleShapeEntity table_a = new SingleShapeEntity(
        new BoxShape(
            new BoxShapeProperties(
            20.0f, // mass in kilograms.
            new Pose(new Vector3(0.0f, 0.345f, 0.0f)), // relative pose
            new Vector3(1.2f, 0.7f, 1.2f))), // dimensions
        new Vector3(-0.5f,1f,-4.0f));
    table_a.State.Assets.Mesh = "table_02.obj";
    table_a.State.Name = "table_angle";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(table_a);

    SingleShapeEntity cone = new SingleShapeEntity(
        new BoxShape(
            new BoxShapeProperties(
            0.5f, // mass in kilograms.
            new Pose(), // relative pose
            new Vector3(0.45f, 0.7f, 0.45f))), // dimensions
        new Vector3(0.7f,2f,-4.0f));
    cone.State.Assets.Mesh = "street_cone.obj";
    cone.State.Name = "StreetCone";

    // Insert entity in simulation.
    SimulationEngine.GlobalInstancePort.Insert(cone);
}
```

시뮬레이션에서는 Physics가 엔터티의 위치를 나타내기 때문에, 3D 모델이 물리적 표현에 대해 중심이 일치하도록 유의할 필요가 있습니다.

```
[...]
new BoxShape(
    new BoxShapeProperties(
    20.0f, // mass in kilograms.
    new Pose(new Vector3(0.0f, 0.345f, 0.0f)), // relative pose
[...]
```

불행히도, OBJ 파일 포멧은 많은 수의 하위 포멧으로 구성되고 텍스트 기반으로 되어 있어서, 성능상에 문제가 있습니다. 이러한 문제를 해결하기 위해 맨 처음 파일이 로드될 때, 시뮬레이션 내부에서 이러한 파일을 .bos 이름을 가지는 바이너리 형태로 변환되어 저장됩니다.

Step 4: 물리 엔터티로부터 Geometric Mesh 사용하기

Simple Shape들

AddColoredShapes와 AddTexturedShapes 메소드는 렌더링 엔터티들이 해당 물리적 대상으로부터 알마나 간단히 얻어질 수 있는 지를 보여주는 예입니다.

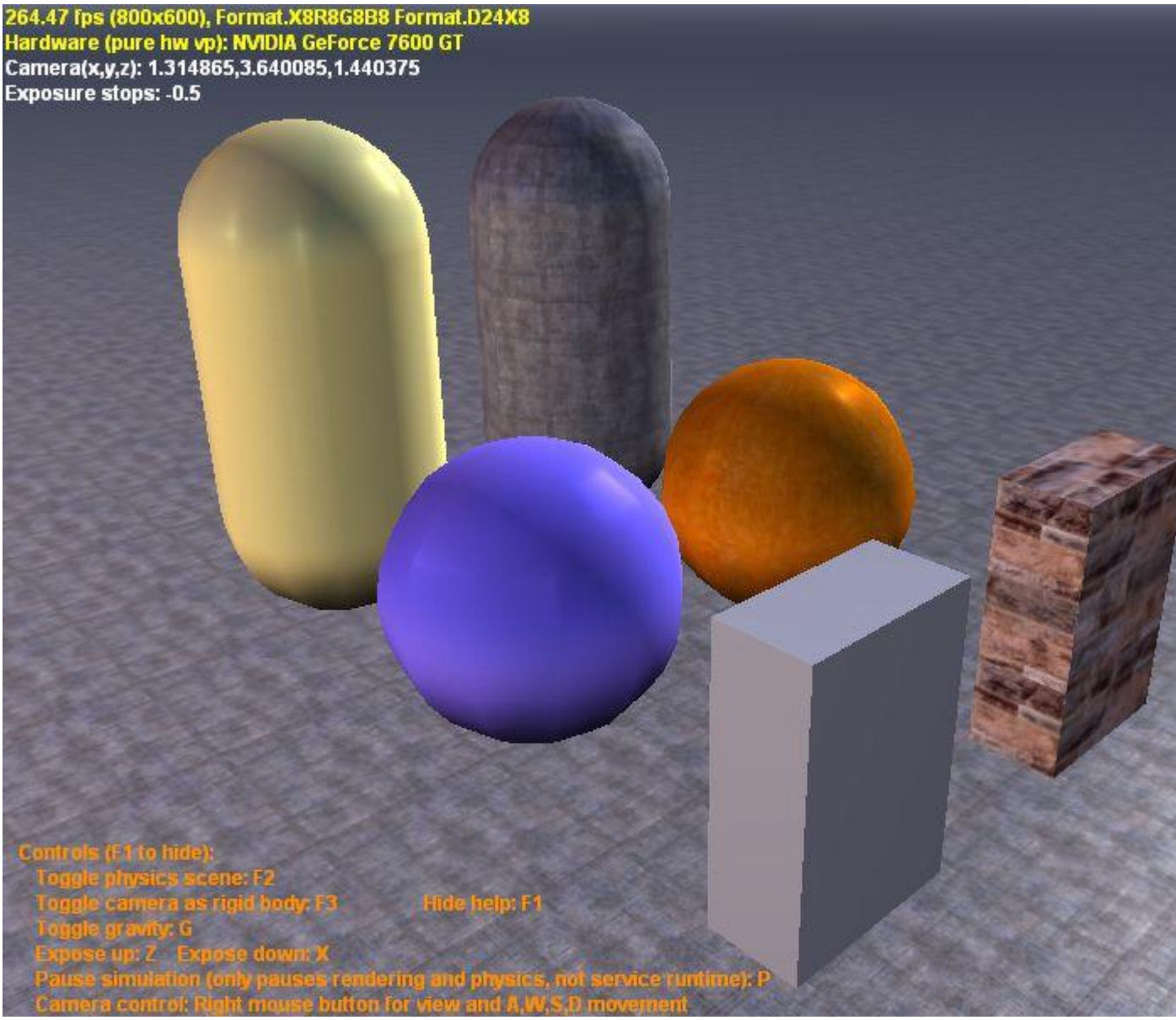


그림 5 - 랜더링된 엔터티

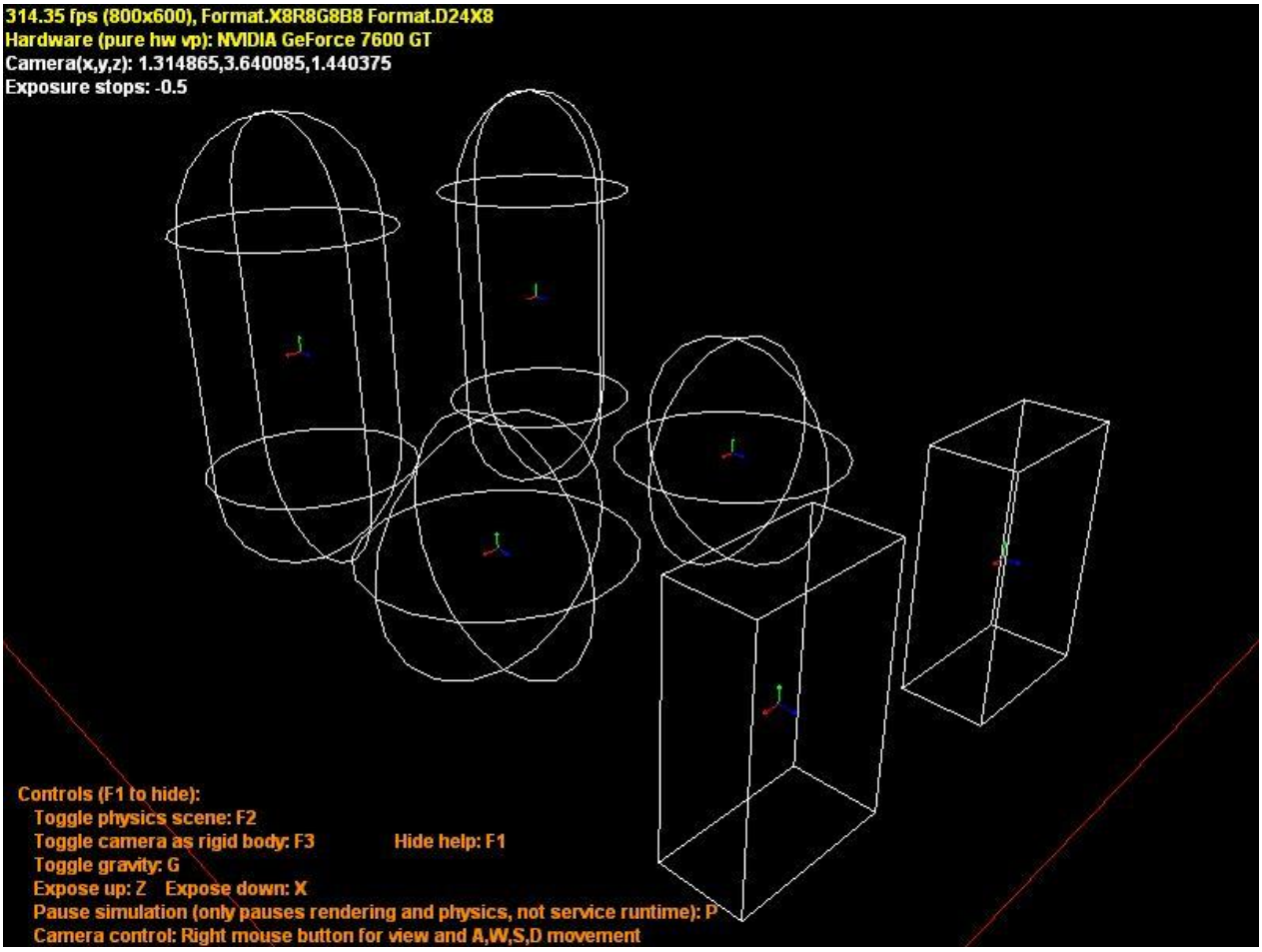


그림 6 - 물리적 표현의 엔터티

아래의 예제에서는 SingleShapeEntity가 physical sphere shape을 통해 생성됩니다. 3D 파일이 생성되지 안았으며, 시뮬레이터가 메쉬 형태를 자동으로 생성합니다. 이러한 경우 diffuse 색상을 임의로 지정할 수 있으며, 만약 별도로 색상을 지정하지 않으면, 시뮬레이터가 자동으로 지정합니다.

```
//---- bluish sphere ----
SphereShapeProperties cSphereShape = null;
SingleShapeEntity cSphereEntity = null;

cSphereShape = new SphereShapeProperties(10f, new Pose(), 0.8f);
cSphereShape.Material = new MaterialProperties("bphere", 0.5f, 0.4f, 0.5f);
cSphereShape.DiffuseColor = new Vector4(0.4f, 0.3f, 0.7f, 1.0f);
cSphereEntity = new SingleShapeEntity(new SphereShape(cSphereShape), new Vector3(-2.0f,
1.0f, -3.0f));
cSphereEntity.State.Name = "bluesphere";
SimulationEngine.GlobalInstancePort.Insert(cSphereEntity);
```

유사한 방식으로 텍스쳐를 적용할 수 있으며, 아래의 예제는 텍스쳐 파일을 엔터티에 적용한 예입니다.

