LVC for Unity

Quick Start Guide

Document Version	0.3
Prepared For	Calytrix Technologies
Prepared By	Andrew Laws
Date	26 Aug, 2013
Classification	Commercial in Confidence

Calytrix Technologies Pty Ltd
Level 2, 110 William St
Perth, 6000
Western Australia
Web: http://www.calytrix.com
Phone: +61 8 9226 4288
Fax: +61 8 9226 0311

Document Revision History

Date	Version	Comments	Author
13 Jun, 2013	0.1	Initial Draft	Andrew Laws
28 Jun, 2013	0.2	Draft Revision	Michael Huynh
26 August, 2013	0.3	Minor Updates	Andrew Laws

Table of Contents

Table of Figures

1 Introduction

This document is a guide to getting started with your first Calytrix LVC Game for Unity project.

1.1 Audience and Purpose

This document should be read by those who are just starting out with the Calytrix LVC for Unity package.

Ideally the reader should already have some familiarity with the DIS and/or HLA simulation protocol, and a working knowledge of the Unity Game Engine.

1.2 Background

The LVC for Unity package provides a mechanism for developing Unity games which can be integrated with existing DIS and HLA simulation environments.

Within this document, the terms “simulation” and “game” will be used more or less interchangeably. In most senses, a simulation is just a “serious game”; indeed the term “serious game” is often used when describing a system which may have begun its life as a game, but has been repurposed or redesigned for training purposes.

1.3 Document Structure

This document is structure such that, by following the steps in order, the reader will create a basic LVC compatible game within the Unity Editor.

Section 2 covers some very important information that you have included because it’s very important.

Section 3 discusses more important information that logically follows, enhances and extends that which was presented in the previous section.

Finally, the conclusion ties everything together into a nice, neat little package that can’t help but to leave your reader stunned at your mastery of both phrase and structure.

2 Overview

Before creating the basic LVC Game Unity project, it may be useful to carry out a very high level overview of a hypothetical distributed simulation.

2.1 The Chess Game

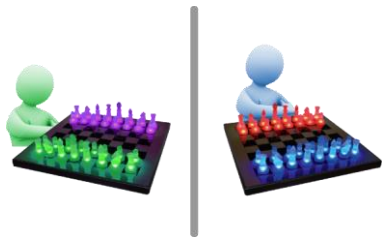
Imagine two people, Qiang and Albert, who wish to play a game of chess together. Life is never easy, though, and there are a few problems.

The first issue is that Qiang lives in China, and Albert lives in France.

Also, Qiang speaks only Mandarin, and Albert speaks only French.

On top of this, while Qiang's chess board pieces are purple and green and styled after mythological creatures like dragons and gryphons, Albert's pieces are red and blue and styled after Harry Potter characters.

Finally, Qiang likes to play chess looking at the board from the side, rather than from the more traditional seat behind his pieces.

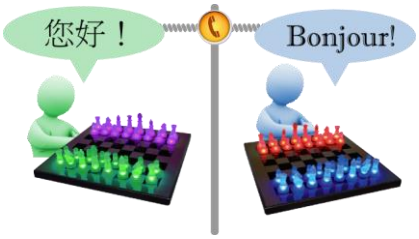


Obviously we need to overcome these issues in order for Qiang and Albert to play their game. Let's take a look at how we might go about this.

Communication:

In order to play the game together, Qiang and Albert need to be able to communicate with each other somehow.

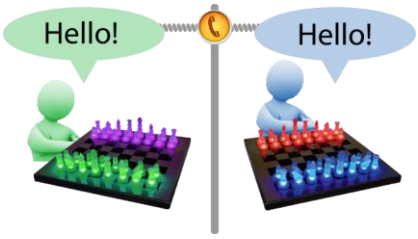
Let's give them a call conferencing service so that they can talk to each other over the phone.



Language:

In order to tell each other their moves over the phone, Qiang and Albert need to be speaking a language they both understand.

Let's say that they both agree to learn and speak English so that they can understand what each other is saying.



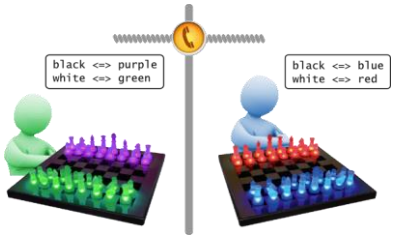
Pieces:

Qiang and Albert have different shaped and coloured pieces, and so if Qiang talks about moving a “Purple Dragon”, Albert has no idea what he means.

Since a standard chess board is black and white, and uses the Rook, Knight, Bishop, King, Queen and Pawn pieces, they agree to convert their own colours and pieces to this common standard before telling the other person their move.

For example, when Qiang moves a “Purple Dragon” piece he will standardise it to “Black King” before telling Albert. A “Green Gryphon” will be “White Bishop”, and so on.

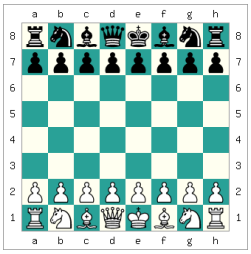
Likewise, Albert will tell Qiang he is moving “Black King” when he moves “Blue Wizard”, “White Rook” when he moves “Red Owl”, and so on.



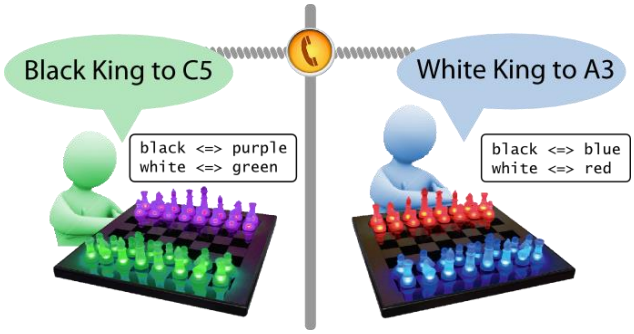
Board:

The two players are looking at the board from different angles – left, right, forward and backward from Qiang’s point of view have different meanings to Albert’s point of view. If Qiang says “Move the black king left 1 space”, Albert will move the piece to the wrong square on his board.

To solve this they agree to use the standard chess coordinate system with the letters A-H across the breadth of the board, and 1-8 across the length, with A1 being the “origin” at the white player’s leftmost rook.



Now that Qiang and Albert have resolved their problems, they can begin to play their game.



One can see that this example could be modified from this simple two player game of chess to a multiple player game such as monopoly.

Again, the same steps would need to be taken to allow communication with a standardised language, an agreement on standardized names for pieces (a Star Wars Monopoly set uses an R2D2 instead of a shoe) and coordinates (a Star Wars Monopoly set uses an Degobah instead of Coventry Street).

Playing the Same Game:

One final issue that has not been dealt with in the example so far is the game itself – there is an implicit assumption throughout the above examples that *all* players are either a) playing the exact *same* game, or b) playing games similar enough that the differences don't affect the overall gameplay.

Clearly, if we attempted to connect two people to play a game together, and one played Chess against the other playing Monopoly, the result would not make much sense.

Video games can be used to provide some other relevant examples. Connecting two "Call of Duty" games together makes complete sense. Connecting "Call of Duty" with another first person shooter game such as "Halo" makes less sense, but since they share some equivalent ideas (soldiers, vehicles, guns…) they *might* be made to work together with some effort and compromising on some rules of play. Connecting "Call of Duty" to a racing game such as "Forza", however, makes virtually no sense at all.

The Shared World:

With all players connected, there is a sense in which they are no longer playing separately in different locations, but in the same imaginary space or world.

Obviously their individual "local" experiences are slightly different - Qiang sees purple and green pieces, Albert see red and blue pieces, and so on.

For all practical purposes, however, they are now sharing the same board and same pieces sitting directly across the board from each other. It's just that they see this shared board through the "window" of their own board.

2.2 Terminology

Unity

Unity (also called Unity3D) is a cross-platform game engine with a built-in IDE developed by Unity Technologies. It is used to develop video games for web plugins, desktop platforms, consoles and mobile devices.

Simulation / Game:

In the context of this document, a simulation or game is an interactive software program that allows individuals to learn and practice real world activities in a realistic way within a computer generated environment.

Distributed Simulation / Networked Game:

A distributed simulation or networked game is a simulation or game carried out across multiple host computers over a communications network. Each computer hosts an individual simulation or game, and these transfer information in an agreed format with the others over the network to create a shared "world".

The shared world does not exist exclusively within any one of the participating computers, but is a result of the shared and combined information. The shared world is viewed and interacted with through the "window" of the individual simulations or games on each computer.

Simulation Backbone

The simulation backbone is the means by which separate simulation systems share information with each other. Usually this is a LAN (local area network).

In the context of the Qiang/Albert example, the simulation backbone is equivalent to phone conferencing service.

DIS and HLA

Distributed Interactive Simulation (DIS) is an IEEE standard for conducting real-time platform-level wargaming across multiple host computers and is used worldwide, especially by military organizations but also by other agencies such as those involved in space exploration and medicine.

High-Level Architecture (HLA) is a general purpose architecture for distributed computer simulation systems. Using HLA, computer simulations can interact with other computer simulations regardless of the computing platforms. The interaction between simulations is managed by a Run-Time Infrastructure (RTI). HLA is an interoperability standard for distributed simulation used to support analysis, engineering and training in a number of different domains.

In the context of the Qiang/Albert example, using DIS or HLA is equivalent to the agreement to use English when communicating with each other.

Entity Instance, or Entity

An entity is an actualised entity type - an individual "thing" which is part of the simulation or game.

In the context of the Qiang/Albert example, each of the 32 pieces on the board is an individual entity. As an example, each of the four Rooks at the corners of the board is an entity instance of the Rook entity type.

Entity Type

The entity type is exactly what it sounds like – the type of an entity. Examples might be 'tank', 'missile', 'soldier', and so on.

In the context of the Qiang/Albert example, there are 6 "standard" entity types: Rook, Knight, Bishop, King, Queen and Pawn which are used to communicate moves.

There are also "internal" entity types which Qiang and Albert use for their own boards – dragons, gryphons, wizards, owls and so on. These entity names are, however, only useful for Qiang and Albert individually – if Albert tells Qiang about a wizard, for example, Qiang won't know what he means.

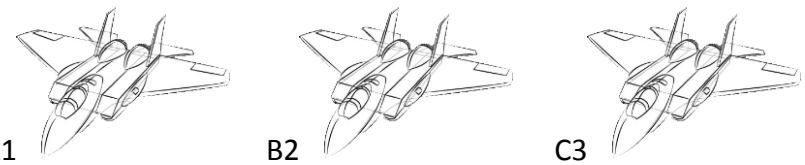
Entity Marking:

Entity instances are identified by a "marking", which is simply an alias or a call sign used for identification. The marking is used to distinguish between entity instances of the same type.

For example, in the Qiang and Albert's chess game there are 8 white pawns, so saying "move White Pawn to A3" is not enough information – which of the 8 white pawns should be moved? We can solve this problem by assigning a different entity marking (perhaps a number or letter) to each white pawn. We can then specify *which* white pawn we mean by providing the marker as well as the type; for example "move White Pawn-5 to A3", or "move White Pawn-2 to A3".

In order to identify a particular entity instance, the marking *must* be unique within the entities of that type.

For example, imagine there are three tank and three jet entity instances. We give the five tanks the markings 'A1', 'B2', 'C3'. We give the jets the markings 'A1', 'B2', 'C3' as well[1].

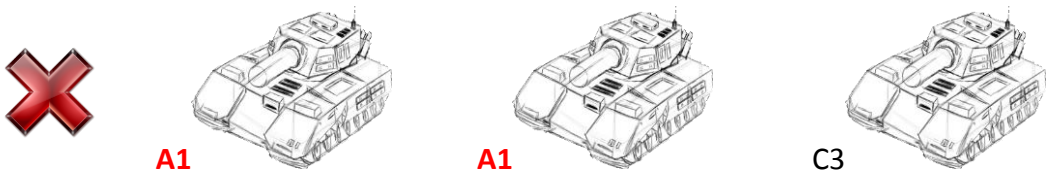


[1] Markings are generally more meaningful than the ones in this example – the call sign of an entity is commonly used as its marking, for example.

We can identify a particular entity by using the Entity Type and Entity Marking in combination – “tank-A1”, or “jet-C3”, for example.

As can be seen, there is no problem using the same marking on Entity Instances of *different* types, as we are still able to uniquely identify the particular Entity. For example the “tank-C3”, and “jet-C3” entity instances both have the same “C3” marking, but are of different types, so there is no confusion.

If we used the same marking more than once on the *same* Entity Type, however, things become confusing. For example, if we use the “A1” marking on two of the three tanks, like so…



…and refer to “tank-A1”, there are two possibilities, and we can no longer determine which specific tank is being referred to. This defeats the purpose of markings, and this situation should always be avoided.

Entity Type Enumeration, or DIS Enumeration

An enumeration is a standardised ‘code’ used by DIS and HLA to identify the *type* of an entity when communicating over the network.

For example, an American M1A1 Abrams tank would be represented by the following entity type enumeration[2]:

`1.1.225.1.1.2.0`

An individual simulation may refer *internally* to the type of an M1A1 tank entity however it likes; ‘m1a1’, ‘abrams’, ‘tank’, or some other descriptor.

However, when it transmits information about that type over the DIS/HLA network it *must* use the standard entity type enumeration.

In the context of the Qiang/Albert example, this is similar to their describing the pieces using the standard “Black King”, “White Bishop” notation over the phone, even though their own boards use “Purple Dragon”, “Green Gryphon” or “Blue Wizard” and “Red Owl”.

[2] For a full list of all standard enumerations, please refer to the latest version of the Simulation Interoperability Standards Organization (SISO) document “Enumerations for Simulation Interoperability”, available from www.sisostds.org. At the time of writing, this document is SISO-REF-010-2011.1.

Entity Type Mappings:

Entity type mappings are a way of connecting the local game types with the enumerations being sent and received over the network. Simply put, it is a lookup table which says that A=X, B=Y, C=Z, and so on.