```
//---- wooden sphere ----
SphereShapeProperties tSphereShape = null;
```

```
SingleShapeEntity tSphereEntity = null;

tSphereShape = new SphereShapeProperties(10f, new Pose(), 0.8f);
tSphereShape.Material = new MaterialProperties("tphere", 0.5f, 0.4f, 0.5f);
tSphereEntity = new SingleShapeEntity(new SphereShape(tSphereShape), new Vector3(-4.0f,
1.0f, -3.0f));
tSphereEntity.State.Assets.DefaultTexture = "Wood_cherry.jpg";
tSphereEntity.State.Name = "texsphere";
SimulationEngine.GlobalInstancePort.Insert(tSphereEntity);
```

위의 예제에서는 tSphereEntity.State.Assets.DefaultTexture를 사용하여 텍스쳐 맵이 엔터티 전체에 걸쳐 적용되었습니다. 또한 특별한 Shape에 한하여 tSphereShape.TextureFileName을 사용하는 것도 가능합니다. 엔터티 내에서는 오직 물리 Shape만 있기 때문에, 결과는 변경되지 않습니다. 그러나 두 개 이상의 물리 Shape을 포함하고 있는 시뮬레이션 엔터티들에 대해서는 적용되지 않을 수 있습니다.

간단한 Shape 구성

아래 그림에서 첫 번째 객체는 튜토리얼 2번에서 사용되는 엔터티와 동이합니다. 이 TableEntity는 MultiShapeEntity로부터 파생되었으며, 여러 개의 물리 엔터티들로 정의되었습니다.

그림 7 - 렌더링된 엔터티

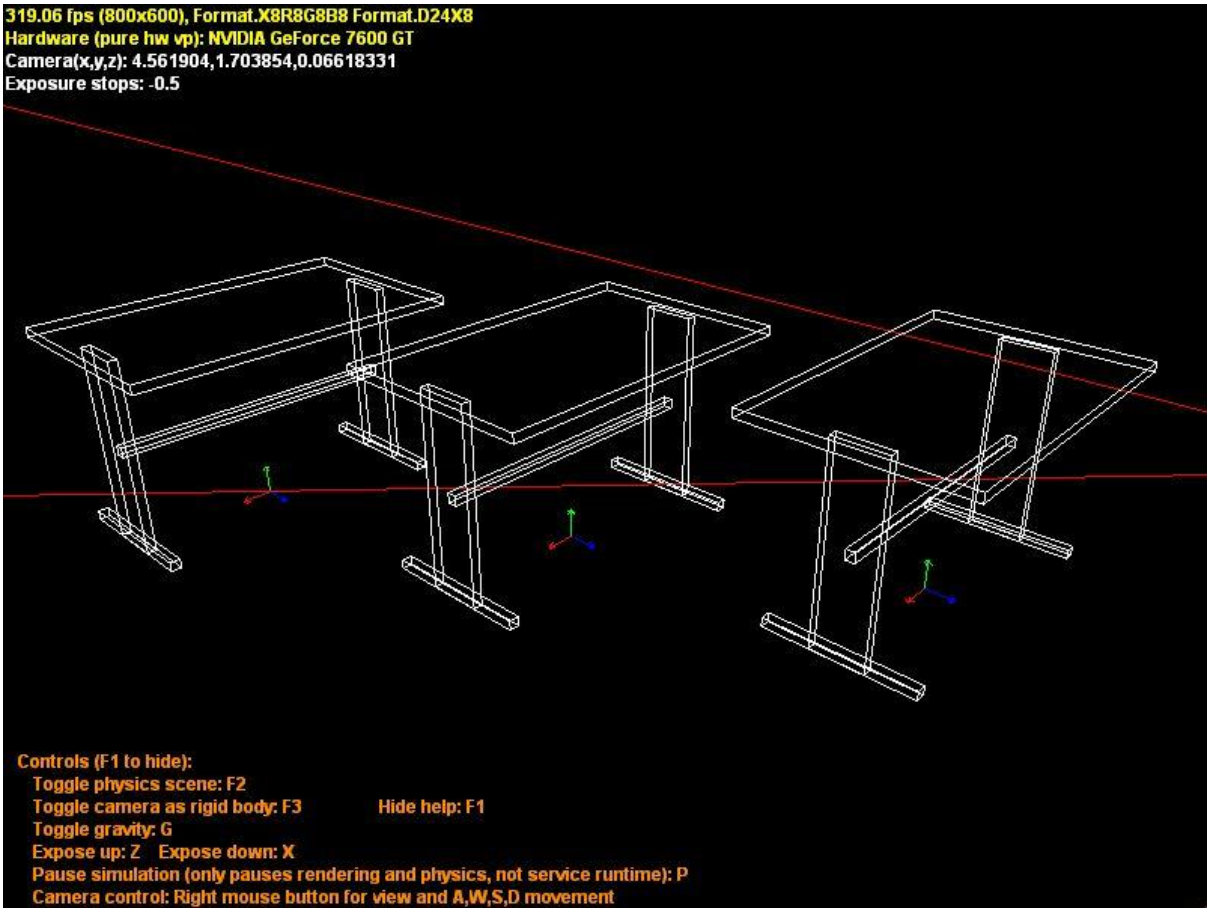


그림 8 - 물리적 표현의 엔터티

만약 3D 메쉬 파일이 정의된다면, 시스템은 해당 물리적 상태에 대응하는 단일의 기하학적 엔터티를 생성합니다. 그리고 메쉬가 정의되지 않는다면, 시스템은 각각의 물리적 shape 별로 기하학적 엔터티를 생성합니다. 결과는 위에서 각각 두번째 테이블과 세번째 테이블로 표현됩니다.

또한 TableEntity에 대해 텍스쳐 파일을 정의함으로써 렌더링된 엔터티 전체에 대해 텍스쳐를 적용할 수 있습니다. 두 번째 테이블에서 보여지는 것 처럼, 모든 Shape은 동일한 텍스쳐를 가집니다.

```
// create an instance of the table custom entity
// using physics-derived rendering entity and a single texture
TableEntity physEntityT = new TableEntity(new Vector3(2.5f, 1.0f, -2.0f));
physEntityT.State.Name = "table_phys_t";
physEntityT.State.Assets.DefaultTexture = "cellwall.jpg";
SimulationEngine.GlobalInstancePort.Insert(physEntityT);
```

반대로, 엔터티 내부에 각 Shape을 특성화시킬 필요가 있는데, 이러한 것은 DefaultTexture 할당을 생략하고 테이블에 사용되는 각 Shape에 대해 텍스쳐나 색상을 지정하므로써 가능합니다. 세번째 테이블의 각 Shape이 평평하지 않은 외관을 가지면서 색상이 변해가는 것을 확인할 수 있습니다.

```
// add a shape for the table surface
            BoxShape tableTop = new BoxShape(
                new BoxShapeProperties(10,
                new Pose(new Vector3(0, tableHeight, 0)),
```

```
                new Vector3(tableWidth, tableThinkness, tableDepth))
            );

            tableTop.State.TextureFileName = "Wood_cherry.jpg";

            // add a shape for the left leg body
            BoxShape tableLeftLeg = new BoxShape(
                new BoxShapeProperties(2, // mass in kg
                new Pose(
                    new Vector3(-tableWidth / 2 + legOffset, tableHeight / 2, 0)),
                new Vector3(legThickness, tableHeight, tableDepth / 4))
            );

            tableLeftLeg.State.DiffuseColor = new Vector4(0.8f, 0.8f, 0.8f,1.0f);

[...]
            BoxShapes = new List<BoxShape>();
            BoxShapes.Add(tableTop);
            BoxShapes.Add(tableLeftLeg);
```

파라메터에서의 우선 순위

여러 개의 파라메터가 동시에 적용된다면, 일부 충돌이 발생할 수 있습니다 (color, per-shape texture, per-entity texture). 이러한 경우 아래의 룰이 적용됩니다.

- 색상과 텍스쳐 파라메터는 엔터티가 추가되지 전에 설정될 수 있습니다.
- 3D 메쉬가 제공된다면, 색상과 텍스쳐들은 메쉬에서 정의되고 이러한 값들이 사용됩니다.
- 각각의 엔터티별 텍스쳐가 정의된다면, 각각의 Shape별 색상과 텍스쳐는 무시됩니다.
- 각 Shape 텍스쳐는 각 Shape 색상에 우선합니다.
- 만약, 색상이나 텍스쳐가 정의되지 않았다면, 객체는 grey (RGB [0.5f,0.5f,0.5f,1.0f]) 값으로 표시됩니다.

로봇

색상과 텍스쳐에 대한 고려는 로봇 객체에 대해서도 동일하게 적용됩니다. 3D 메쉬 파일이 없어도 로봇 구성이 가능하여, 기본적인 물리 객체들의 조합으로 구성이 가능합니다. AddModularRobot 함수는 튜토리얼 2번에서 소개되었던 Pioneer3 로봇을 생성하지만, 메쉬 파일이 없기 때문에 기본적인 물리 Shape들로만 구성이 됩니다. 각 Shape에 대한 색상을 지정함으로써 원본 로봇과 비슷하게 로봇을 꾸밀 수 있습니다.

```
[...]

            Pioneer3DX robotBaseEntity = new Pioneer3DX(position);

            // specify color.

            robotBaseEntity.ChassisShape.State.DiffuseColor = new Vector4(0.7f, 0.3f,
0.3f, 1f);

[...]
```

```
            // Create a Laser Range Finder Entity.
            // Place it 30cm above base CenterofMass.
            LaserRangeFinderEntity laser = new LaserRangeFinderEntity(
                new Pose(new Vector3(0, 0.30f, 0)));
            laser.State.Name = "LaserRangeFinder";

            laser.LaserBox.State.DiffuseColor = new Vector4(0.3f,0.3f, 0.7f, 1f);