In the context of the Qiang/Albert example, Albert's mapping table might look like this:

Internal Name	Standard Enumeration
Blue Wizard	`Black King`
Red Owl	`White Rook`
…	…

When information is received over the network that an entity with enumeration "Black King" has moved, Albert looks up the entry in the mapping table to find out that he needs to move his "Blue Wizard".

Similarly, when Albert moves his "Red Owl", he uses the lookup table to tell Qiang the standard name "White Rook".

If there is no mapping for a type, there is a problem. For example, if Qiang tells Albert he is moving a "Black Tiger", there is no corresponding mapping entry, so Alberts no way to determine what he should move. The only real solution in such a case (without updating the mappings) is to ignore the information and/or note that there was an error in communication during the game.

2.3 The Distributed Simulation

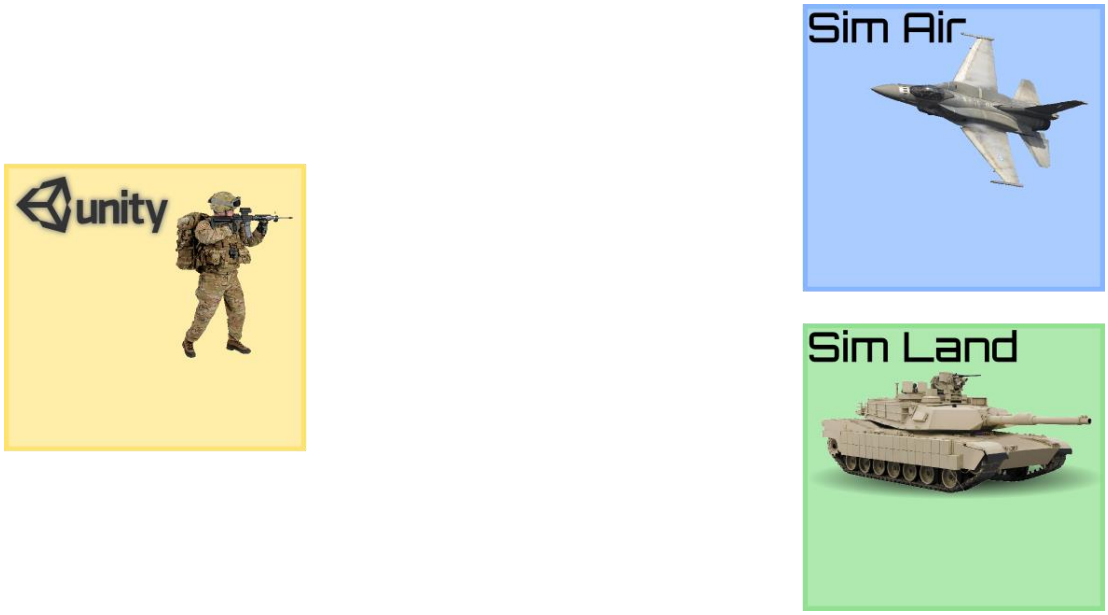


Figure 1: The Simulators

Figure 1 shows 3 simulators, each running separately: a first person shooter game (Unity) a flight simulator (Sim Air), and a tank simulator (Sim Land).

Each is fully capable of tracking the details of the “entity” (vehicle or player) it is simulating, and can be operated, or played, independently.

It is not possible, however, for the simulators to ‘see’ what is happening in any of the other simulators. For example, the pilot flying the flight simulator cannot see or interact with the tank or soldier.

It would be useful to connect them all together so that each simulator can see and interact with the entities in the other simulations.

This is achieved by connecting the simulators over a network, and having them “talk” to each other.

However, each system simulates a different type of world - a flight simulator cares about very different things to a tank simulator or a first person shooter. Because of this, they speak in different languages, and about concepts which the other simulators may not even be aware of.

The following are some very simple examples with regard to our 3 simulators:

- The tank simulator may not understand the concept of “altitude” in the way that a flight simulator does, because tanks are always stuck to the ground – the concept of “flying” is not something it is aware of.
- The first person shooter may not understand what a tank simulator means when it says that a tank’s engine has started up. This is because soldiers don’t have engines, and so is not something it has been programmed to keep track of.
- The flight simulator may not understand what a first person shooter means when it says that a soldier is kneeling, because aircraft cannot change their posture.

DIS and HLA are the most commonly used protocols used in “serious game” military and commercial simulators. They provide a common way of talking about the type of an entity, its position, speed, state of damage, and many other details important to simulation.

Simulators which are able to speak these protocols can be connected together, and share details of the worlds they are simulating with each other.

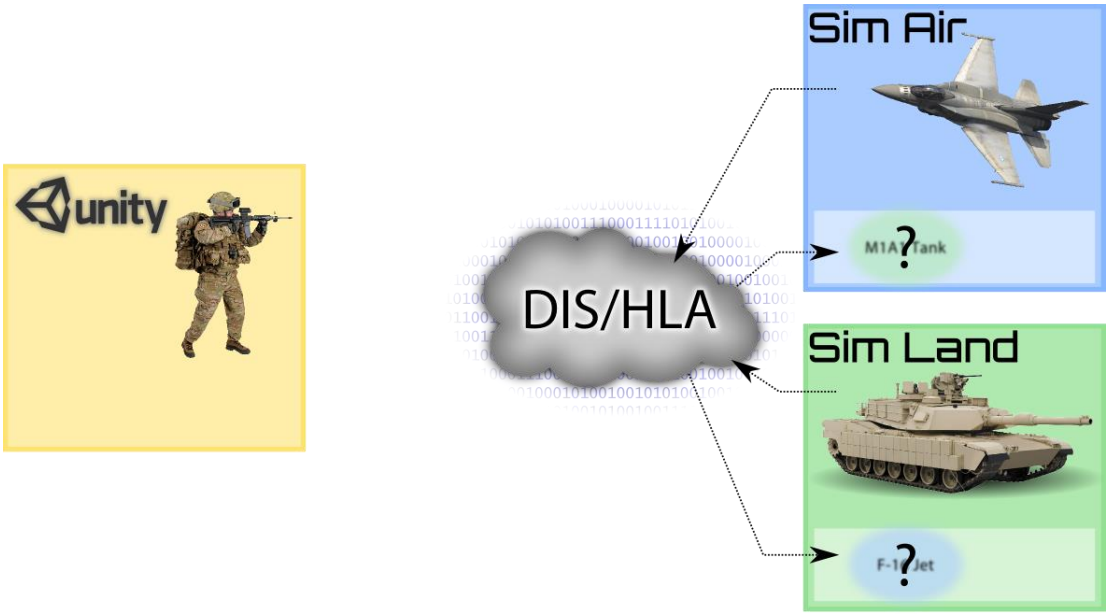


Figure 2: Connecting with DIS/HLA

In Figure 2, we see that Sim Air and Sim Land are now connected and talking across the network using either DIS or HLA.

Because of this common language, Sim Air now understands that Sim Land is providing details about a "M1A1 Tank[3]" entity which it is simulating. Likewise, Sim Land understands that Sim Air is talking about an "F-16 Jet[3]" entity.

There is an additional problem, however. Although Sim Air is able to hear about the "M1A1 Tank" from Sim Land, because it is a flight simulator it may not actually know what an "M1A1 Tank" is. Similarly Sim Land may not know what an "FA-16 Jet" is.

Sim Air could simply ignore this unknown item, but if it does that there's not much point in connecting the simulators together.

So, at this point we need to make an assumption that Sim Air has something it can use to represent "M1A1 Tank" Entity of Sim Land.

Let's say Sim Air has a generic "tank" entity which is usually used for scenery in the flight simulator. This is probably good enough to represent the "M1A1 Tank" for our purposes, so we create an entity mapping entry for Sim Air:

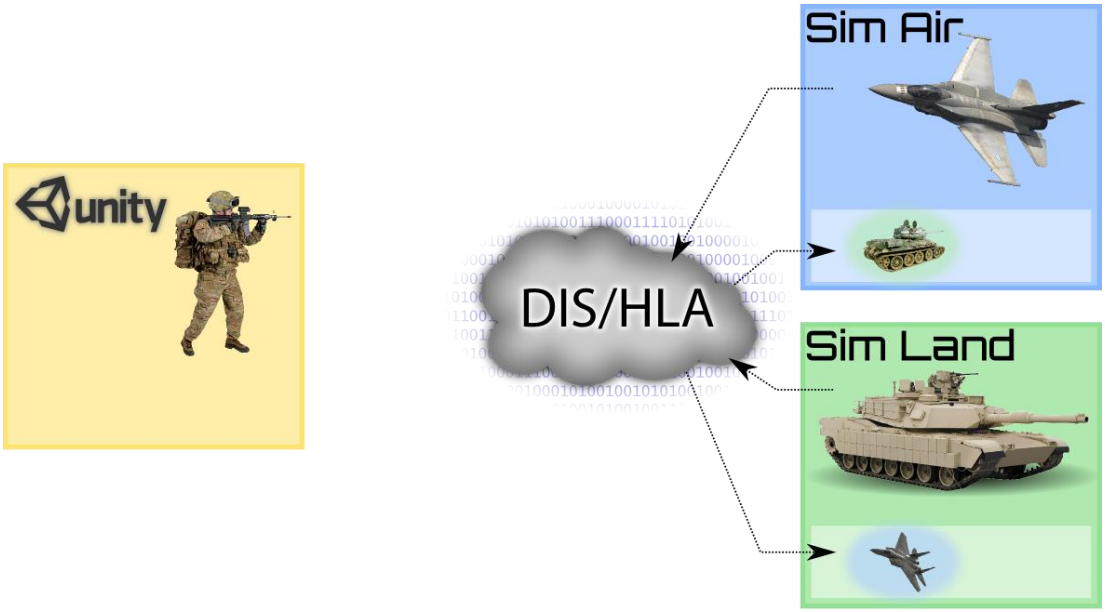
M1A1 Tank = tank

Similarly, we will assume that Sim Land has a generic aircraft called "plane" which can be used to represent the "F-16 Jet" Entity of Sim Air. We create the mapping for Sim Land so that…

F-16 Jet = plane

…and restart both simulators with these mappings.

[3] This would actually be a standard DIS enumeration, such as `1.1.225.1.1.2.0`, but we substitute with more readable examples here.



Notice that, because we are using “best match” Entities, the view of Sim Land from Sim Air, and vice versa, are slightly different to each other.

The compromises are close enough for the overall simulation to be unaffected, however.

We now have two of the three simulators talking to each other. We’d like to get Unity connected as well, but it cannot use DIS or HLA, so how do we connect it to the other simulators?

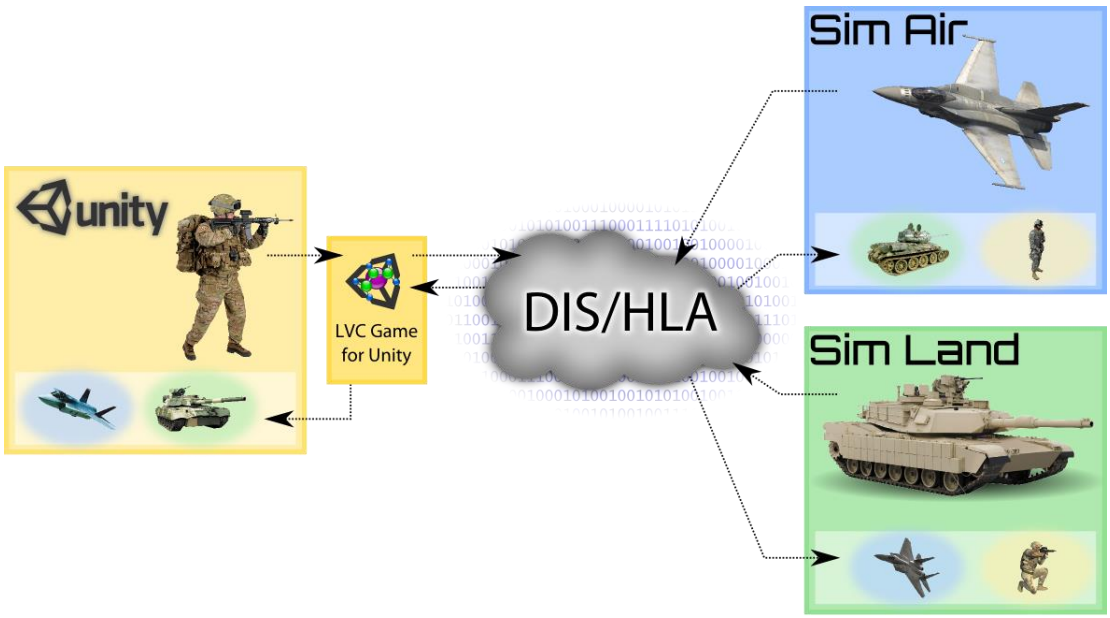


Figure 3: LVC Game for Unity

LVC Game for Unity provides the answer. By using the LVC Game for Unity plugin, appropriately configured Unity games can be made to send and receive information over a network using the DIS or HLA protocols, thus allowing them to talk to other similarly enabled simulators.

Note that there may again be some compromises on the mappings about what a "Soldier", "F-16 Jet" and "M1A1 Tank" is in each of the simulations when we introduce Unity. However, once the mappings are organised, Sim Air, Sim Land and Unity can now all share information with each other about the entities being simulated.

Conceptually, we can now imagine that there is a shared world created by the combination of the original simulators. This shared world can now, in principle, be seen through the "window" of each of the simulators.

Figure 4: The Simulator Views of the Shared World

2.4 Other Considerations

There are a few other considerations which, for simplicity, we have skipped over until this point. These are now worth considering.

Different Coordinate Systems:

Different simulators may not use the same coordinate system.

A common disparity is that one simulation uses latitude, longitude and altitude, and the other uses (X, Y, Z) coordinates with (0, 0, 0) being the centre of the Earth. There are, of course, other coordinate systems which might be used.

Even in the case that two systems use an (X, Y, Z) coordinate system, they may disagree on the orientation of the axes. While some systems use the Y axis for up/down, others may use the Z axis for this purpose.

Because of this it is often necessary to convert from one coordinate system into another when sending or receiving positional information.

Different Locations:

The separate simulations may be configured to take place in entirely different locations, even once coordinates are converted.

If we simply connect games together, players in one game might never see the players in the others because they are too far apart in the simulation.

One solution is to “move” one or more of the game maps so that they can overlap. For example, instead of the centre of the map being at (0, 0, 0), we adjust it to (200, 200, 0) to match the location of the other maps.

Another solution is to create a new map for each game, all of which are in the same location.

Different Terrain:

Simply matching up the coordinates does not necessarily mean that the terrain will match. There will be hills in Sim Land’s map which are not in Sim Air’s map. There will be missing trees, extra lakes, different buildings, and so on.

This problem is related to the previous one, except that rather than the location, the *shape* of the terrain does not correspond.

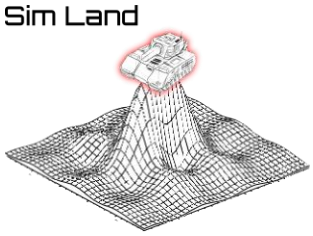
This may mean that an aircraft which is in open sky in Sim Air would appear in Sim Land to be flying through a mountain. A tank which is on a hill in Sim Land may appear to be floating in the air in Unity. A solider hiding inside a building in Unity may be standing out in the open in Sim Land.

This is probably the most common, and potentially the most difficult problem to resolve when connecting games together. This is especially so if the simulations are not all running on the same system.

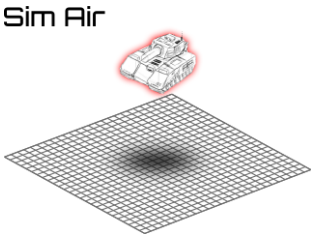
The only real solution is to create an identical map for each connected game, and this should be aimed for as an “ideal” solution.

If this is not possible, it may be feasible to compromise in some ways, such as by using “ground clamping”. Ground clamping is a method of ensuring that a ground based entity, such as a tank, is forced onto the ground during the game.

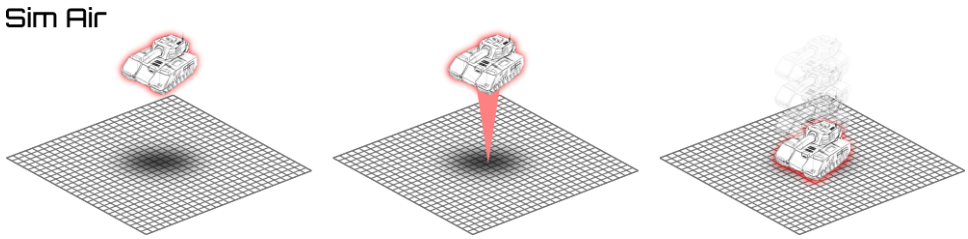
For example, imagine Sim Land tells Sim Air that a tank is at a certain latitude and longitude, and at 50m above sea level. In Sim Land, the tank is 50m above sea level because at that position the tank is sitting on top of a hill.



However in Sim Air the terrain does not have a hill at that position, so when the tank is positioned at those coordinates it appears to be floating in the air.



Sim Air could “ground clamp” the tank by determining the ground height of its own terrain model at that location, ignoring the altitude provided by Sim Land. It then re-positions the tank on the ground.



This means that the tank will look more realistic in Sim Air because it is on the ground, but the positioning of the tank in Sim Air is no longer entirely accurate.

This could cause problems however. For example, when shooting at the tank, where should a gun aim? At the original location provided by Sim Land, or at the adjusted location in Sim Air? Or is there an ongoing reconciliation whereby the coordinates are juggled back and forth across the simulations?

Sometimes compromises are unavoidable – due consideration should be given to their consequences, and the possible impact on the effectiveness on the simulation.

Different Weapons and Units:

There is no guarantee of a one-to-one entity relationship between simulations, and simply connecting simulations together does not “provide” entities to the other simulations.

One solution is to ignore the entities for which there is no exact match, and not display them at all. For example, Sim Land could ignore the “F-16 Jet” entity entirely.

For obvious reasons, this is not always a viable solution – missing out the aircraft in our 3 simulator example would significantly affect gameplay, and even defeat the purpose of connecting them together in the first place.

However if the entity is less significant, this may be an acceptable solution – for example, a “bird” entity in Sim Land could probably be ignored in Sim Air without affecting the simulation noticeably.

Alternatively, we could display an “equivalent” entity. In our example, we chose to display a generic tank in Sim Air.

If there had been no tank, we could have perhaps used a truck instead, or even a rock. Mappings allow the ability to substitute anything, though there is a trade-off in realism if the substitute is not a close match – substituting a bird for an aircraft may not be acceptable, for example.

Using equivalent or substitute entities does have the advantage of requiring no modification to the games themselves, however.

Another solution is to update the underlying game code and models of the simulations themselves so that they have the exact model required it.

If this were done, when Sim Land advertises that there is an M1A1 in play, we could display one in Sim Air and Unity too. This is an “ideal” solution, since all sides will now see the same unit.

The disadvantage of this solution is the amount of work involved in making this change to the game. Indeed, in many situations, modifying the game in such a fundamental way may not even be possible.

3 My First LVC for Unity Game

Let's create a simple LVC enabled game in Unity using the LVC Game for Unity plugin.

3.1 Microsoft Visual C++ 2010 Redistributable Package

The Microsoft Visual C++ 2010 Redistributable Package is required for the LVC Game for Unity Plugin.

Note that if the Microsoft Visual C++ 2010 Redistributable Package is already installed, it does not need to be re-installed.

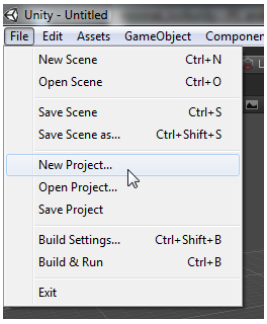
The installer for this package is available from Microsoft at the following URL:

`https://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=5555`

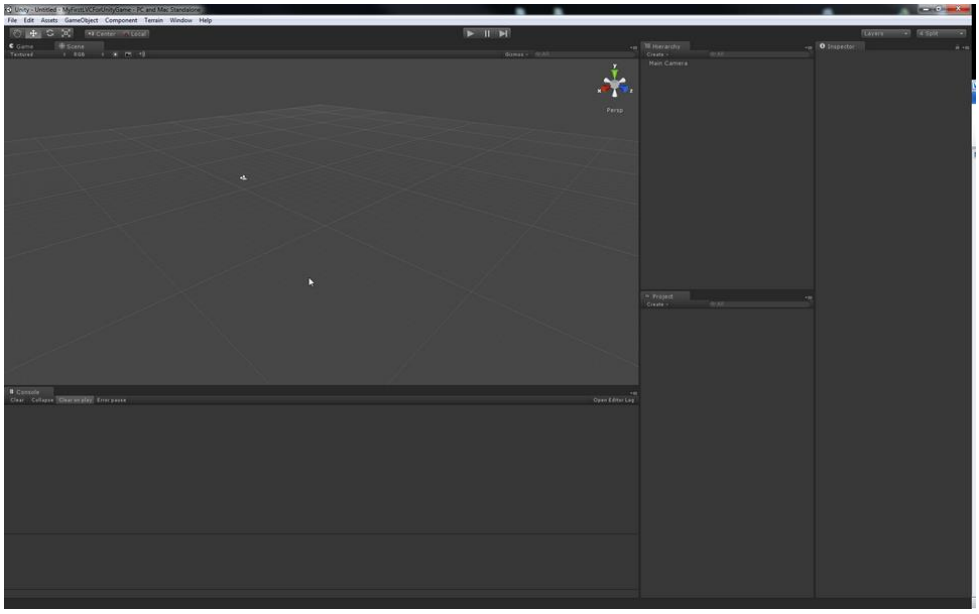
Download the installer and run it, following the prompts to complete the installation of the package.

3.2 Start a New Project

From the `File` menu, select `New Project`:

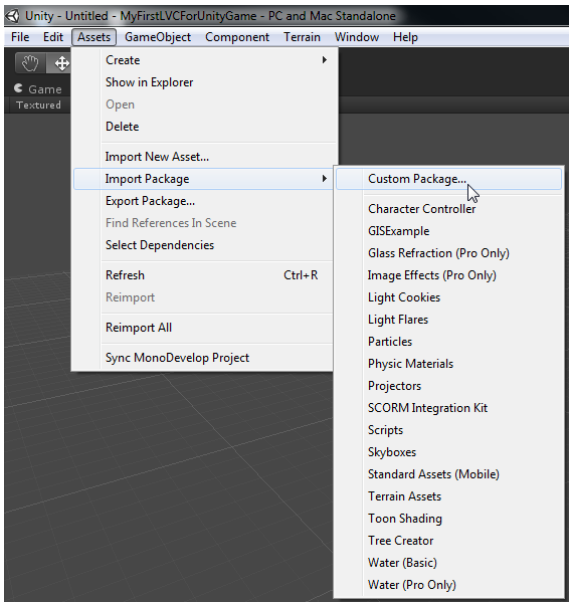


Select or create an empty folder for your project, and click `Create` (no other packages are required). Unity will restart, and open the new project.



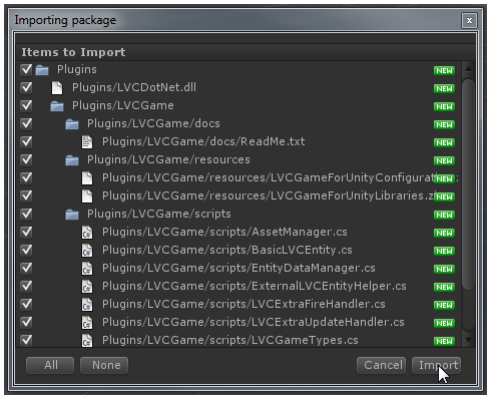
3.3 Import the Minimal LVC for Unity Package

Select `Import Package` ▶ `Custom Package` from the `Assets` menu:

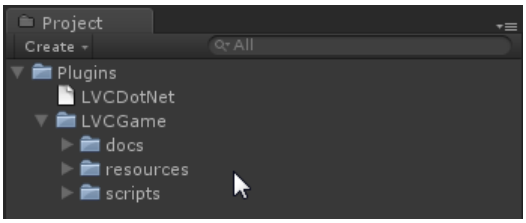


Browse to find the `minimal_lvc_for_unity` Unity Package, and click `Open`. When the package preview appears, ensure that all items are ticked, and click `Import`.

It is worthwhile noting at this point that the minimal LVC for Unity package was created using version 3.5.7 of the Unity editor. It cannot be guaranteed that importing the package into an older version of the Unity editor will work successfully. On the other hand, higher versions of the Unity editor should be able to cope, and may require some asset conversions.



In the `Project` pane, you should see a `Plugins` folder entry. Expand the entry out to see the contents:

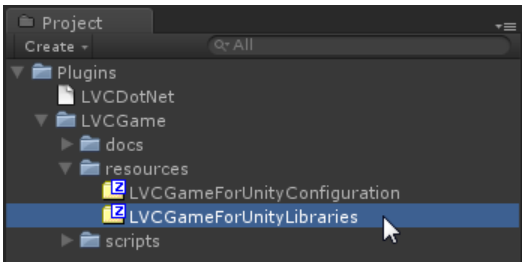


3.4 Install LVC Game for Unity Plugin Libraries

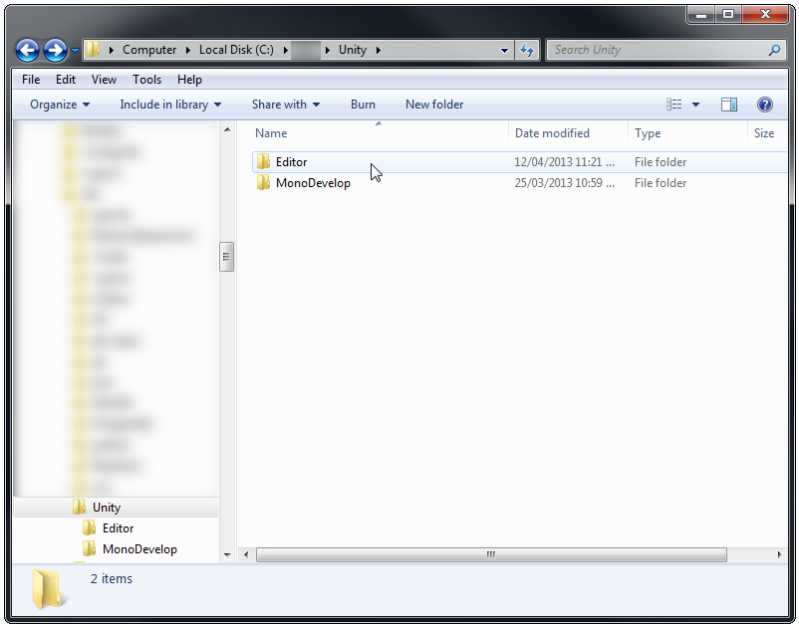
In order for the LVC Game for Unity Plugin to be used within the Unity Editor, some additional libraries are required. These must be placed in the installation directory of Unity itself.

Please note that this step only needs to be carried out once per Unity installation – if you have already carried out this step for another Unity project, you may skip this step and proceed to add the LVC configuration files to your Unity project as detailed in 3.5.