[...]
```

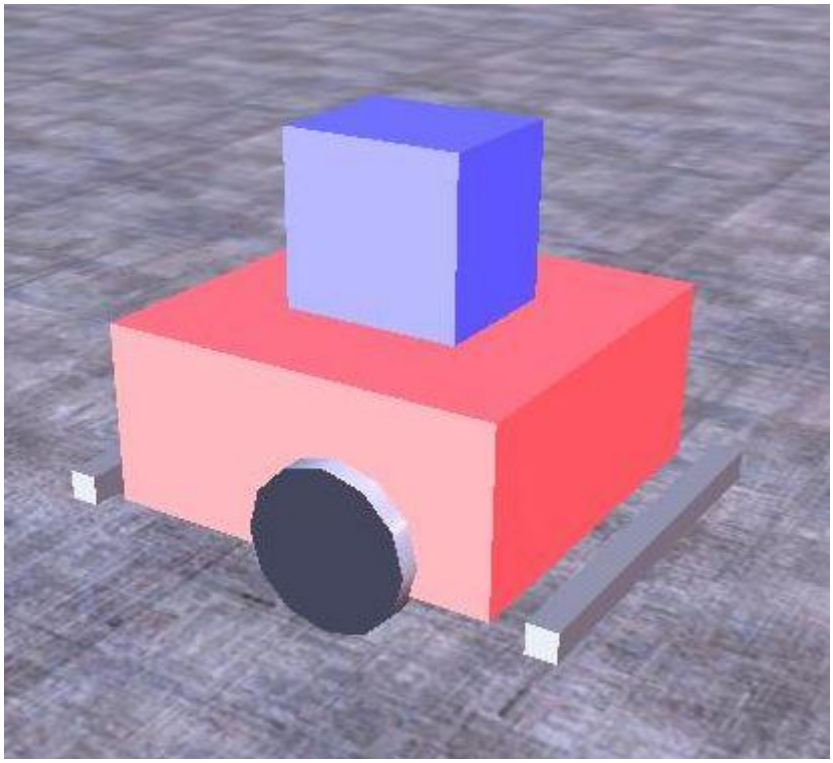


그림 9 - 렌더링된 엔터티

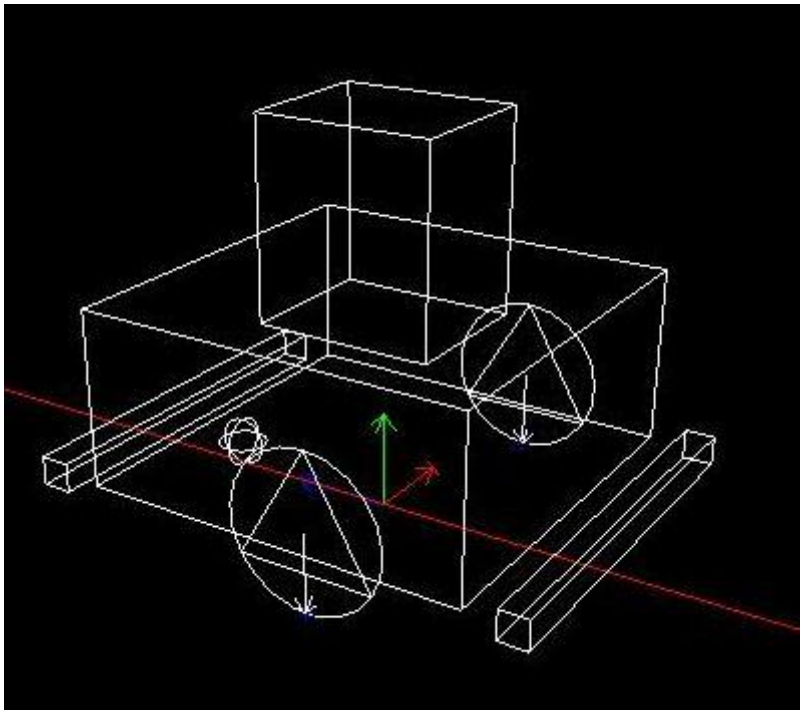


그림 10 - 물리적 화면으로 표현된 엔터티

이 예제에서는 로봇을 단순히 박스들로만 구성하였지만, 테이블과는 달리 각 엔터티들이 좀 더 정확한 표현을 가질 수 있도록 여러 개의 Shape들로 구성이 되었습니다. 이러한 방식으로 3D 메쉬파일이 없어도 로봇을 간단한 Shape들만의 구성으로 쉽게 만들 수 있습니다.

Step 5: Terrain 엔터티 사용

일부 시뮬레이션에 있어서, 간단한 바닥만으로는 충분하지 않을 수 있습니다. TerrainEntity는 각각의 높이를 지정한 데이터 소스로서 비트맵을 사용하여 보다 더 복잡한 바닥을 생성할 수 있습니다. Terrain을 다루고 렌더링하기 위한 많은 데이터 구조가 있습니다만, 정밀한 시뮬레이션을 위하여, Robotics Studio에서는 Ageia 데이터 구조와 유사한 구조를 사용합니다. Ageia HeightFieldShape은 최적화된 메모리 공간과 충돌 관리 기능을 제공하는 샘플들의 그리드를 사용하여 heightfield를 표현합니다. Terrain은 렌더링에 대해 물리 Primitive와 균일한 메쉬로서 HeightFieldShape을 사용하여 구성되며, 해당 텍스쳐가 Terrain의 맨 위에 매핑됩니다.

```
private void AddGround()
{
    TerrainEntity ground = new TerrainEntity(
                    "terrain.bmp",
                    "terrain_tex.jpg",
                    new MaterialProperties("ground",
                        0.2f, // restitution
                        0.5f, // dynamic friction
                        0.5f) // static friction
                    );

    SimulationEngine.GlobalInstancePort.Insert(ground);
}
```

엔터티는 비트맵을 로드하고 높이 값으로서 픽셀 값을 사용하여 height field를 구성합니다. 가능한 높이의 범위가 0 부터 255 사이의 정수형 이기 때문에, 약간 블록화된 형태로 바닥이 표시

가 됩니다. 이러한 이유로 인해, 픽셀 값은 오프셋 (128는 높이 0을 나타냄)을 가지며, scale을 다운시켜 (10으로 나눈 값이 10cm의 해상도를 가짐) 사용합니다. 샘플들 간의 거리는 1미터로 가정을 합니다. 정확히 작동하게 하기 위하여, Terrain 비트맵은 32 배수에 1을 더한 차원으로 구성됩니다 (예 129*129, 65*257).

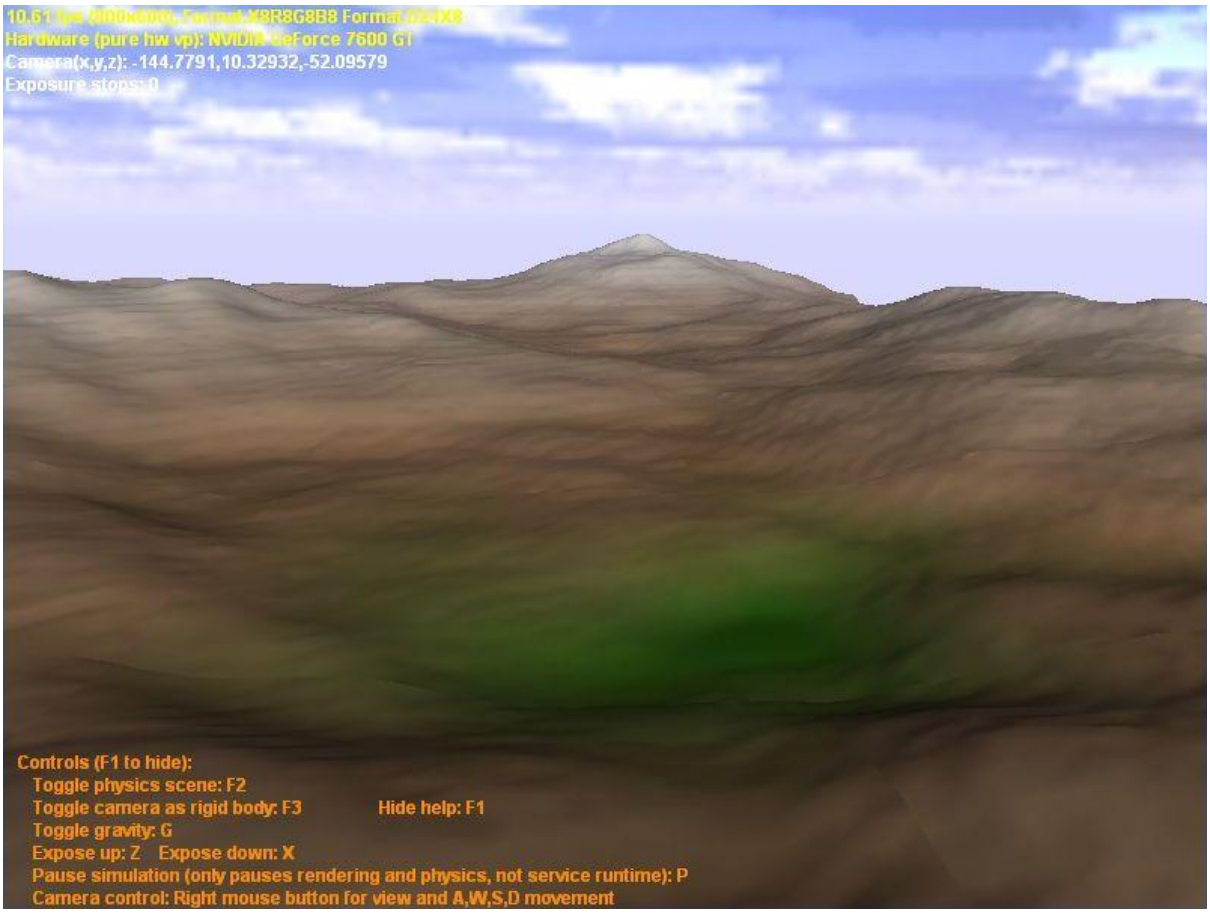


그림 11 - 렌더링된 Terrain 엔터티

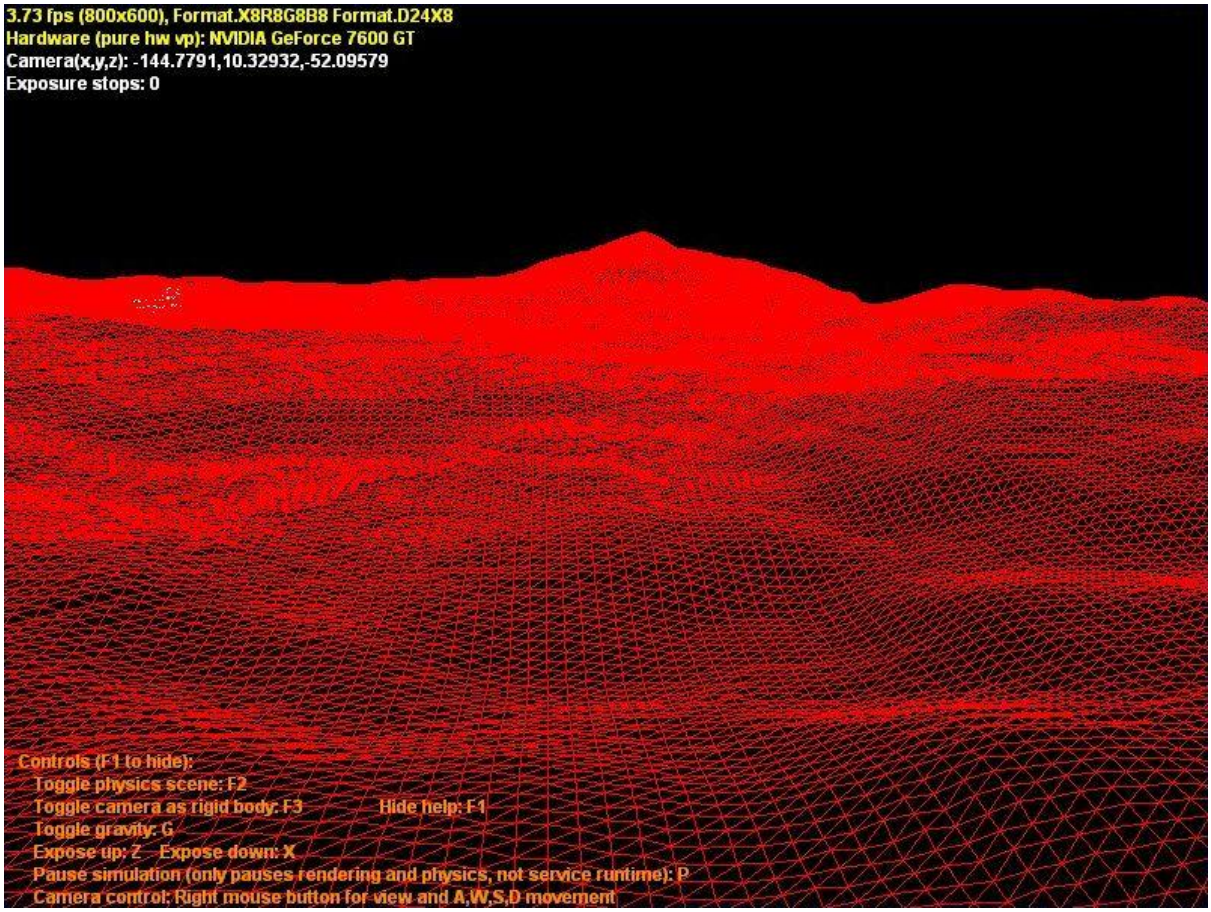


그림 12 - 물리적 화면으로 표현된 Terrain 엔터티

로딩된 height field는 Chunk로 분할되며, 각 Chunk에 대해 물리적 HeightField와 해당되는 메쉬가 생성됩니다. Field를 Chunk로 분할하여 임의의 크기의 Terrain을 만드는 것이 가능합니다. 단일 HeightFieldShape는 크기가 제한되어 있지만 임의의 숫자의 HeightFieldShape들을 사용할 수 있습니다. 또 다른 장점은 렌더링 시간을 들 수 있으며, 이를 통해 Frustum Culling을 수행할 수 있고 현재의 View 안에 존재하지 않는 모든 Chunk들을 렌더링하는 것을 방지할 수 있습니다. 추가적으로, Chunk 구조는 Level-Of-Detail (LOD) 접근법과 같은 보다 더 유연한 렌더링 전략을 구현하기 위해 수정할 수도 있습니다.

TerrainEntity 대신에, TerrainEntityLOD를 사용하는 것도 가능하며 이 시뮬레이션 엔터티는 그려지는 삼각형의 수를 감소시키기 위한 대략적인 Level-Of-Detail 접근법을 사용합니다. 각 Chunk는 카메라와의 거리에 의존하는 Detail Level의 렌더링과 연관되며, 이 Level은 각 프레임 단위로 갱신됩니다.

개발자들은 Terrain 데이터 구조를 각 개발자들의 요구에 맞게 수정할 수 있습니다. 이러한 용도로 만들어진 두 개의 엔터티들에 대한 소스코드가 entities.cs에 포함되어 있으며, 이 파일은 samplesWSimulationWEntitiesW 폴더에 있습니다.

Terrain에 대한 비트맵을 생성하기 위해, Windows의 그림판 프로그램이나 또는 인터넷에서 구 할 수 있는 비디오 게임용의 다양한 Terrain 편집기를 사용할 수 있습니다. 이러한 편집기의 대부분은 heightfield를 이미지 형태로 추출할 수 있습니다. 비트맵의 또 다른 대안으로서, 엔터티는 GridFloat 파일로부터 로딩하는 것을 지원합니다. 이 파일은 GIS 포멧으로서 지형적 데이터를 표현하기 위해 사용됩니다. 이 포멧은 상당히 단순하며 데이터 크기와 속성에 대한 정보를 헤더 정보로 포함하고 헤더 파일(.hdr)과 그리드 상에 정렬되어 있는 height 샘플들의 리스트를 포함하고 있는 데이터 파일(.flt)로 구성됩니다. 이러한 포멧을 이용한다면, 실제 기하학적으로 측정된

데이터를 등록시킬 수 있습니다. 이 방법은 미국의 United States Geological Survey (USGS) 웹 사이트로부터 데이터를 다운로드 받을 때 선택 가능한 포멧 중의 한 가지 입니다. USGS의 "Seamless" 웹페이지에서 USGS 데이터베이스 중에서 임의 지역을 선택 한 후, 포멧을 선택하여 웹기반 방식으로 데이터를 자유롭게 다운로드 받을 수 있습니다.

Step 6: 기하학적 엔터티로 부터 물리 엔터티 얻기

어떠한 경우에서는 간단히 그대로 사용 가능한 기하학적 기능을 제공하는 것이 더 편리할 수 있으며, Ageia 사의 물리 엔진은 이러한 것을 위해서 내장된 기능을 제공합니다. 즉, 기하학적 엔터엔터부터 시작하여, 물리엔진은 물리적 표현으로서 사용 가능한 근사한 형태의 Convex Hull을 생성해 낼 수 있습니다.