Expand the `resources` folder in the `Project` pane, and double click on the `LVCGameForUnityLibraries` zip file to open it:



Use the file explorer to navigate to the directory which contains your Unity installation:

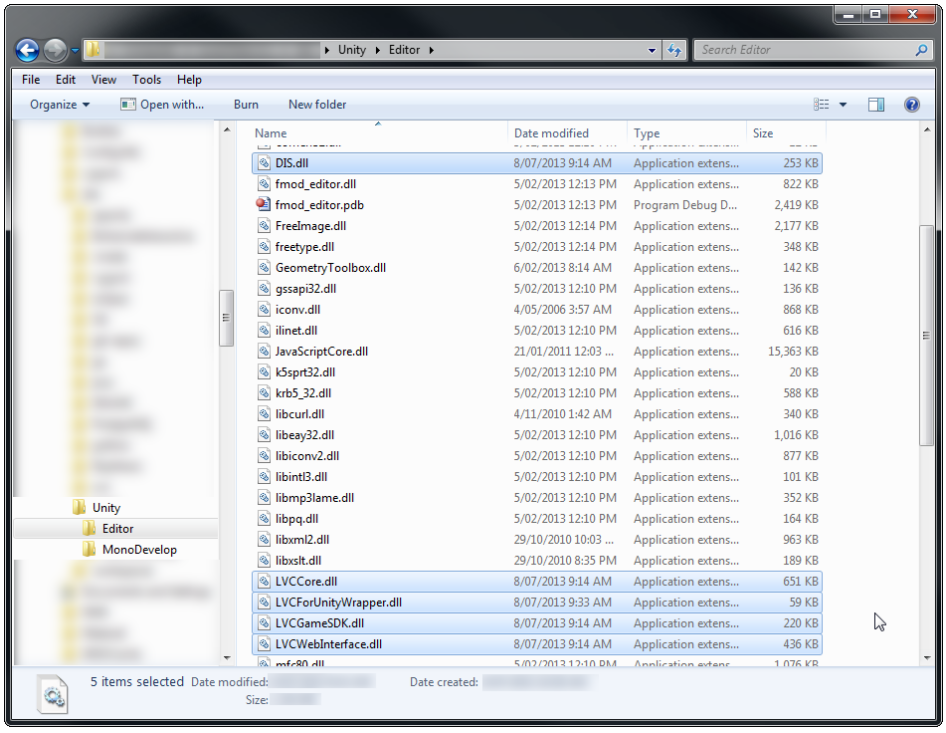


You should see two folders called `Editor` and `MonoDevelop`.

Navigate inside the `Editor` folder, and unzip the LVC Game for Unity Libraries ZIP file contents into it.

After the unzipping completes, the `Editor` folder should contain five new files:

- `DIS.dll`
- `LVCCore.dll`
- `LVCForUnityWrapper.dll`
- `LVCGameSDK.dll`
- `LVCWebInterface.dll`



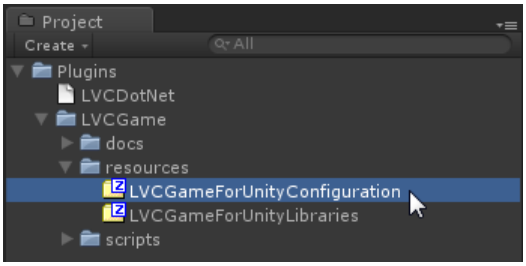
Unity needs to be restarted at this point. Exit from Unity and start again – it should automatically open with the previous project.

3.5 Add LVC Game for Unity Configuration

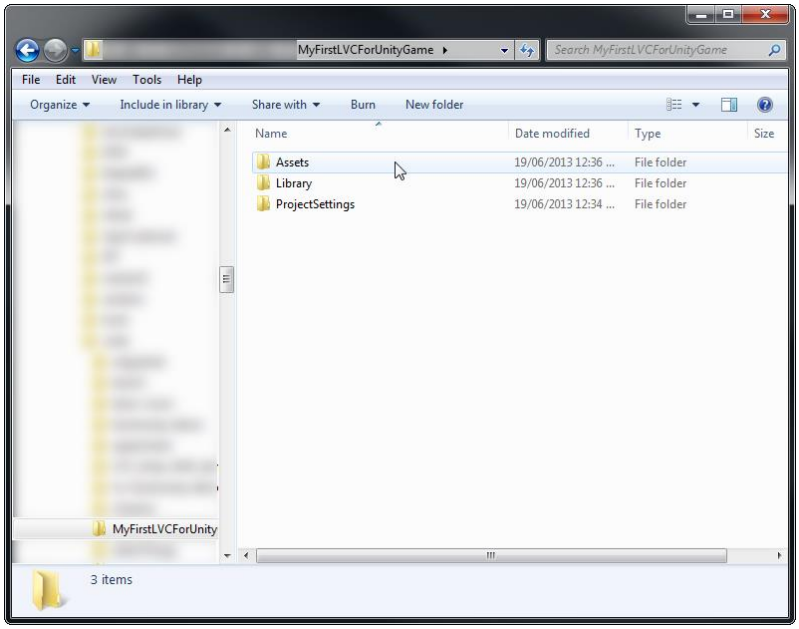
In order for the LVC Game for Unity to initialise and start, some configuration files are required. These must be placed in the `Assets` folder of the Unity project.

Please note that this step needs to be carried out *each time* you create a new LVC Game for Unity project.

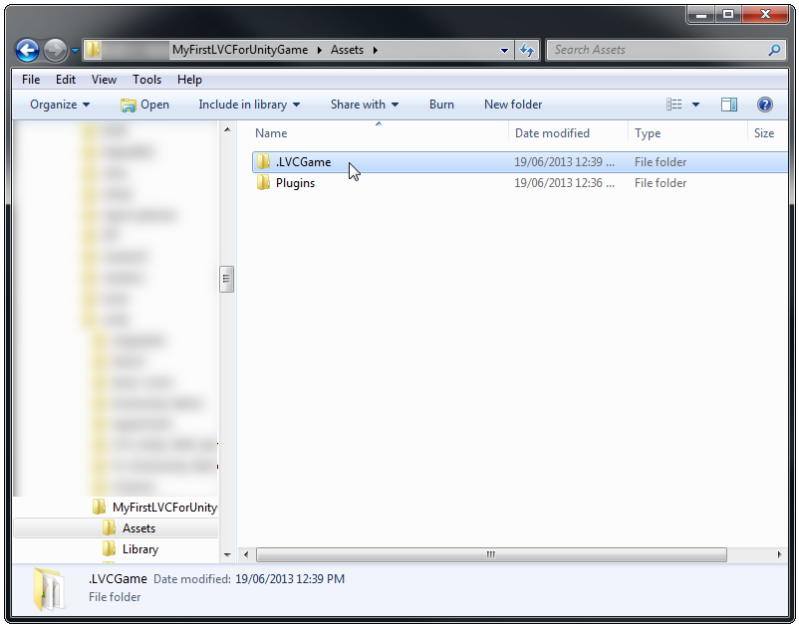
Expand the `resources` folder in the `Project` pane, and double click on the `LVCGameForUnityConfiguration` zip file to open it:



Use the file explorer to navigate to the directory which contains the Unity project.



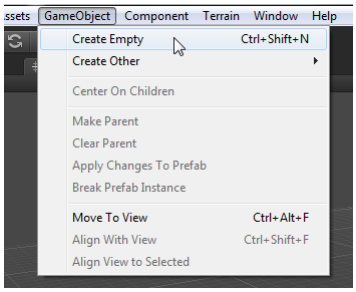
Navigate into the `Assets` folder of the project, and unzip the LVC Game for Unity Configuration ZIP file contents. After the unzipping completes, the `Assets` folder should contain a new folder called `.LVCGame`:



This folder[4] contains configuration information for the LVC Game for Unity plugin.

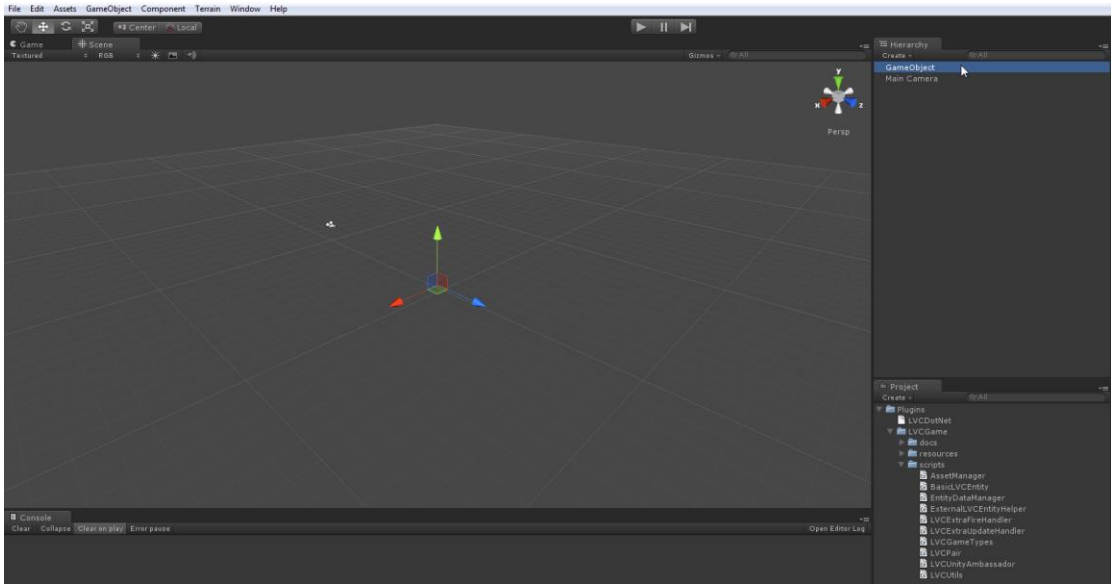
3.6 Create the LVC Helper

From the `GameObject` menu, select `Create Empty`.

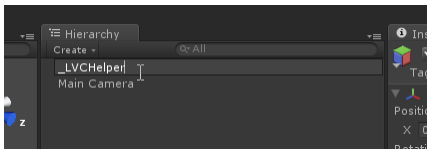


The new empty `GameObject` will appear in the `Scene` view and in the `Hierarchy` pane.

[4] Though the `.LVCGame` folder is contained within the project's `Assets` folder, its content is excluded from compilation by Unity. This is because the folder name begins with a "." character. This is necessary because the folder contains resources which Unity may mistakenly identify as requiring compilation, causing errors.

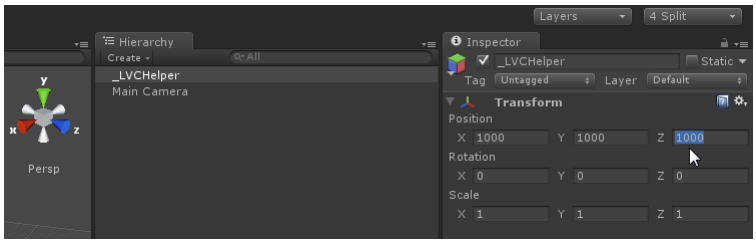


Click on the `GameObject` in the `Hierarchy` view to select it, then hit the [F2] key to rename it – type "`_LVCHelper`" (without the quotes) as the new name[5].



The LVC Helper object exists in the game to assist with LVC Game related operations, but takes no direct part in the game visually. Because of this, it is not generally helpful to leave it in its default location. It tends to get in the way of selecting and editing other game objects and clutters the `Scene` view unnecessarily.

Relocate the LVC Helper object somewhere out of the way by changing its coordinates to something extreme. With `_LVCHelper` still selected in the `Hierarchy` pane, use the `Position` value fields in the `Inspector` pane to move it to (1000, 1000, 1000)[6].

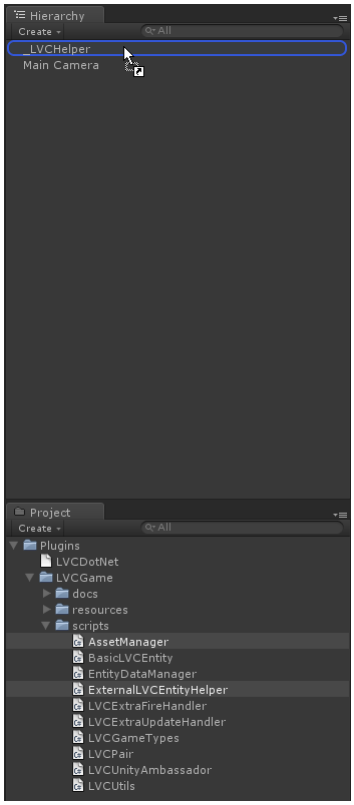


Next add the `ExternalLVCEntityHelper` and `AssetManager` behaviour scripts to the LVC Helper object from Plugins/LVCGame under the `Project` pane. As with all

[5] The name of the empty entity is not particularly important, but "`_LVCHelper`" describes its purpose quite well. The leading underscore "_" character ensures that it remains at the top of the hierarchy list, which is sorted alphabetically, making it easy to locate.

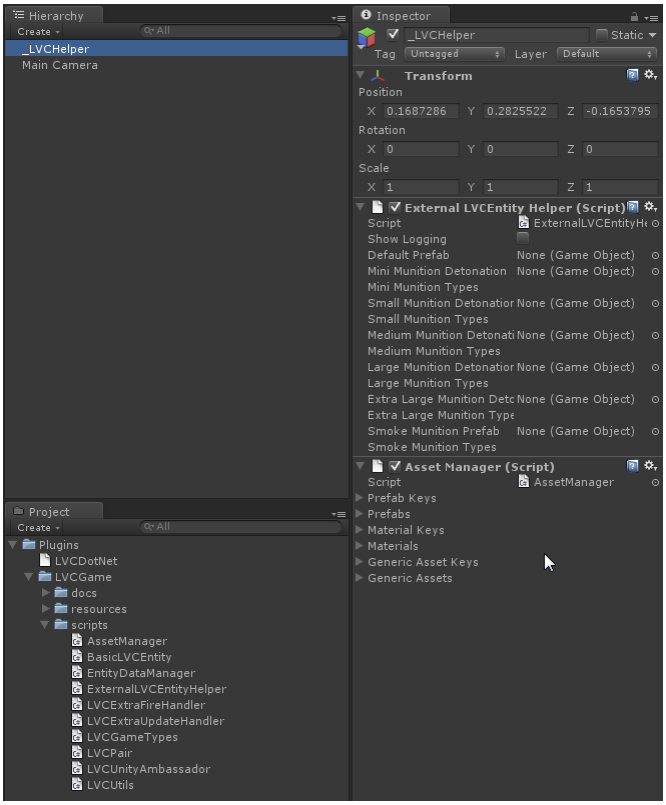
[6] The exact coordinates do not matter here – the main purpose is to shift the `_LVCHelper` `GameObject` out of the way so it doesn't interfere with scene editing.

behaviour scripts in Unity, this is done by dragging them from the `Project` pane on to the `_LVCHelper GameObject` in the `Hierarchy` pane.



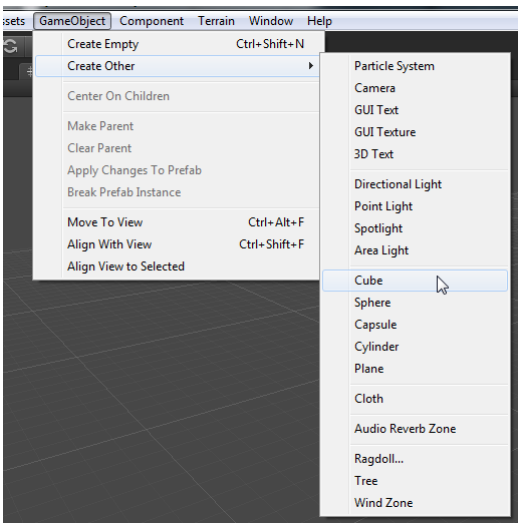
After this is done, select the `_LVCHelper GameObject` in the `Hierarchy` pane by clicking on it. The `Inspector` pane should now show entries for "`External LVCEntity Helper (Script)`" and "`Asset Manager (Script)`" to identify them as components of `_LVCHelper`[7].

[7] The ordering of the `AssetManager` and `ExternalLVCEntityHelper` behaviour scripts may be different in the Inspector pane, depending on the order that the scripts were dragged onto the `_LVCHelper`.

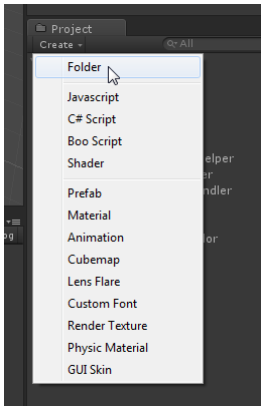


3.7 Creating a Unity Controlled LVC Game Entity

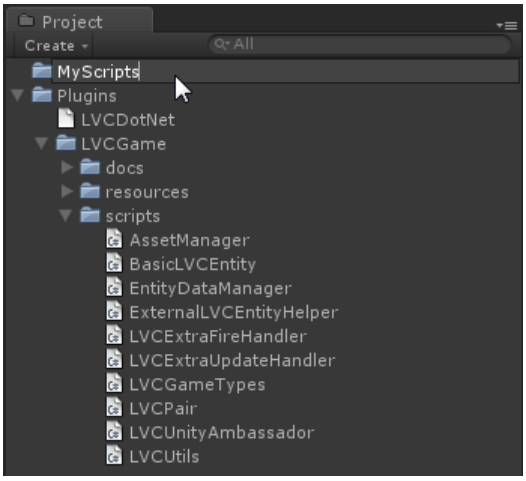
Add a cube to the scene by selecting `Create Other` ▶ `Cube` from the `GameObject` menu



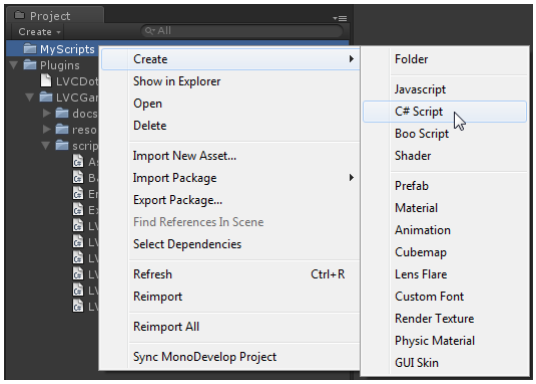
Next, create a behaviour script for the cube so that it will move when the game is played. To keep things organised though, first create a new folder in the `Project` pane. With nothing selected in the `Project` pane, select `Folder` from the `Project` pane's `Create` drop-down.



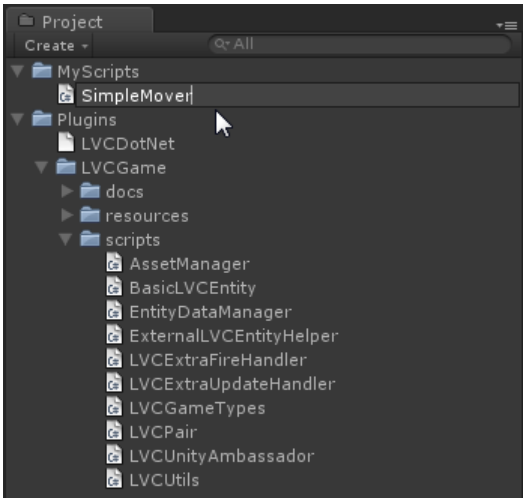
Name the new folder "`MyScripts`" (without quotes), and press [ENTER].



Right-click on the new `MyScripts` folder, and select `Create ▶ C# Script` from the context menu.



Call the new C# Script "`SimpleMover`" (without quotes).



Double click on the `SimpleMover` entry to launch Unity's MonoDevelop editor.

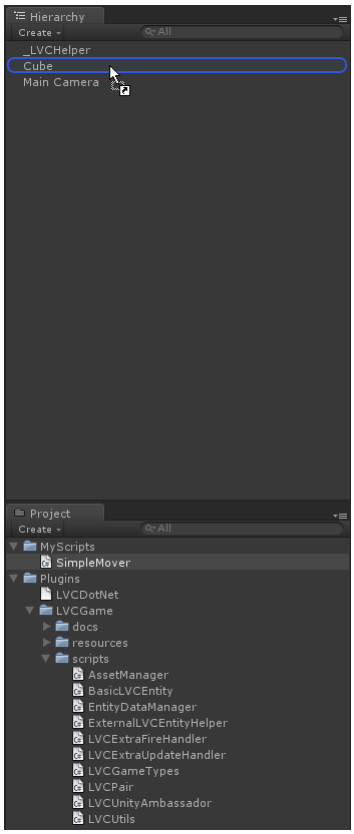
Use the following code for the `SimpleMover` class:

```
using UnityEngine;
using System.Collections;

/**
 * This is a very simple MonoBehaviour which makes the GameObject it is applied
 * to move in a circle around a central point.
 *
 * The radius of the circle and the speed which the GameObject travels around the
 * circle are able to be set by the user.
 *
 * The center of the circle is taken to be the location of the GameObject at the
 * time the game started.
 */
public class SimpleMover : MonoBehaviour {
    // the radius of the circle the object will travel around
    public float circuitRadius = 5.0f;
    // the number of seconds it takes to complete one circuit. This means that lower
    // values are faster, though 0 means no movement. -ve values are counter-clockwise.
    // by default the GameObject will take 5 seconds to complete a circuit.
    public float circuitDuration = 5.0f;
    // the start angle of the circle the object will start at
    public float startAngle = 0.0f;

    // the starting position of the GameObject, which will be the centre of the circle
    private Vector3 center;
    // track the current angle of the circuit around the circle
    private float angle;

    // a convenience constant for converting between radians and degrees
    private static readonly float DEG_TO_RAD = Mathf.PI / 180.0f;

    // Use this for initialization
    void Start ()
    {
        // record the start location of the GameObject
        center = gameObject.transform.position;
        // a circuit is from 0 to 360 degrees, starting at 0
        angle = startAngle;
    }

    // Update is called once per frame
    void Update ()
    {
        // make sure speed is a sensible value, and exit straight away if not
        if( circuitDuration == 0.0f )
                return;

        // how many degrees should the GameObject have moved around the circle?
        float step = ( 360.0f/circuitDuration ) * Time.deltaTime;
        angle += step;

        // keep angle inside 0-360 degrees
        while ( angle>360.0f )
                angle -= 360.0f;
        while ( angle<360.0f )
                angle += 360.0f;

        // update the GameObject's position appropriately
        Vector3 position = gameObject.transform.position;
        float angleInRadians = DEG_TO_RAD * angle;
        position.x = center.x + Mathf.Sin( angleInRadians ) * circuitRadius;
```

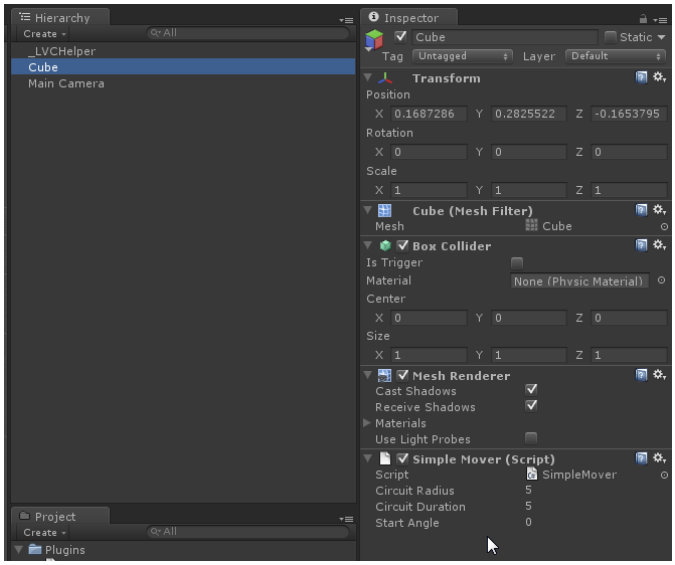
```
        position.z = center.z + Mathf.Cos( angleInRadians ) * circuitRadius;
        gameObject.transform.position = position;
        // also "roll" the game object around the circle, just to make it more interesting
        gameObject.transform.rotation = Quaternion.Euler( 0.0f, angle, -angle*5.0f );
    }
}
```

Save the `SimpleMover` code, and return to the Unity Editor. Check the `Console` pane to make sure that there are no compilation errors – if there are, check the `SimpleMover` code for errors in the MonoDevelop editor, fix them, and re-save.

Assign the `SimpleMover` behaviour script to the cube by dragging it from the `Project` pane onto the cube in the `Hierarchy` pane.

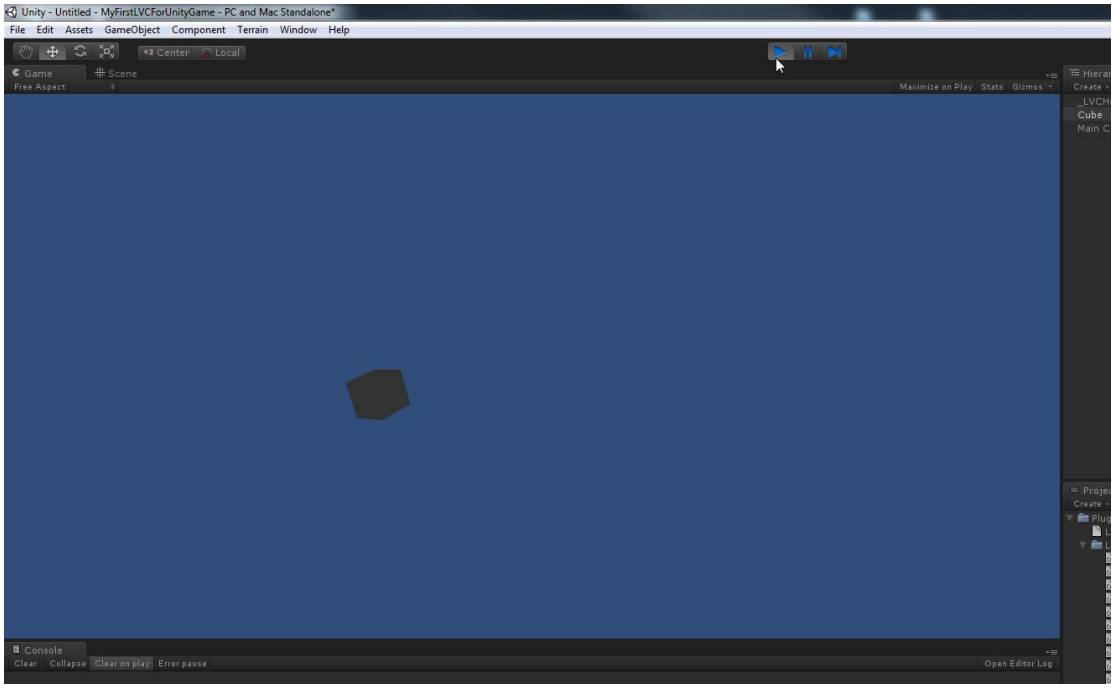


After this is done, select the cube `GameObject` in the `Hierarchy` pane by clicking on it. The `Inspector` pane should now include a "`Simple Mover (Script)`" entry to identify it as being a component of the cube.



Click the `Play` button to play the game. The cube should roll around in a circle in front of the camera.

If the cube is not visible, check that it is within the field of view of the main camera. Adjust the position of the main camera by selecting it in the `Hierarchy` pane, switching to `Scene` view, pressing [W] (i.e. to activate the object repositioning tool) and then dragging the camera using the displayed axes guides.



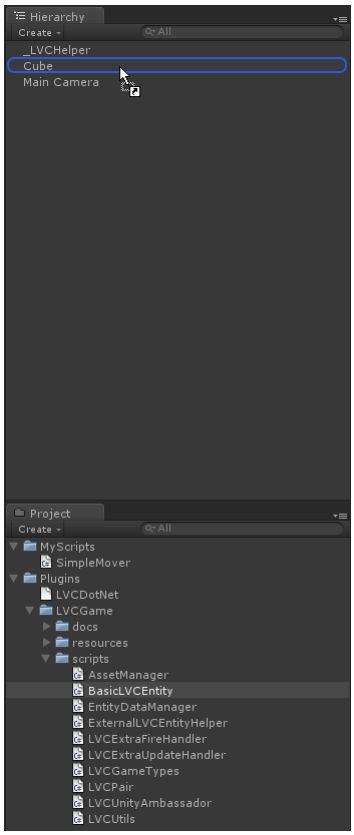
Click the `Play` button again to stop the game[8].

[8] In an actual game, the cube might be replaced with game character, and controlled through mouse or joystick input rather than being animated in this simple manner. Adding better models and more advanced controls to the game is left as an exercise for the reader.