```
private void AddConvexMesh(string name, Vector3 pos)
{
    Shape cshape = null;
    SimplifiedConvexMeshEnvironmentEntity centity = new
SimplifiedConvexMeshEnvironmentEntity(
        pos,
        name,
        cshape);
    centity.State.MassDensity.Mass = 30;
    centity.State.Name = "Convex mesh:" + name + ":" + Guid.NewGuid().ToString();
    SimulationEngine.GlobalInstancePort.Insert(centity);
}
```

3D 메쉬가 로드되고 엔터티를 렌더링 하기 위해 사용됩니다. 생성된 물리 엔터티는 Ageia 엔진에 의해 사용됩니다.

물리적 형태로서 원래의 기하학적 데이터를 사용할 수 있는데, 이러한 물리적 엔터티는 엔터티를 렌더링할 때의 동일한 삼각형들을 사용하여 구성됩니다. 이러한 방식을 통해서 만들어지는 결과는 3D 모델 데이터에의 정교함과 유사한 정도로 만들어 질 수 있습니다. 엔터티들은 세밀한 충돌을 감지할 수 있으나 상당히 높은 컴퓨터 연산을 요구합니다. 그럼에도 불구하고, 이러한 경우에서는 해당 메쉬가 이동이 불가능한 엔터티로서만 사용될 수 있습니다.

모든 물리 객체들은 엔터티들과 충돌할 때 정확하게 반응을 해야 하지만, 만약 mass 값을 0으로 설정하면, 해당 엔터티들은 움직이지 않게 됩니다.

```
private void AddTriangleMesh(string name, Vector3 pos)
{
    Shape tshape = null;
    TriangleMeshEnvironmentEntity tentity = new TriangleMeshEnvironmentEntity(
        pos,
        name,
        tshape);
    tentity.State.MassDensity.Mass = 0f;
    tentity.State.Name = "Triangle mesh:" + name + ":" + Guid.NewGuid().ToString();
    SimulationEngine.GlobalInstancePort.Insert(tentity);
}
```

위의 예제에서는 두 개의 3D 모델 데이터가 AddSimpleModels 메소드에서 사용되었습니다. 아래의 화면을 통해서도 확인할 수 있듯이, 주어진 3D 모델 데이터에 대해 물리 형상들이 각각 다른 수준의 근사치를 가질 수 있음을 확인할 수 있습니다. (아래 그림에서는 매우 조잡한 형태의 박스

형상과, Convex Hull 형상, 그리고 물리적 충돌 엔터티를 구현하기 위해 사용된 기하학적 형상 등의 예제를 볼 수 있습니다).

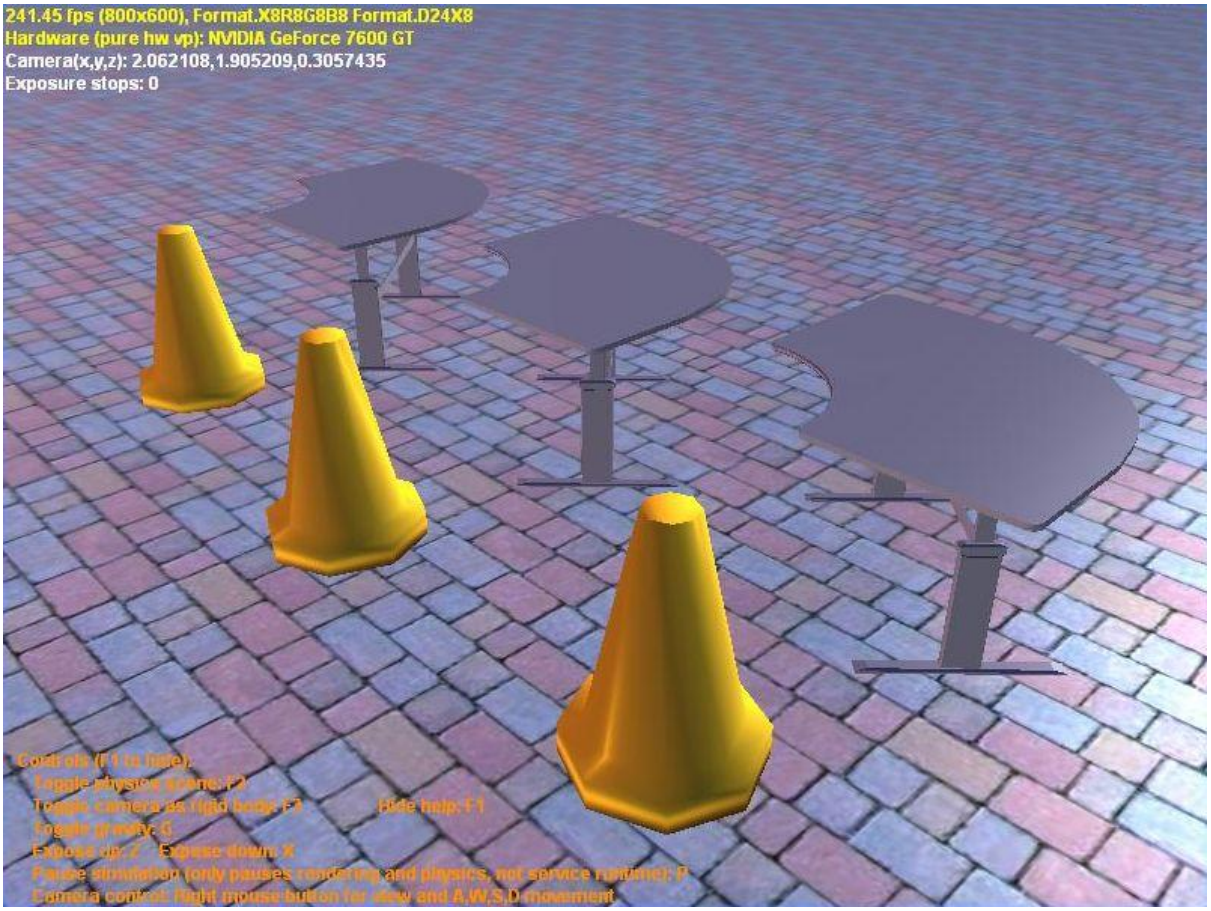


그림 13 - 렌더링된 엔터티들

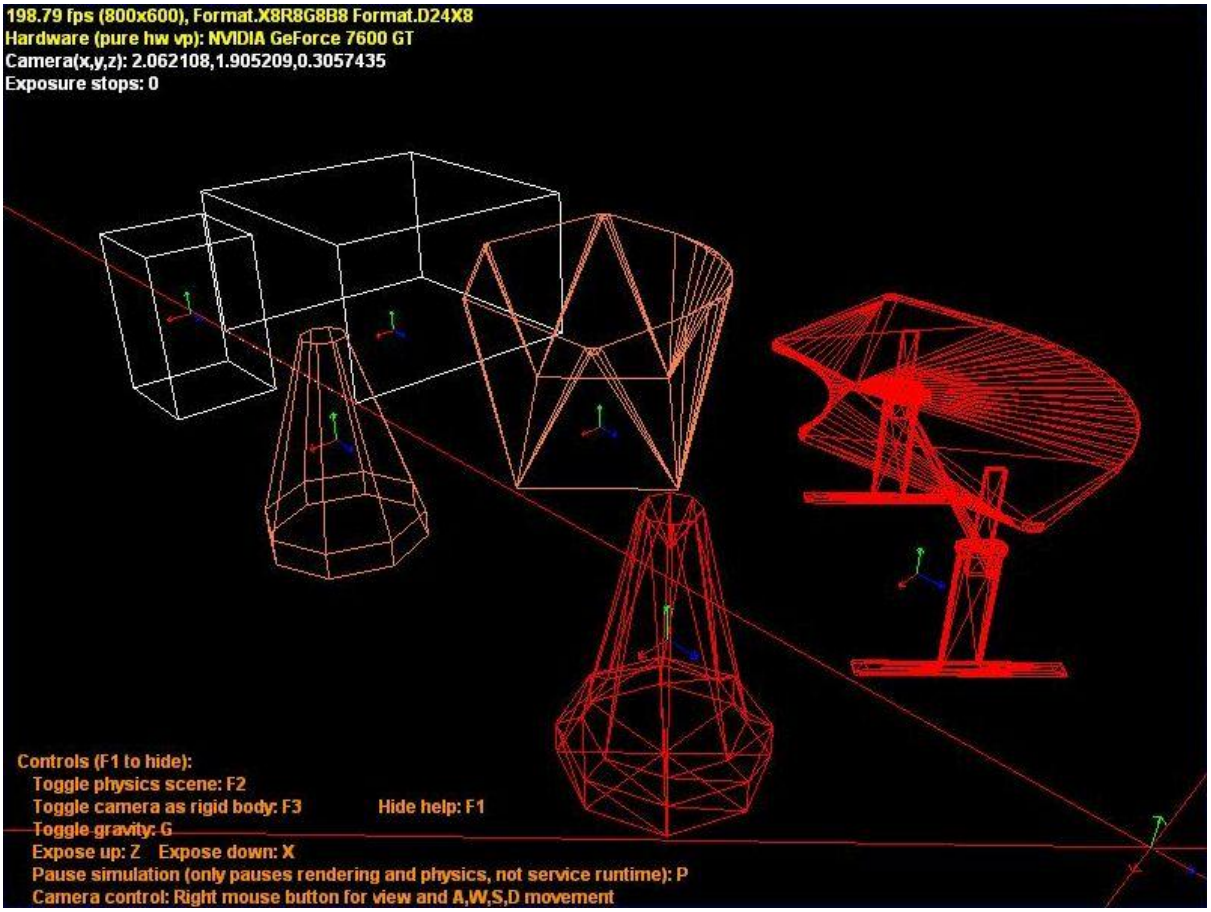


그림 14 - 물리적 형태로 표현된 엔터티들

주의사항

이들 두 가지 형태의 연산들은 상당히 복잡하고 완성되기 위해 상당한 시간이 소요될 수 있습니다 (예를 들어 감지할 수 있을 정도의 몇 초 정도). 하지만 첫 번째 실행 후에는 결과가 바이너리 파일 형태로 원래의 메쉬 파일 경로와 동일한 경로안에 저장되기 때문에, 두 번째 실행부터는 빠르게 샐행됩니다. (.convex.physics.bin 또는 .triangle.physics.bin 와 같이 해당 파일 뒤에 붙습니다).

이러한 방식에 있어서, 한 가지 문제점은 이러한 연산들은 잘 정의된 메쉬 파일이 입력될 때 가능하다는 부분입니다. Flipped 삼각형, Topological 불균형, 또는 self-intersecting 된 부분과 같은 문제를 표현하는 메쉬들에 대해서는 Convex 형태 생성이 안될 수 도 있습니다. 이러한 경우에 있어서는 엔터티들이 시뮬레이터에 안 나타날 수 있습니다. 삼각 모양의 생성은 이러한 유형의 문제들 보다는 좀 더 잘 작동되는 편이나 기타 일부 3D 모델에 대해서는 여전히 실행이 되지 안을 수 있습니다.

Step 7: 튜토리얼 실행

프로젝트를 빌드한 후, Robotics Studio command 창에서 아래와 같이 실행시킬 수 있습니다.

```
binWdsshost /port:50000 /tcpport:50001
/manifest:"samplesWconfigWSimulationTutorial5.manifest.xml"
```

시뮬레이션 튜토리얼 6 (C#) - 시뮬레이션 편집기

이번 튜토리얼에서는 Visual 시뮬레이션 환경 툴을 활용하여, 엔터티들을 생성하거나 수정하는 방법들에 대해 소개합니다.

시작하기

이번 튜토리얼을 위한 C# 코드는 없으며, 시뮬레이터 환경 툴 만이 사용됩니다.

Step 1 : 시뮬레이터 실행

시작 메뉴에서 MSRS 메뉴를 선택한 후, Visual Simulation Environment 하부에 있는 Basic Simulation Environment 메뉴를 선택하여 실행합니다.

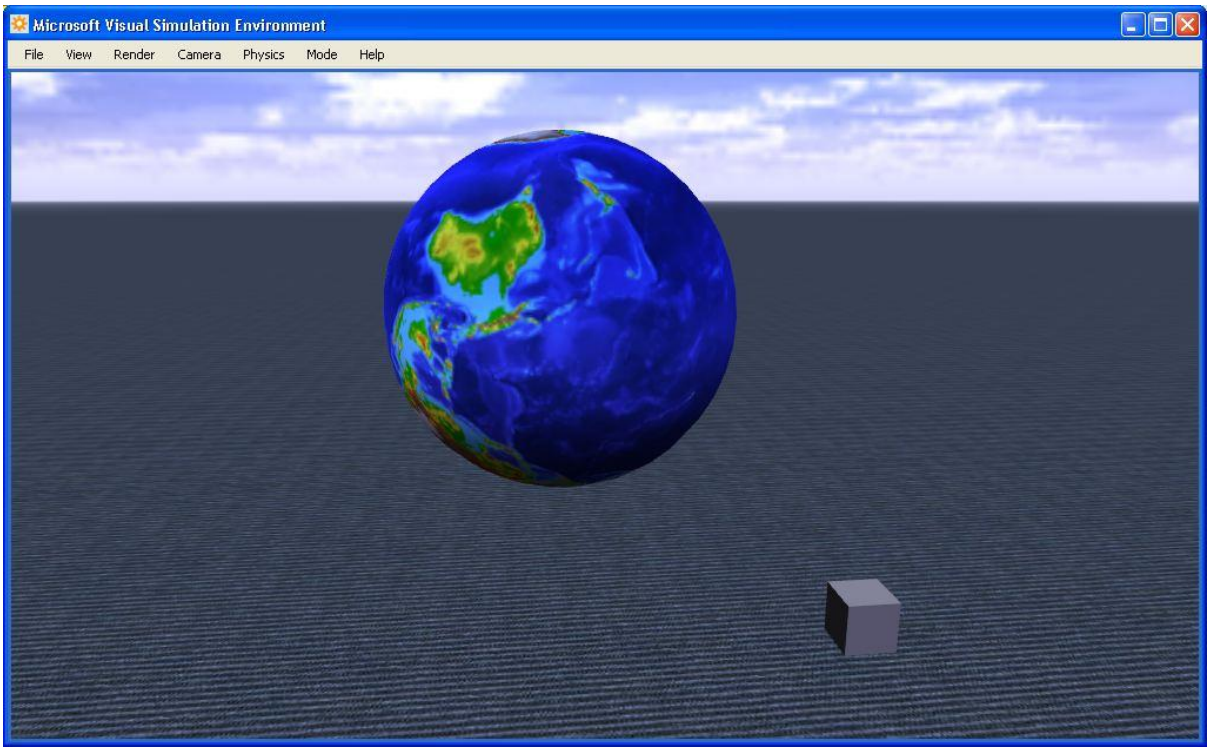


그림 1 - Microsoft Visual 시뮬레이션 환경

시뮬레이션이 실행되고 나면 마우스를 움직여서 카메라의 방향을 움직일 수 있습니다. 단, 마우스를 움직이는 것은 카메라의 방향만 변경시킬 뿐, 카메라 자체의 위치를 변경시키지는 못합니다. 카메라의 위치를 변경시키기 위해서는 W 키를 눌러 앞으로 이동시키거나, S 키를 눌러 뒤로 이동합니다. A와 D 키는 각각 왼쪽과 오른쪽으로 이동시키는 기능을 수행하며, Q와 E 키는 각각 위 또는 아래로 이동시키는 기능을 수행합니다.

파일 메뉴에서 Open 메뉴를 선택한 후 samplesWconfigWLEGO.NXT.Tribot.SimulationEngineState.xml 파일을 오픈합니다.

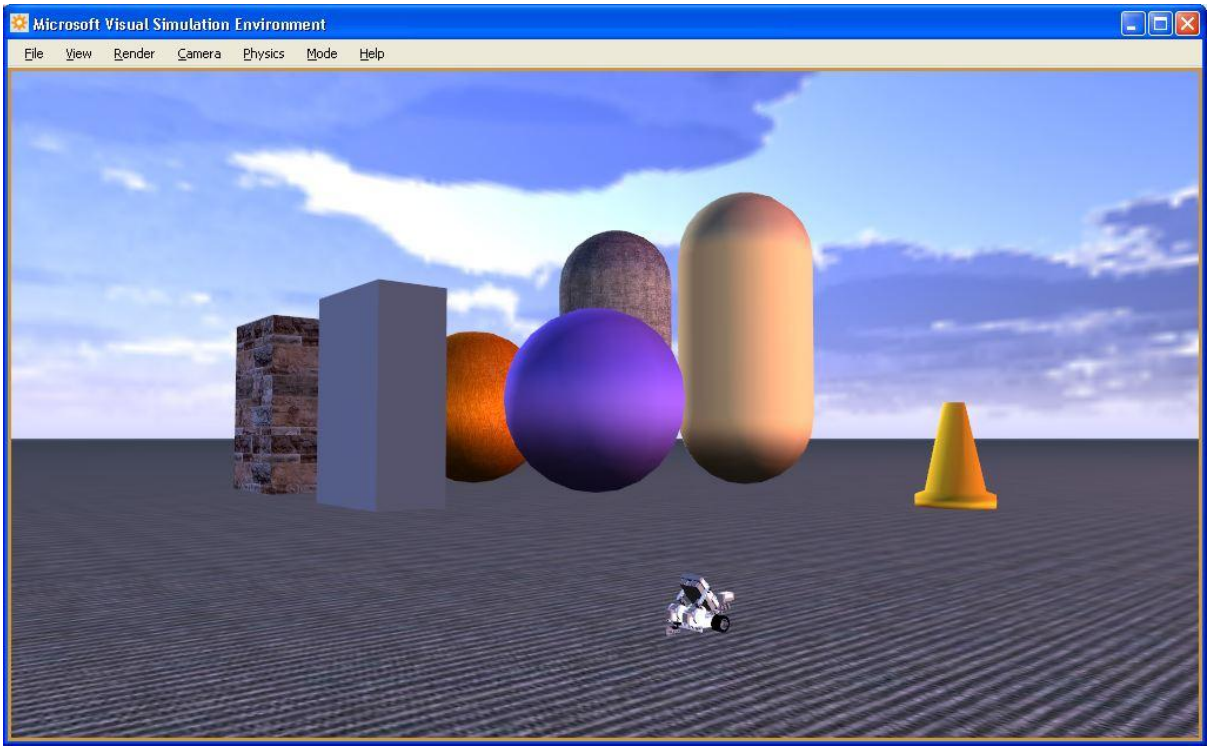


그림 2 - LEGO NXT Tribot 예제를 오픈한 결과

Open Manifest 메뉴는 해당 매니페스트 파일을 실행시키며, Capture Image 메뉴는 현재의 실행 상태를 이미지 파일로 저장합니다. 기타 각각의 메뉴에 대한 대략적인 설명은 시뮬레이션 사용자 가이드 부분에 포함되어 있는 비주얼 시뮬레이션 환경 메뉴 페이지를 참고 바랍니다.

Settings 메뉴에서는 렌더링 설정에 대한 상세한 옵션이 제공됩니다. 이러한 설정항목은 configWSimulationEditor.config.xml에 저장되어 있어서, 다음 번 실행시에도 그대로 적용됩니다.

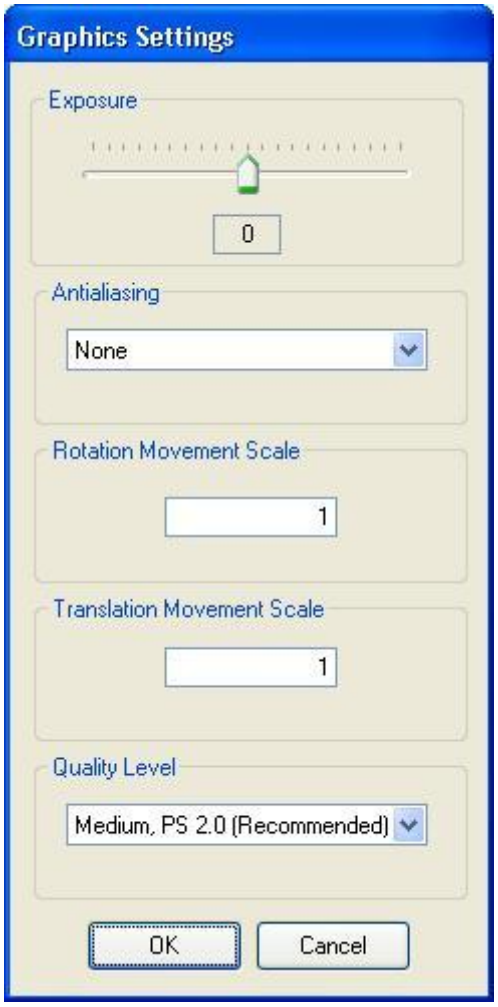


그림 3 - Graphics Settings 다이얼로그

Exposure는 보여지는 화면의 밝기를 조정합니다. 보다 작은 값은 어둡게 만들고 큰 값을 밝게 만듭니다.