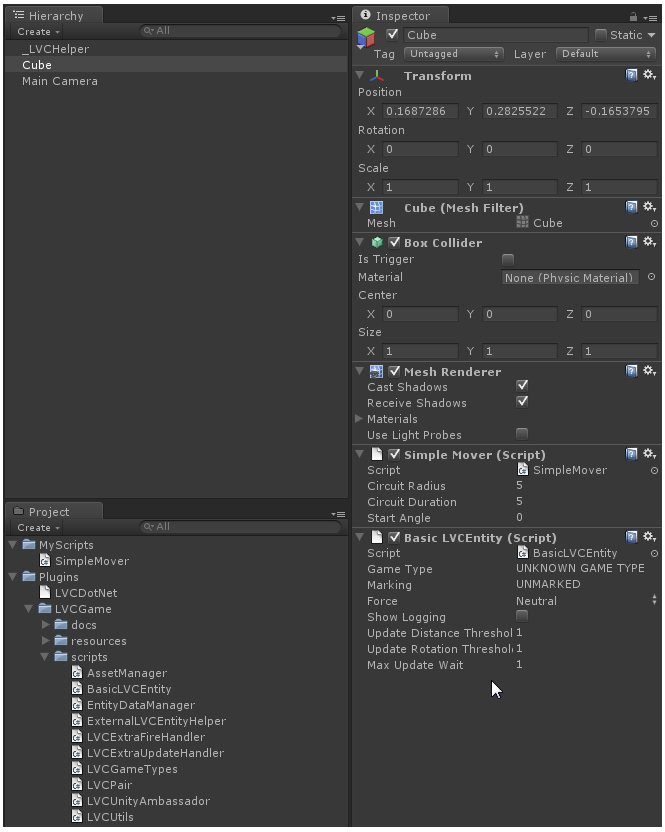
Though the cube is in the game and moving, it is not sending any updates about itself over the network. Even if other simulations are connected, they will never know the cube exists.

The `BasicLVCEntity` behaviour script can be used to rapidly turn any `GameObject` into an entity that can participate in a distributed simulation. The script automatically sends updates about the associated `GameObject` over the simulation network using the LVC for Unity plugin.

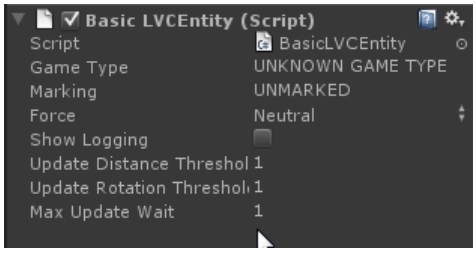
Assign the `BasicLVCEntity` behaviour script to the `Cube` by dragging it from the `Project` pane onto the cube in the `Hierarchy` pane.



After this is done, select the cube `GameObject` in the `Hierarchy` pane by clicking on it. The `Inspector` pane should now include a "`Basic LVCEntity (Script)`" entry to identify it as being a component of the cube.



Notice that there are several parameters which can be set for the `BasicLVCEntity` behaviour script:



`BasicLVCEntity`	
Parameter:	**Notes:**
Game Type	The internal type name of the entity, such as ‘truck’ or ‘tank’. Note that this must match a type in your outgoing LVC mappings configuration file (unity_to_LVC.config), otherwise data about this entity will not be transmitted to the distributed simulation.
Marking	A text marking which uniquely identifies this instance of the entity type.

Force	Whether this entity should be considered as Friendly, Opposing, Other or Neutral[9].
Show Logging	If this checkbox is ticked, logging information will be displayed in the unity console regarding LVC activities.
Update Distance Threshold	This is the minimum distance the entity must move in order for an update to be sent. For example, if the distance threshold is 1m (the default), no update will be sent until the entity moves at least 1m away from its last updated position[10].
Update Rotation Threshold	This is the minimum angle, in degrees, which the entity must rotate (around any axis) in order for an update to be sent. For example, if the distance threshold is 1° (the default), no update will be sent until the entity has rotated at least 1° from its last updated orientation.
Max Update Wait	This is the maximum time, in second, which is allowed to pass with no update sent before an update *must* be sent. An update will be sent at least this often, regardless of the distance and rotation thresholds. For example, if the time threshold is 1 second (the default) and no update has been be sent for 1 second, an update will be sent at the next possible opportunity.

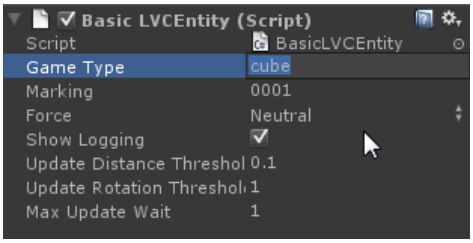
The distance, rotation and maximum wait thresholds provide a throttle on the number of updates being sent across the network for an entity instance. It would of course be possible to send an LVC update every time a frame is drawn (60 times per second, for example). However if an entity is not moving then the updates would always contain the same, redundant information. The thresholds limit the updates so that they are only sent when there are substantial changes in position or orientation, or significant passages of time.

Note that the distance, rotation and maximum wait thresholds operate in combination – if *any* of these thresholds is exceeded, an update will be sent.

[9] For flexibility it is possible to have up to 10 different forces of each of `Friendly`, `Opposing`, `Neutral` or `Hostile` forces. Which forces are actually recognised depends on the capabilities of the individual simulations, however.

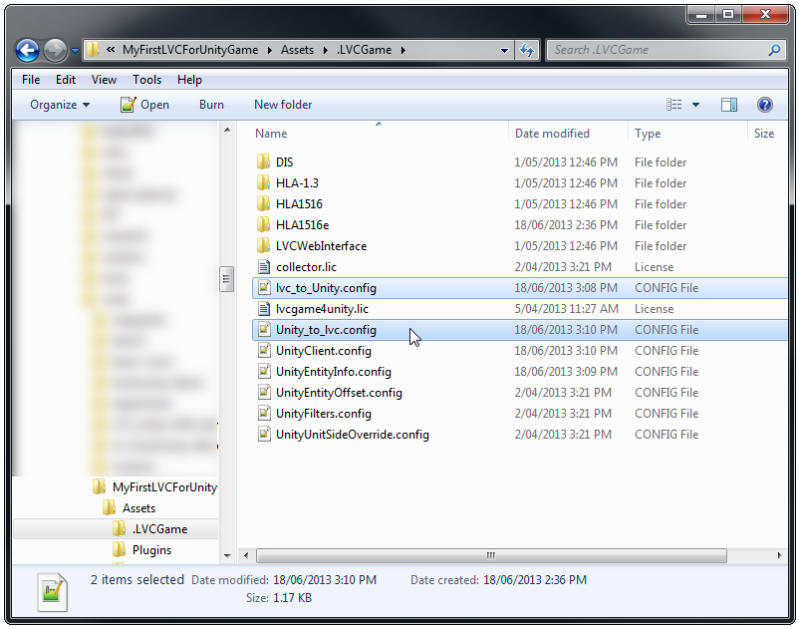
[10] Note that positional units in Unity are arbitrary; that is 1 unit in Unity could represent 1 mile, 1 centimetre, 1 inch, or any other measurement. For the purposes of standardization within a distributed simulation environment, it is most useful assign a scale of 1 unit = 1 meter.

Enter 'cube' for the `Game Type`, '0001' for the `Marking`, check the `Show Logging` checkbox, and set the `Update Distance Threshold` to 0.1 (i.e., 10cm). The other settings may be left with their default values.

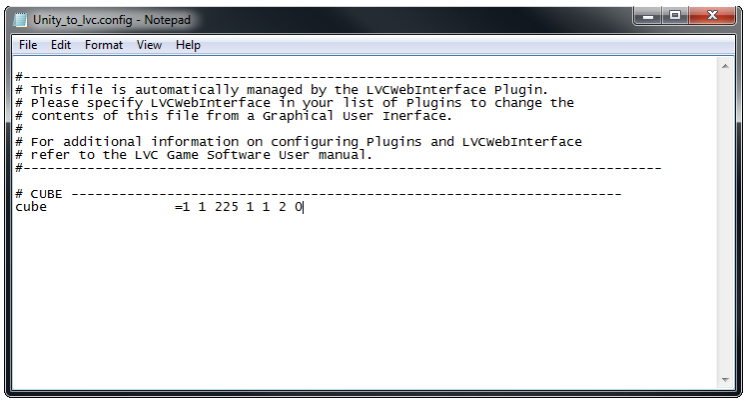


At this point, we also need to ensure that the entity mapping configuration used by LVC Game for Unity contains an entry for the `Game Type` 'cube'.

Using the file explorer, navigate into the `.LVCGame` folder under the `Assets` folder of the project.



Open the `Unity_to_lvc.config` file in a text editor. This configuration file determines how the internal entity type names used by Unity are converted to the standardised DIS/HLA compatible entity enumerations.



The format of the file is quite simple:

- Lines beginning with a '#' symbol are treated as comments for humans to read, and are ignored by LVC Game.
- All other lines are in the following format:
  `UNITY GAME TYPE=STANDARD ENUMERATION`

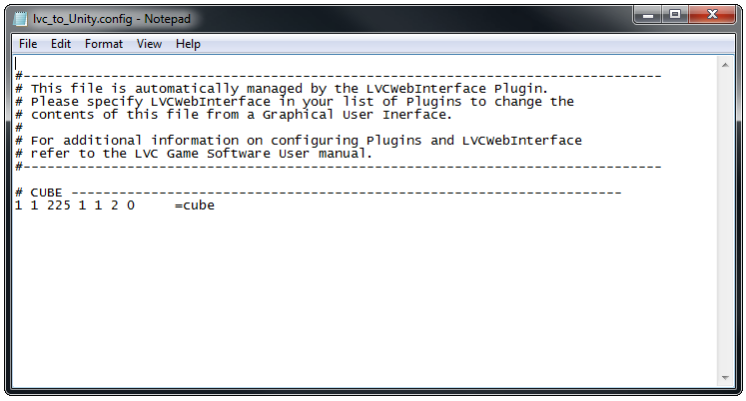
You should find that the line for the cube type has already been completed for you:

```
cube = 1 1 225 1 1 2 0
```

This means that whenever an update for the type `cube` is sent over the network, it will send the information using the standard enumeration `1 1 225 1 1 2 0`.

This enumeration actually corresponds to a US M1A1 Abrams tank, so other simulations connected to this one should "see" our cube as if it were a tank.

Close the file, and then open the `lvc_to_Unity.config` file in a text editor. This configuration file is the opposite of the previous one – it determines how the standardised DIS/HLA compatible entity enumerations are converted to the internal entity type names used by Unity.



The format of the file is, again, very simple:

- Lines beginning with a '#' symbol are treated as comments for humans to read, and are ignored by LVC Game.
- All other lines are in the following format:
  `STANDARD ENUMERATION=UNITY GAME TYPE`

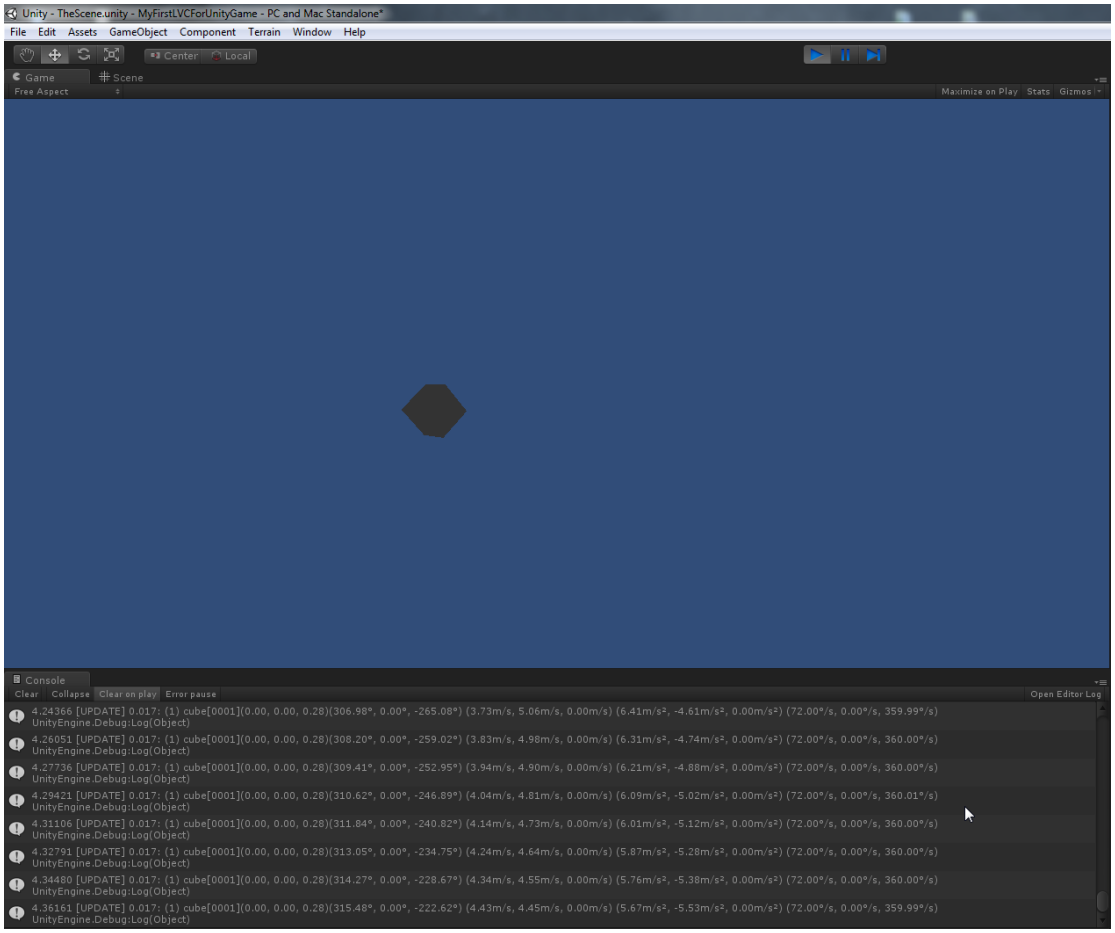
As before, you should find that the line for the cube type has already been completed for you in this configuration file:

```
1 1 225 1 1 2 0 = cube
```

This means that whenever an update is received from the network using the standard enumeration `1 1 225 1 1 2 0`, it will recognise that the information is for an entity of type `cube`.

This enumeration actually corresponds to a US M1A1 Abrams tank – this means that if other simulations are connected to this one, we intend to represent these tanks as cubes in our game.