그래픽 카드가 anti-aliasing을 지원할 경우, Antialiasing 값을 설정할 수 있습니다. 이러한 모드의 적용은 그래픽 처리 성능을 약간 저하시킬 수 있으나, 아래 그림에서와 같이 적용 후에는 각 면이 만나는 부위가 계단 현상처럼 보이지 않고 좀 더 매끈한 형태로 표시됩니다.

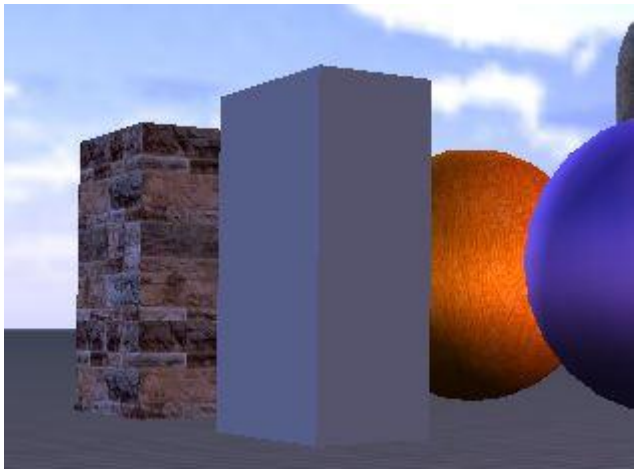


그림 4 - Antialiasing가 지원되지 않는 경우

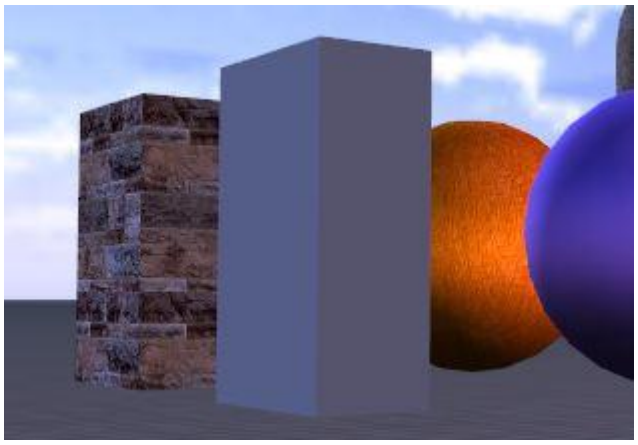


그림 5 - Antialiasing가 지원되는 경우

Rotation Movement Scale과 Translation Movement Scale은 시뮬레이터에서 마우스와 키보드의 반응 속도를 지정합니다. 높은 값일수록 동일한 입력 조건에 대해서 좀더 빨리 인터럽을 발생시켜 더 빨리 회전과 변환이 이루어 지도록 합니다.

Quality Level은 사물들에서의 빛의 처리 상태를 설정합니다. 높은 값을수록 보더 더 정교하게 처리되나, 높은 연산을 요구하므로, [recommended]로 표시되어 있는 항목 보다 높일 경우, 렌더링이 적용되지 않을 수 있습니다. 이 부분은 컴퓨터 그래픽 카드에서 지원하는 Pixel Shader 버전과 연관이 있으며, 버전이 낮을 경우 지원이 되지 않을 수 있습니다.

Physics 메뉴에서는 Physics 엔진을 활성화하거나 또는 비활성화 시킬 수 있으며, F3키에 의해서도 설정이 가능합니다.

Settings... 메뉴에서는 메인 카메라를 하나의 사물로 활용하는 옵션과 중력을 설정하는 기능을 지원합니다. 만약 카메라를 강체로 활용하는 것으로 체크한다면, 메인 카메라는 하나의 강체로 작용하며, 카메라를 이동시켜서 다른 사물과 충돌하게 하여 다른 사물을 움직이게 할 수 있습니다.

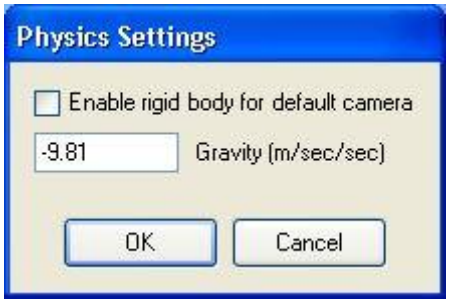


그림 6 - Physics Settings 다이얼로그

또한 중력값을 변경하게 되면, 다양한 현상을 만들 수 있습니다. 예를 들어 우주에서와 같은 무중력 상태를 만들고자 할 경우에는 값을 0으로 설정하면 되며, 반대로 값을 0 보다 큰 값으로 설정하면, 위쪽으로 중력이 작용하여, 사물들이 모두 위로 날아가 버리는 현상을 확인할 수 있습니다.

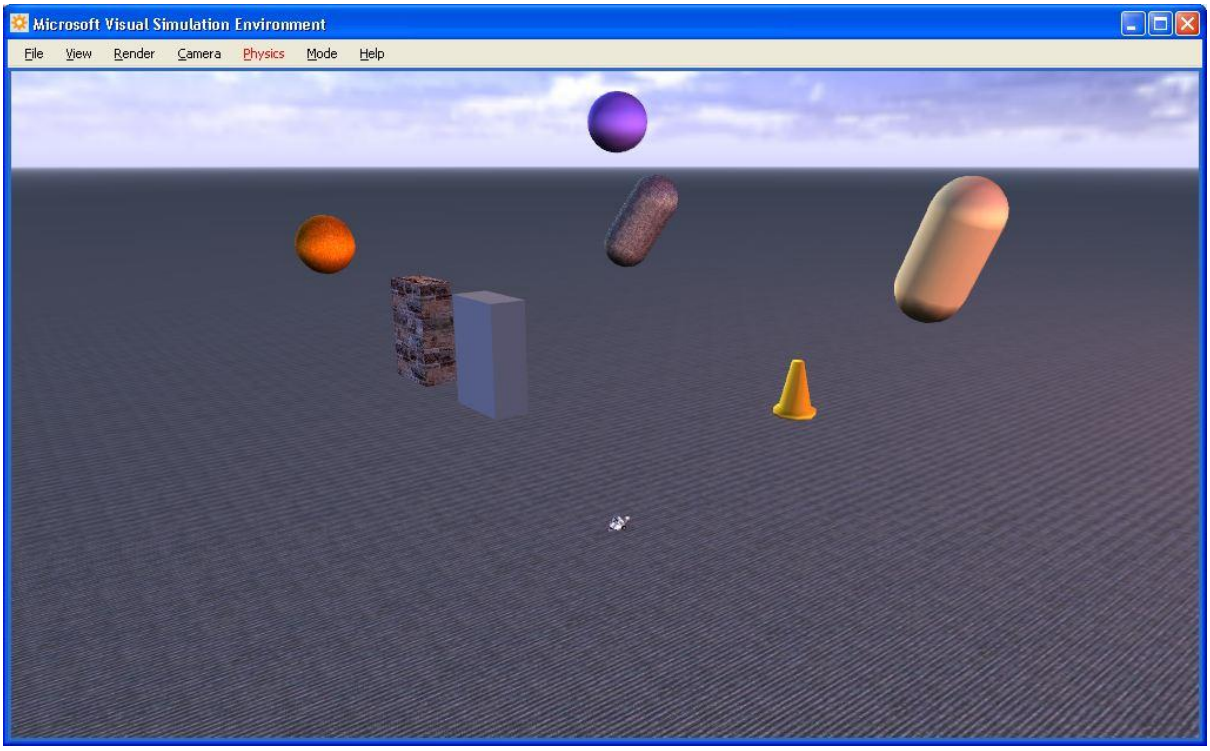


그림 7 - Gravity를 0으로 설정하였을 경우의 무중력 상태

Mode 메뉴는 시뮬레이션의 실행 모드와 편집 모드를 토글로 선택하며, F5 키를 통해서도 수행 가능합니다.

Edit 모드의 실행결과는 아래와 같습니다.

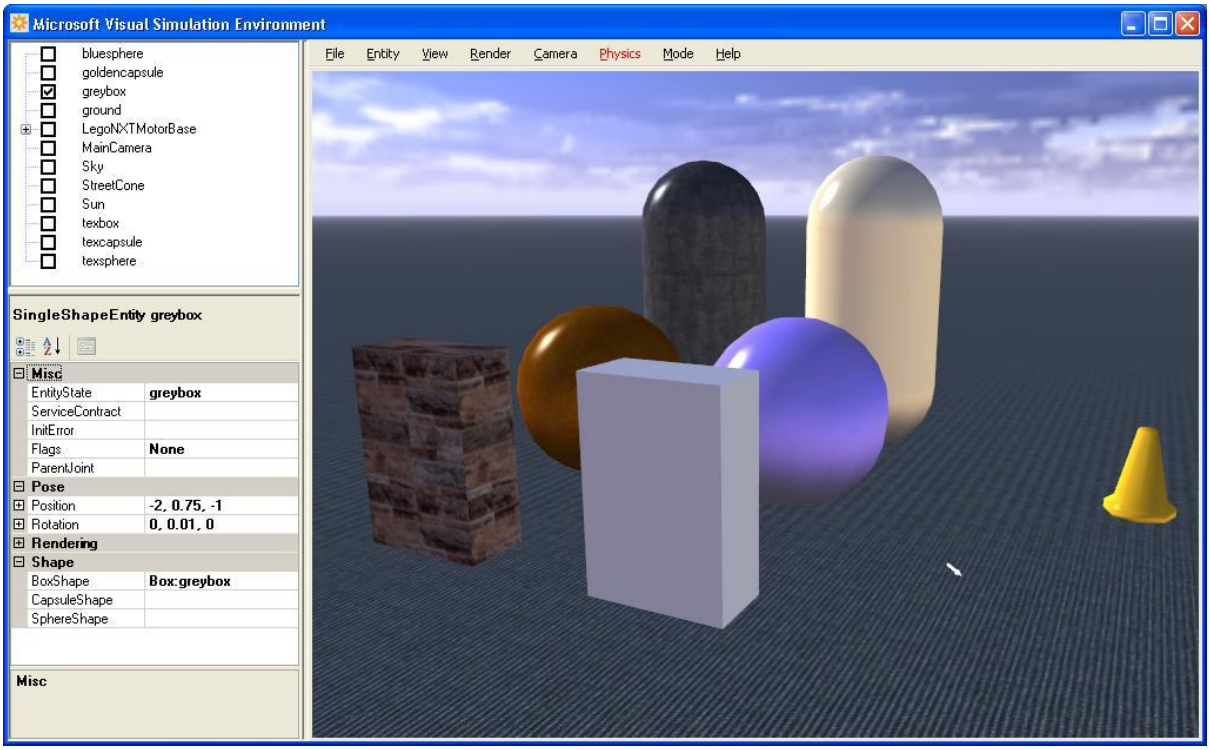


그림 8 - Edit 모드

왼쪽 상단은 엔터티 윈도우 창이며, 왼쪽 하단은 속성 창입니다. 편집모드에서는 각각의 엔터티들에 대해 복사, 잘라내기 또는 붙여 넣기 등의 기능을 수행할 수 있습니다.

또한 이미 설정되어 있는 값도 변경이 가능한데, Sky 엔터티를 선택한 후, 속성 창에서 VisualTexture 항목에 설정되어 있는 텍스쳐 맵을 Directions.dds 값을 선택하여 변경하면 아래와 같이 하늘이 변경된 것을 볼 수 있습니다.

그림 9 - Sky Texture 변경

또한 각 엔터티의 Rotation 또는 Position 항목을 선태한 후, Ctrl 키를 누르고 마우스를 움직여서 해당 엔터티를 직접 원하는 형태로 회전시키거나 옮겨 놓을 수 있습니다. 그리고 Ctrl+C 키를 눌러서 각 엔터티를 복사한 후 Crtl+V를 눌러 붙여넣기 해 놓을 수 있습니다. 또한 크기 값도 변경이 가능한데, 예를 들어 첫 번째 튜토리얼 샘플을 로드한 후, 속성 창에서 BoxShape의 Height 값을 1에서 2로 변경하였을 경우, 아래와 같이 크기가 변경되는 것을 확인할 수 있습니다.

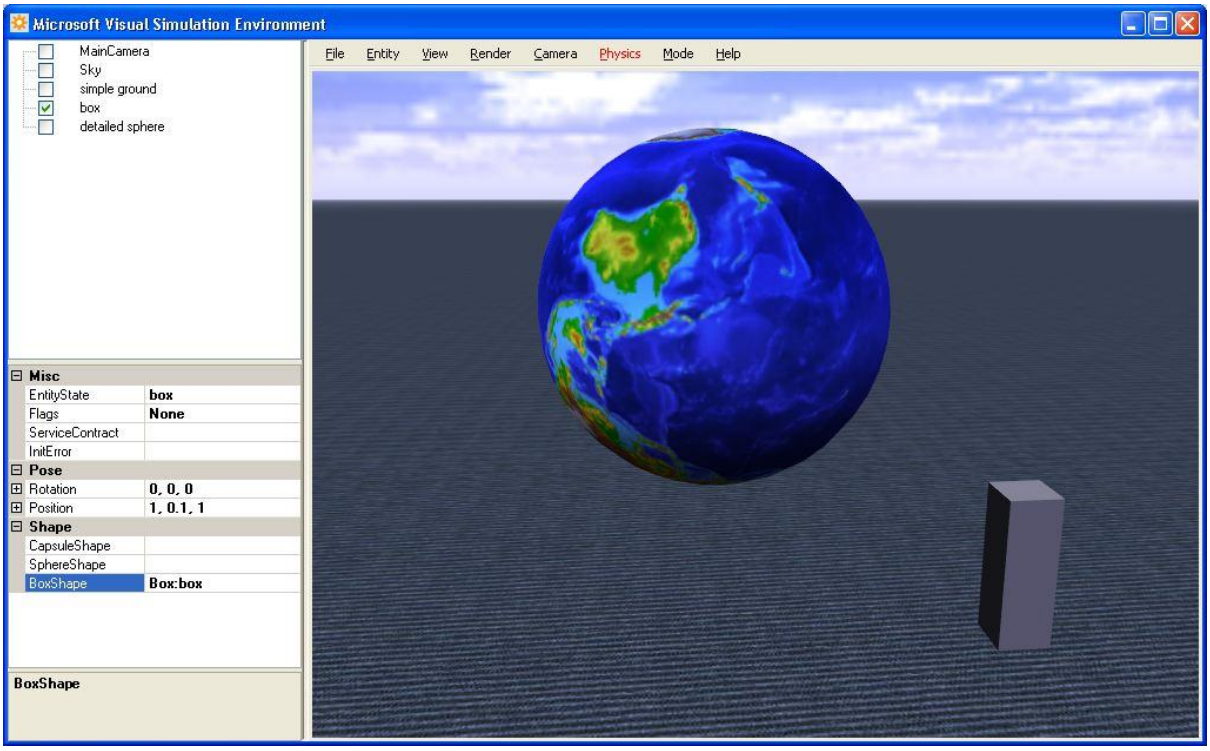


그림 10 - Shape의 속성 변화

Sun과 같은 조명과 관련된 엔터티의 경우에는 빛을 제어하기 위한 매우 세부적인 속성항목들이 존재합니다. Sun은 LightSourceEntity로 생성된 객체이며, LightProperites 안에 세부적인 속성 항목들을 가집니다.

중요한 속성 항목으로서 Type이 있는데, Type 값으로 Omni, Directional, 그리고 Spot 항목이 선택 가능합니다. Omni는 하나의 지정된 지점에서 모든 방향으로 빛을 비추는 옵션이며, Directional은 특정 방향에서 모든 사물에 빛을 비추는 옵션입니다. Spot은 오직 Umbra 또는 해당 Cone 영역에만 빛을 비추는 옵션입니다. Umbra 값이 커지면 Cone의 영역이 넓어집니다. Spot은 또한 position과 rotation 속성을 통해 방향과 위치를 지정할 수 있습니다. SpotUmbra 값을 45로 설정하였을 경우, 바닥에 빛이 비추어 지는 영역이 작아지는 것을 볼 수 있을 것입니다.

또 하나의 유용한 속성으로 CastsShadows가 있습니다. 이 값을 True로 설정하면, 모든 사물에 그림자가 만들어 지는 것을 볼 수 있습니다. 이러한 속성의 설정은 다양한 시뮬레이션 결과를 만들어 내기는 하나 사전에 미리 계산하는 작업이 많이 수행되어 성능 저하를 유발할 수 있습니다.

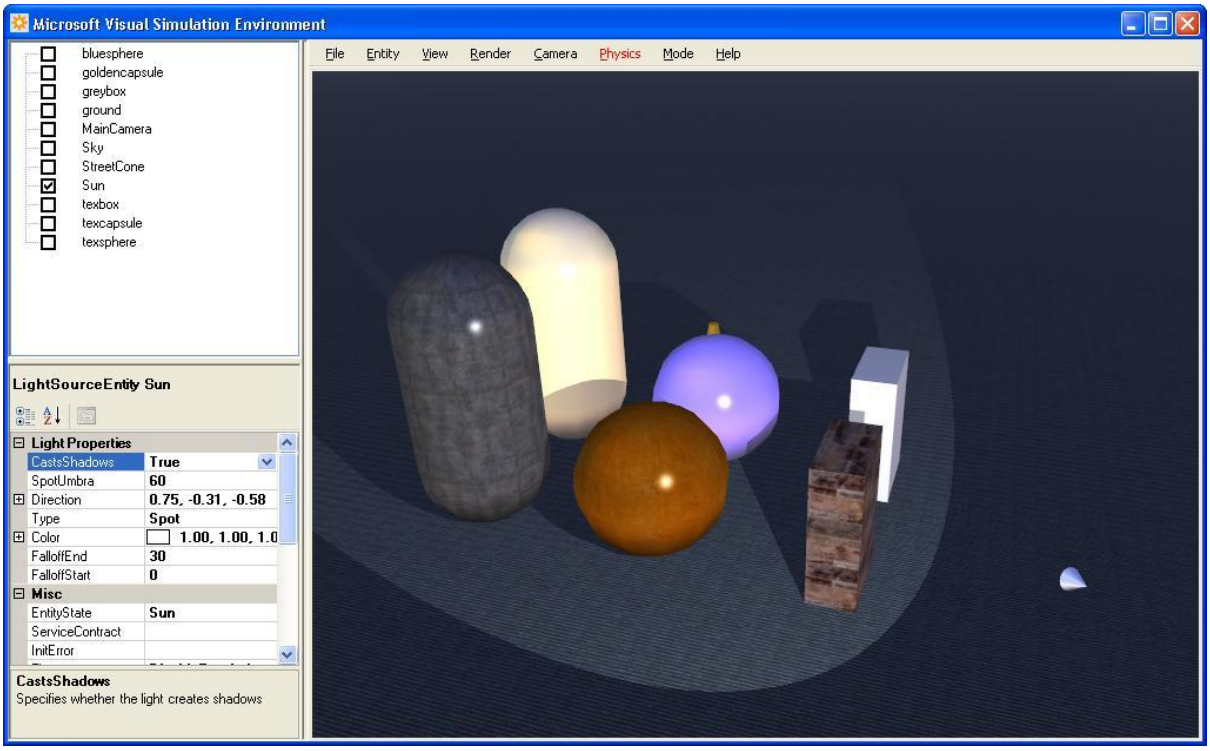


그림 11 - Spotlight에서의 Shadows Cast