Click the `Play` button to play the game again. As before, the cube should roll around in a circle in front of the camera. This time, however, there should be logging in the Unity console area displaying the updates being sent out across the network by the LVC Game for Unity plugin.



The logging output lines will, for the most part, look something like this:

```
6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)
(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s)
(-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)
```

The following information is contained in each log line:

- The elapsed time since game start (in seconds)
  6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)
- The outgoing LVC message type (one of [CREATE], [UPDATE] or [DELETE])
  6.65164 **[UPDATE]** 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)
- The elapsed time since the last update was sent (in seconds)
  6.65164 [UPDATE] **0.017**: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)
- The unique identifier for the entity instance (automatically generated by the LVC Game for Unity plugin)
  6.65164 [UPDATE] 0.017: (**1**) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)
- The (internal) Entity Type
  6.65164 [UPDATE] 0.017: (1) **cube**[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)
- The Entity Marking
  6.65164 [UPDATE] 0.017: (1) cube[**0001**](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°) (-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²) (72.00°/s, 0.00°/s, 359.99°/s)

- The Entity's position (along the X,Y and Z axes)

```
6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°)
(-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²)
(72.00°/s, 0.00°/s, 359.99°/s)
```

- The Entity's orientation (along the X,Y and Z axes)

```
6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°)
(-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²)
(72.00°/s, 0.00°/s, 359.99°/s)
```

- The Entity's velocity (in meters per second along the X,Y and Z axes)

```
6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°)
(-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²)
(72.00°/s, 0.00°/s, 359.99°/s)
```

- The Entity's acceleration (in meters per second squared along the X,Y and Z axes)

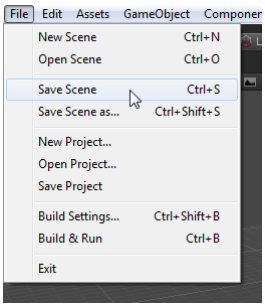
```
6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°)
(-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²)
(72.00°/s, 0.00°/s, 359.99°/s)
```

- The Entity's angular velocity, or rotation speed, (in degrees per second along the X,Y and Z axes)

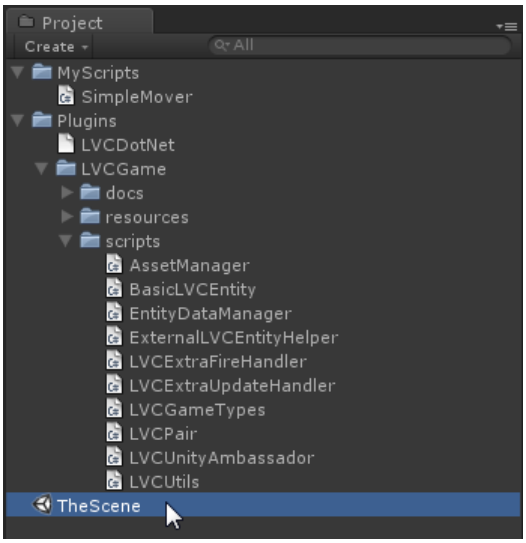
```
6.65164 [UPDATE] 0.017: (1) cube[0001](0.00, 0.00, 0.28)(120.36°, 0.00°, -118.21°)
(-3.12m/s, -5.46m/s, 0.00m/s) (-6.90m/s², 3.83m/s², 0.00m/s²)
(72.00°/s, 0.00°/s, 359.99°/s)
```

We now have a fairly complete (if simple) simulation of a moving game entity.

Save the current scene. From the `File` menu, select `Save Scene`.



Choose a name for the scene (such as "`TheScene`"), and click `Save`. The new scene will appear in the `Project` pane:



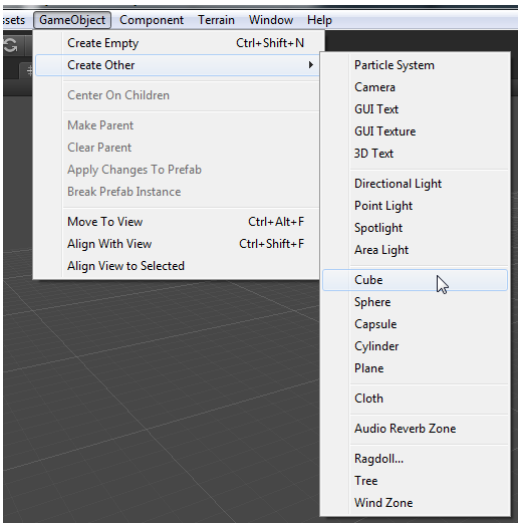
3.8 Representing Externally Simulated LVC Game Entities in Unity

At the moment we are sending updates about our cube entity out across the network, but we not doing anything with *incoming* simulation information.

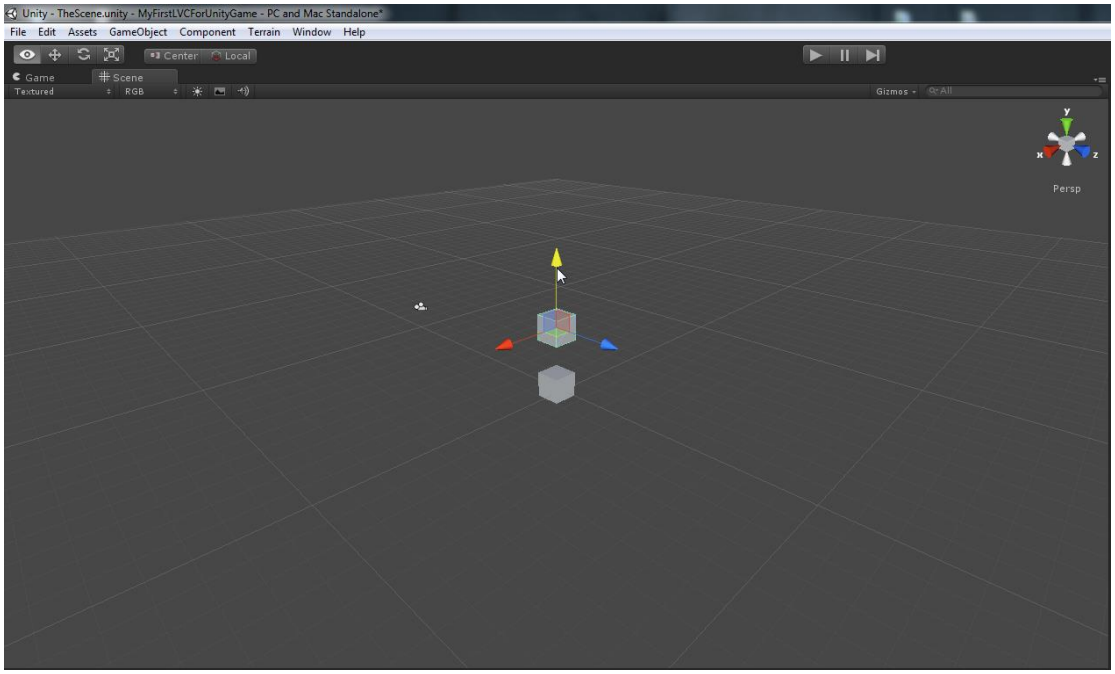
The `AssetManager` and `ExternalLVCEntityHelper` behaviour scripts which we assigned to the `_LVCHelper` `GameObject` at the beginning of this tutorial do most of the work for us, however.

First, let's create a Unity "prefab" to represent 'external' cubes – that is, cubes which are simulated by other games on the network. A prefab is essentially a "cookie cutter" or "stamp" object which we can make copies of to represent externally simulated entities.

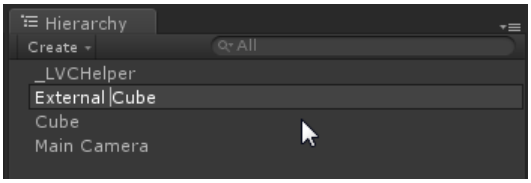
Add a new cube to the scene by selecting `Create Other` ▶ `Cube` from the `GameObject` menu:



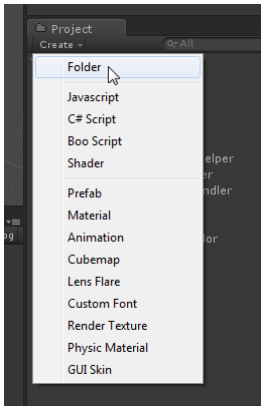
The newly created cube will appear "on top of" the existing cube – click and drag on the green Y-axis handle (it will turn yellow once you start dragging) to drag it up so that it can be seen as a separate object (the exact position does not matter):



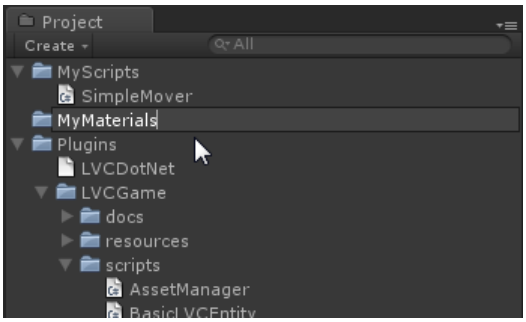
The selected cube will be highlighted in the `Hierarchy` pane – click on it, and press [F2] to rename it as "External Cube"



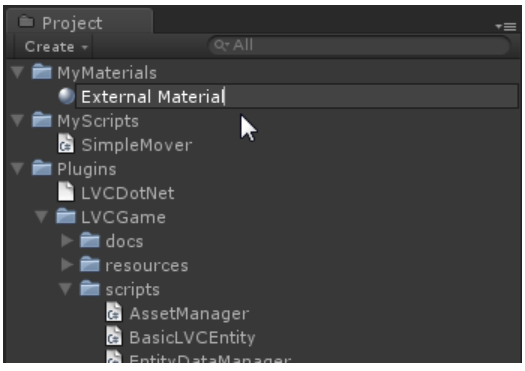
Next, let's change the colour of the external cube so that we can distinguish it from the other cube. First create a new folder in the `Project` Pane called `Materials`. With nothing selected in the `Project` pane, select `Folder` from the `Project` pane's `Create` drop-down.



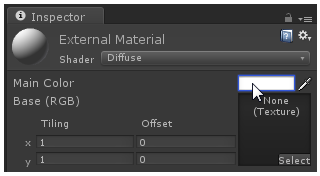
Call the new folder "`MyMaterials`" (without quotes), and press [ENTER].



Right-click on the new `MyMaterials` folder, and select `Create` ▶ `Material` from the context menu. Name the new material "`External Material`".

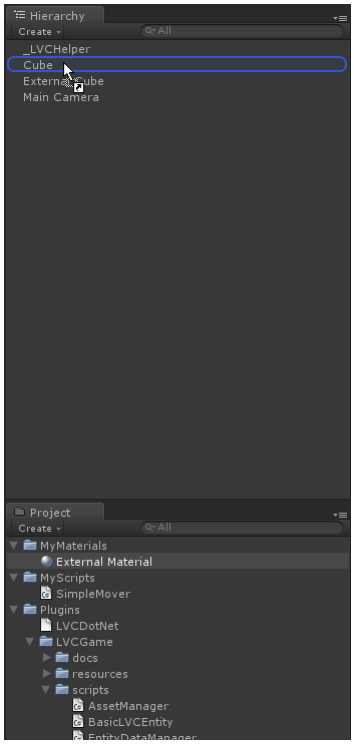


Click on the new material to select it, and then click on the white colour swatch in the `Inspector` pane labelled `Main Color`:

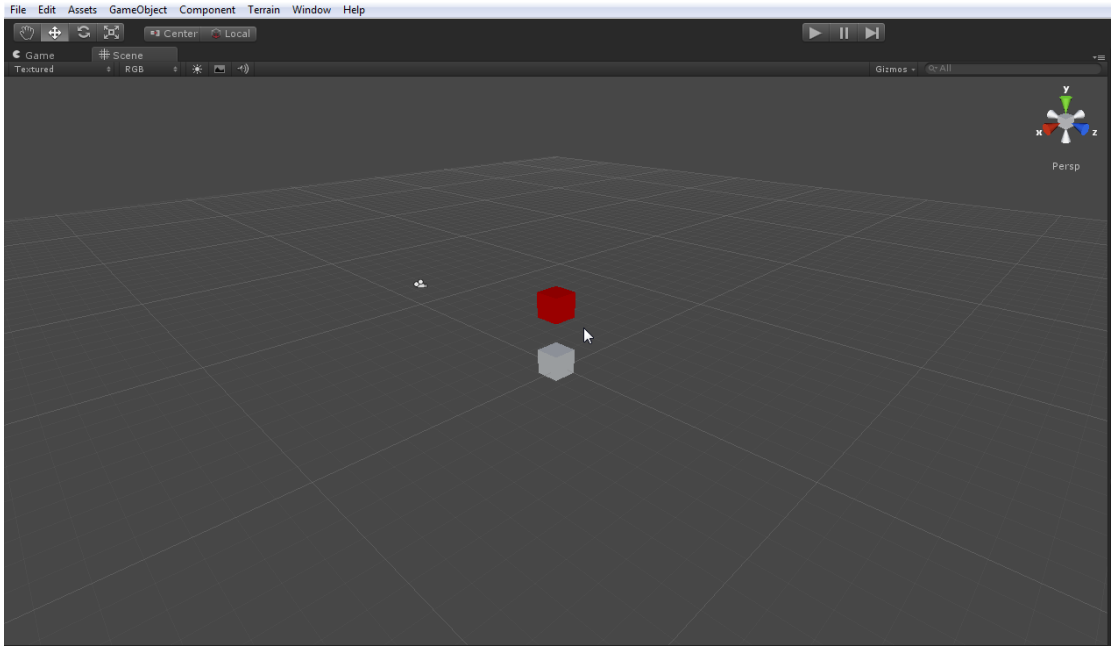


A colour selection tool will appear – use it to select a colour which will stand out, such as red:

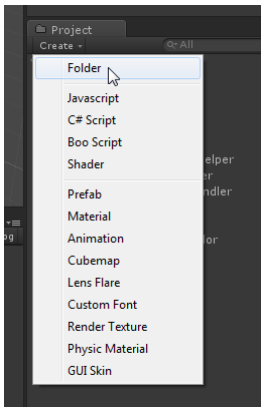
Next, assign the material to the external cube by dragging the material from the `Project` pane onto the external cube in the `Hierarchy` pane.