시뮬레이션 상의 사물에 대해서 다양한 색상 옵션 또한 세부적인 설정이 가능합니다. 해당 엔터티에 대해 Mesh 항목을 세부적으로 확장 선택해 나가면, Ambient , Diffuse, Specular, 그리고 Power 속성 값들을 볼 수 있습니다. Ambient 값을 1, 0, 1, 1로 설정하면 강한 농도를 가진 핑크색으로 엔터티가 표현되는 것을 볼 수 있습니다.

각 엔터티의 외부 표현 방식을 쉽게 수정하는 방법은 RenderingMaterial에 표시되어 있는 버튼을 눌러 실행시킬 수 있는 material editor를 이용하는 방법입니다. 이 편집기에서는 ambient, diffuse, specular, power (shininess) 속성들을 손쉽게 수정하는 기능을 제공합니다.

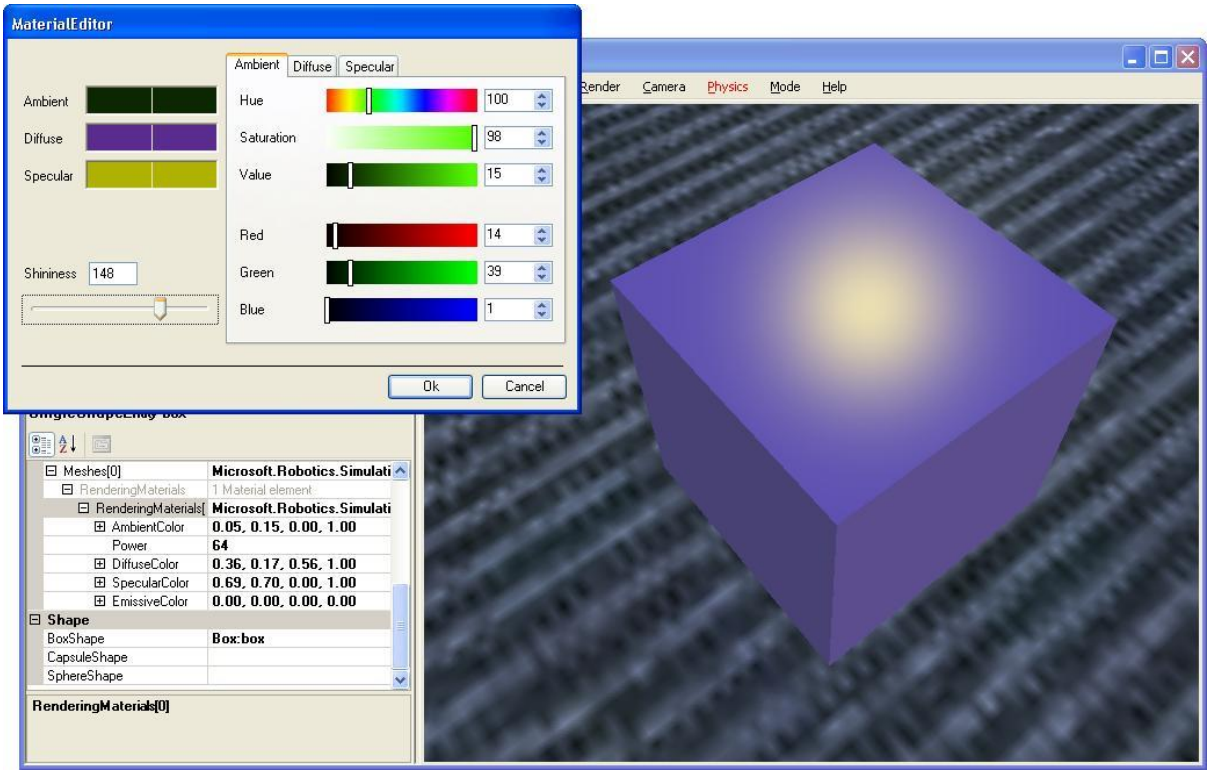


그림 12 - Material editor

시뮬레이션 편집 모드에서는 또한 새로운 엔터티를 추가해 놓을 수 있습니다. Entity 메뉴에서 New Entity를 선택하면 아래와 같은 다이얼로그 창이 보여지며, 적절한 값의 선택을 통해 새로운 엔터티를 추가시킬 수 있습니다.

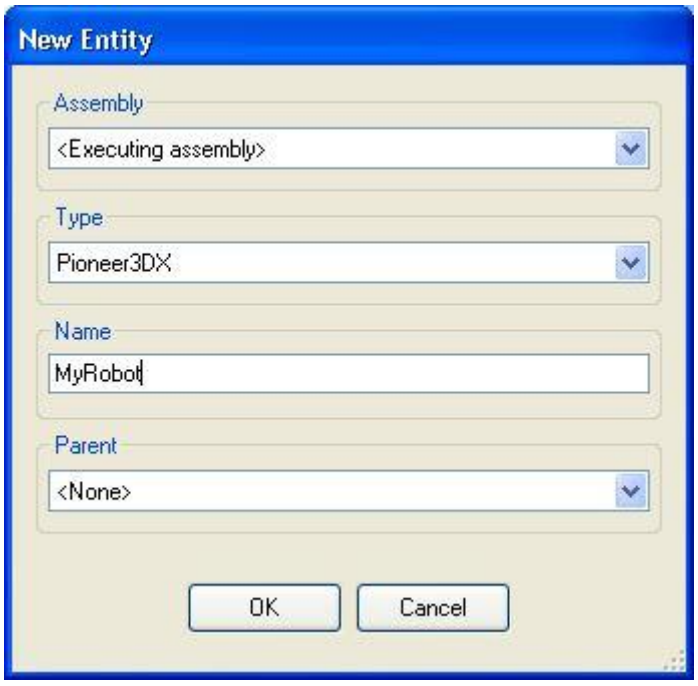


그림 13 - New Entity 다이얼로그

Type에서 Pioneer3DX를 선택하고 이름을 "MyRobot"로 지정한 후 OK 버튼을 클릭하면, 또 Pioneer3DX의 세부적인 옵션을 지정하는 또 다른 창이 표시됩니다. 이 창에서 Position 벡터 값을 0, 0, 0으로 지정합니다.

이번에는 New Entity 메뉴를 다시 실행하여 LaserRangeFinderEntity를 선택하고 해당 이름을 lrf로 지정합니다. 또한 Positon을 (0, 0.3, 0)으로 지정합니다. Lrf 엔터티를 선택한 후, InitError 속성을 확인해 보면, 해당 엔터티가 오직 Child이어야 한다는 오류 메시지를 확인할 수 있을 것입니다. 따라서, Lrf 엔터티를 선택한 후 Ctrl+X를 눌러 잘라내기 합니다. 그리고 앞서 생성한 Pioneer3DX 로봇을 선택한 후 붙여넣기 하면, 자동으로 Lrf가 해당 로봇에 Child로서 추가됩니다.

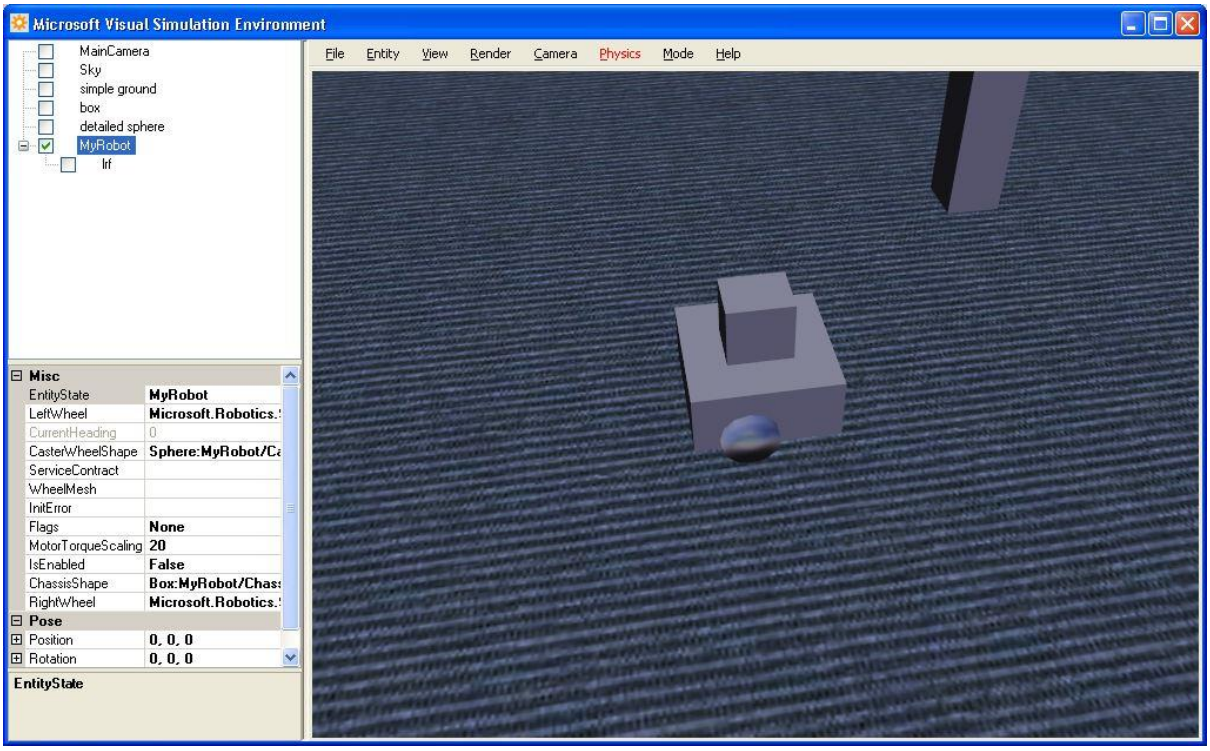


그림 14 - Pioneer3DX에 추가된 Laser Range Finder

기타 다른 속성들에 대해서도 프로그램 개발 작업에서 지정되는 것과 동일하게 값을 지정하여, 직접 코드로 개발한 것과 동일한 작업들을 수행할 수 있습니다. 즉, ServiceContract 등의 속성 등을 SimulatedDifferentialDrive 값으로 지정함으로써, SimpleDashboard 등을 통하여 로봇을 직접 제어할 수 있도록 할 수 있습니다.