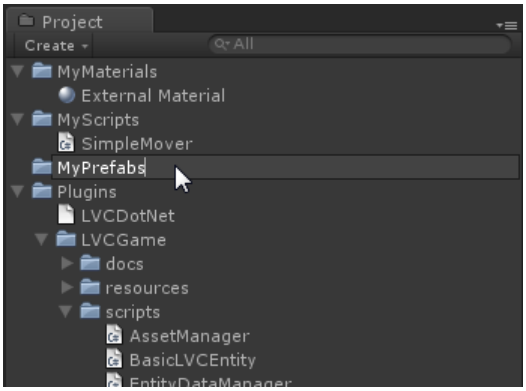
The `External Cube` should turn the same colour in the `Scene` view:



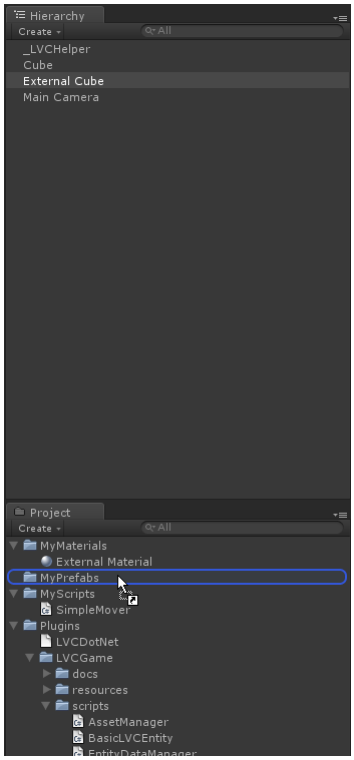
Finally, we need to turn the `External Cube` into a Unity prefab. To keep things organised, create a new folder in the `Project` Pane called `MyPrefabs`. Once again, with nothing selected in the `Project` pane, select `Folder` from the `Project` pane's `Create` drop-down.



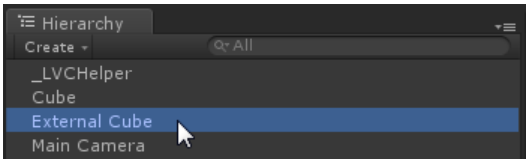
Call the new folder "`MyPrefabs`" (without quotes), and press [ENTER].



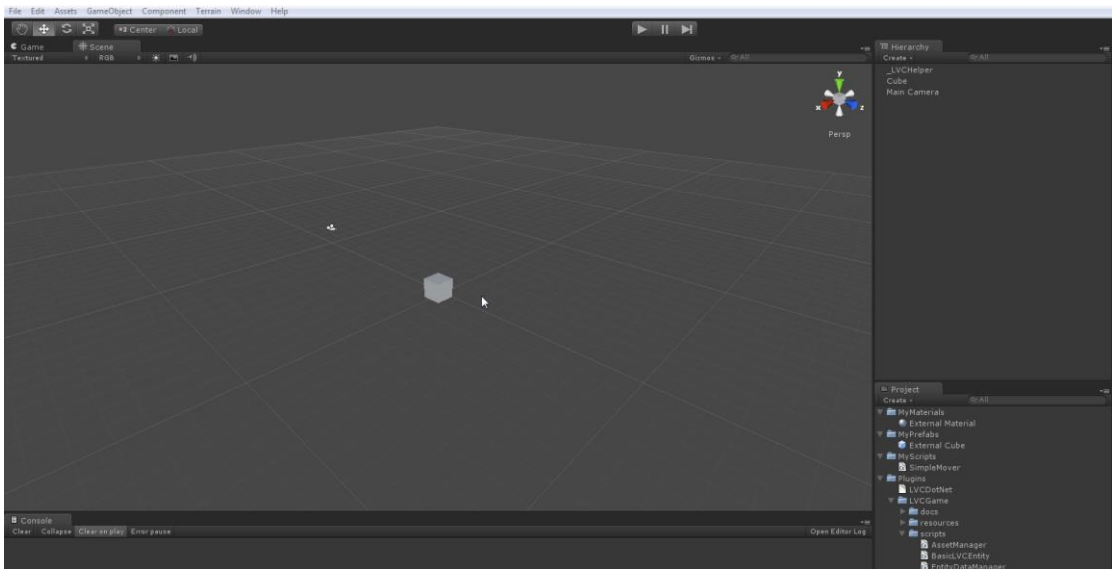
Now drag the external cube from the `Hierarchy` pane, and drop it onto the `MyPrefabs` folder in the `Project` pane:



New that the external cube is available as a prefab, it is no longer needed in the Scene. Select the external cube in the `Hierarchy` pane by clicking on it, and press the [DELETE] key.

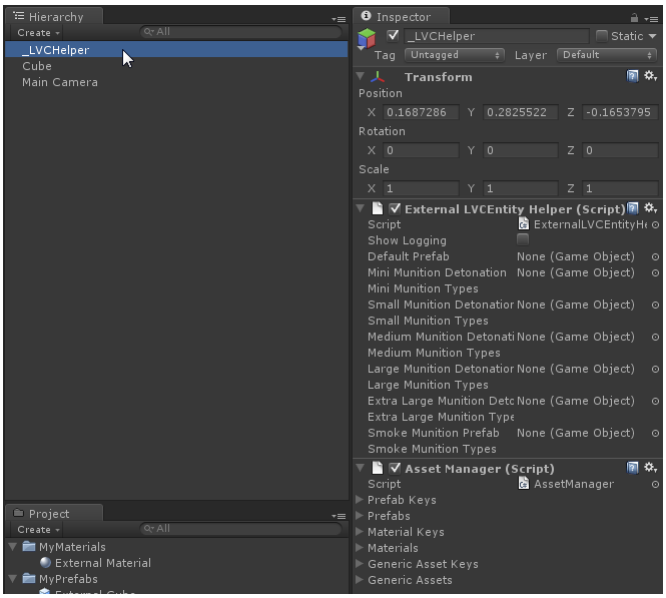


The external cube will disappear from the `Hierarchy` pane, and also from the `Scene` view, leaving only the original cube.

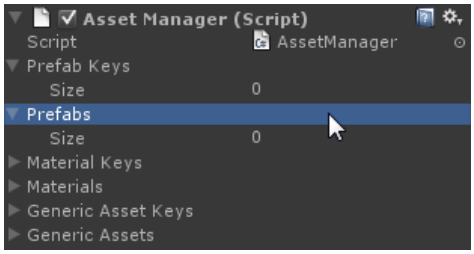


Finally, we need to assign the external cube prefab so that it is used to represent externally simulated cube entities. In other words, if an external simulation has a cube, we should be able to see it within Unity as one of our red external cubes[11].

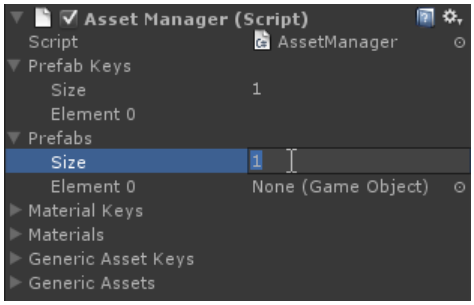
Click on the `_LVCHelper` `GameObject` in the `Hierarchy` pane to select it. Once it is selected, the `Inspector` pane should show the `ExternalLVCEntityHelper` script, and the `AssetManager` script:



Under the `AssetManager` script, unfold both the `Prefab Keys` and `Prefabs` entries by clicking on them:

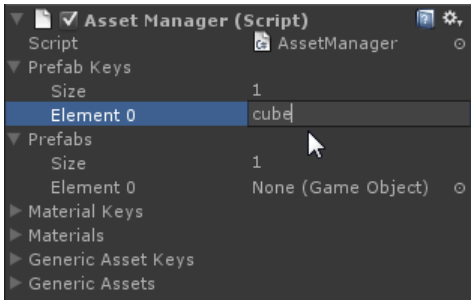


Modify the Size entry of both to 1 so that we can register our single prefab:



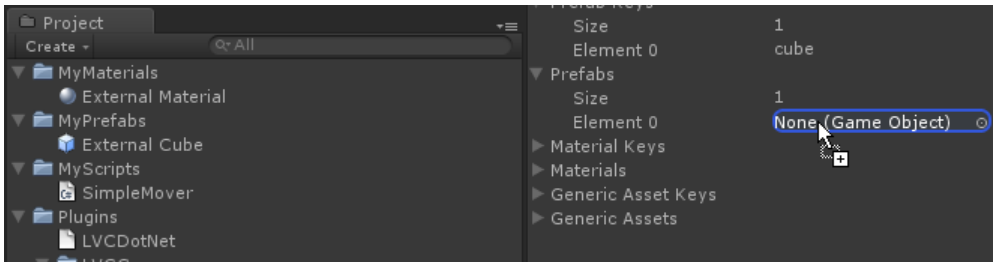
Enter the entity type name "cube" for `Element 0` of the `AssetManager` `Prefab Keys`:

[11] In an actual game, the external entity cube might be replaced with a more realistic model, and more external entity representations could be added. Such modifications are left as an exercise for the reader.

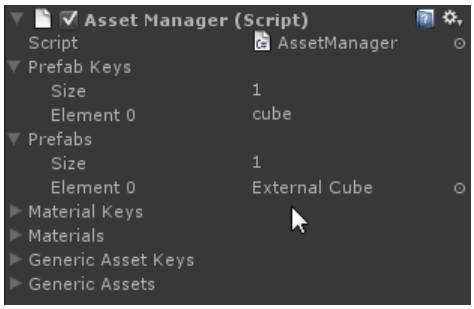


The "cube" entity name in this case corresponds to the "cube" entry in the `Unity_to_lvc.config` file.

Drag the `External Cube` prefab from the `Prefabs` folder of the `Project` pane onto `Element 0` of the `AssetManager`'s `Prefabs` list:



The `AssetManager` properties in the `Inspector` pane should now look something like this:



The `AssetManager` matches the elements of the `Prefab Keys` list with the corresponding elements of the `Prefabs` list , and uses them as a lookup table. By setting `Element 0` of `Prefab Keys` to "cube", and `Element 0` of `Prefabs` to `External Cube`, we are essentially saying that "if there is an external entity with type name 'cube', represent it using the '`External Cube`' prefab".

That's all that needs to be done to represent externally simulated cubes – the `ExternalLVCEntityHelper` script and the `AssetManager` will do the remainder of the heavy lifting for us (at least for this simple example).

If we now run two (or more) appropriately configured instances of Unity with this project on two separate machines, each Unity game should see the cubes in the other connected games[12].

The following diagram is a high level overview of the completed example:

[12] Note that the configured site and application IDs for *each* of the communicating instances of the Unity editor (and/or external simulations) must be unique in order to work together.

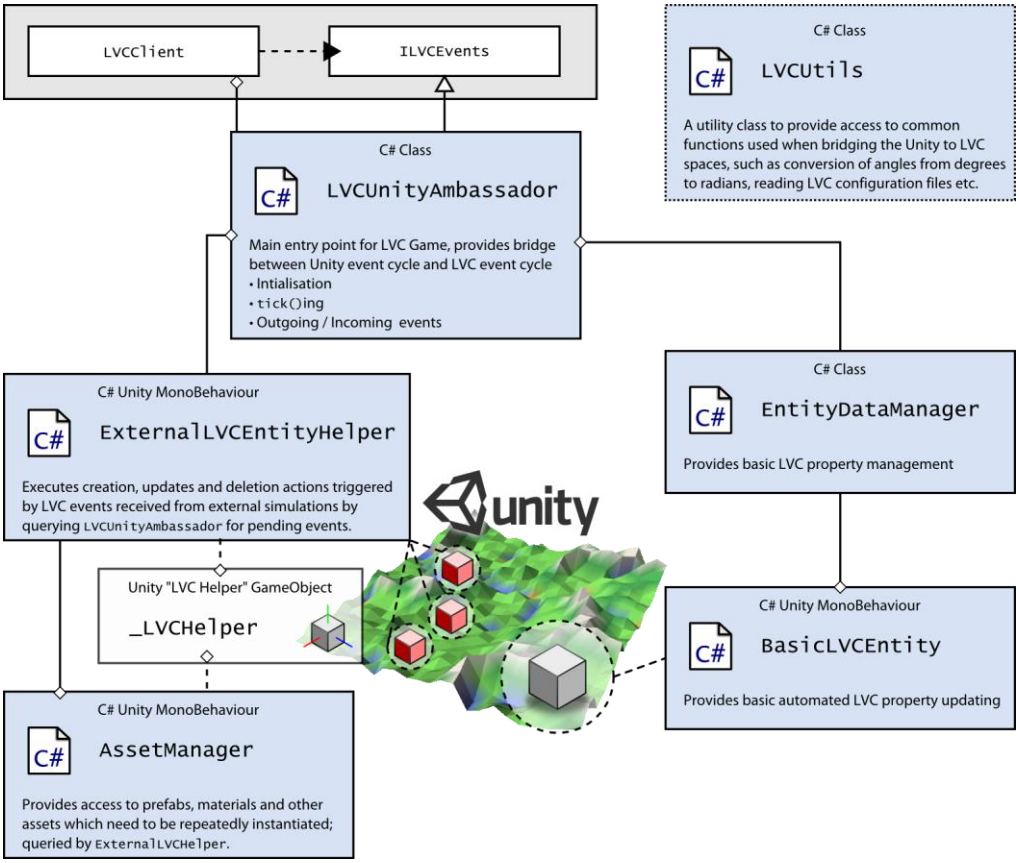


Figure 5 High Level Overview of Completed Example

4 Conclusion

This has been a simple introduction to the fundamentals of creating a game object in Unity. The example sends updates about the state of a locally created and managed game entity, and can provide a representation of one type of externally simulated entity within Unity.

The scripts provided provide basic handling suitable for this example, but will most likely need to be customised, expanded, or even replaced, in order to accommodate the requirements of more complex distributed simulations.