LVC for Unity
Developer's 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
20 Jun, 2013	0.1	Initial Draft	Andrew Laws
1 Jul, 2013	0.2	Draft Revision	Michael Huynh
26 August, 2013	0.3	Minor Revisions	Andrew Laws

Table of Contents

Table of Figures

1 Introduction

LVC Game for unity provides an abstracted C# layer that can be used to integrate directly with your Unity game to obtain network interoperability over several communication protocols.

Abstracted outgoing and incoming events are sent through the LVC Game interfaces to interact with third party applications.

For example, entity creation, position and state updates are handled bi-directionally by LVC Game to provide a complete integration solution to your application. Specific protocols such as DIS and HLA are implemented behind a facade as plugins and as such your application remains code and protocol-agnostic.

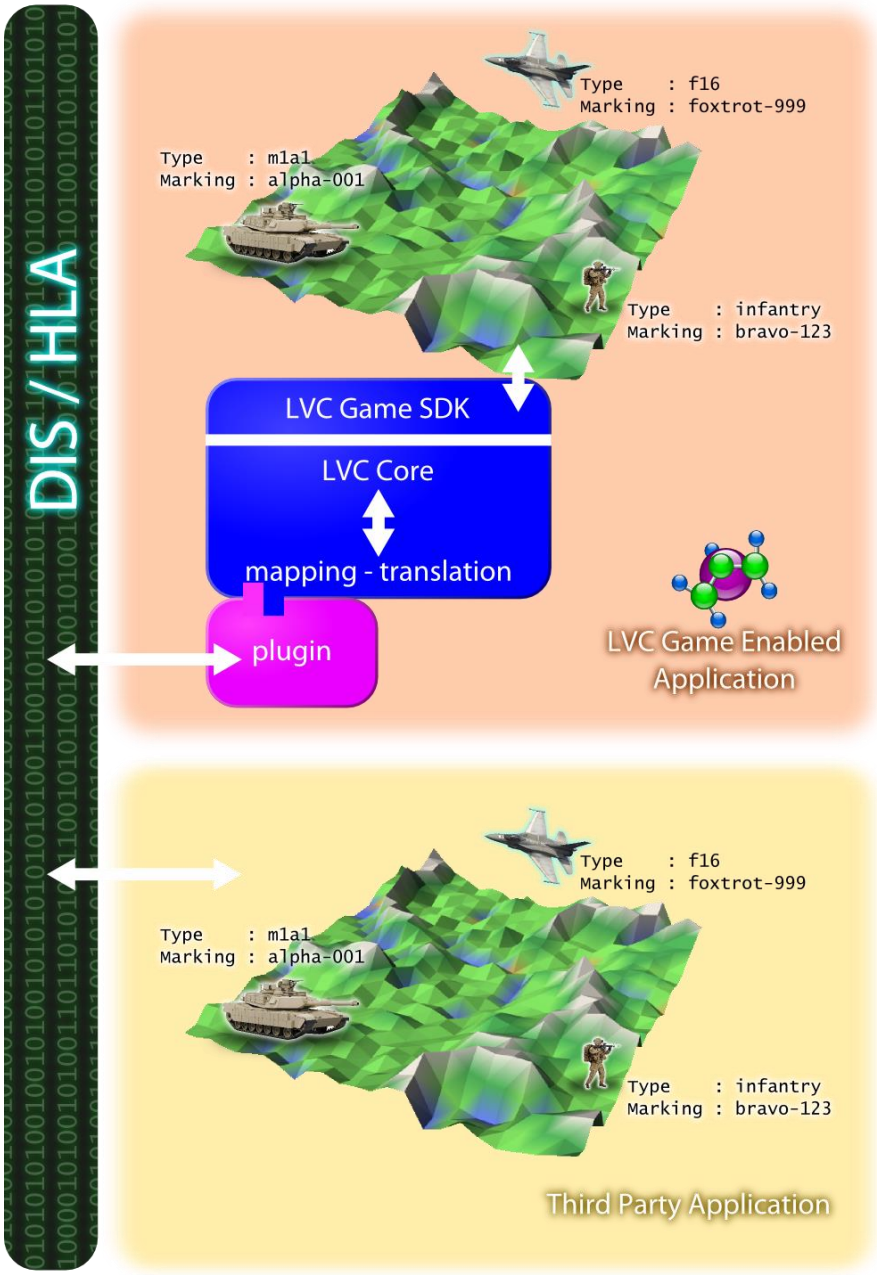


Figure 1 LVC Game Overview

1.1 Audience and Purpose

This guide is intended for software developers and illustrates how LVC enabled Unity games can be built with LVC Game for Unity.

2 LVC Game for Unity Package Overview

2.1 Package Contents

The LVC Game for Unity package contains the following folders:

`/Plugins`

`/LVCGame`

`/docs`	User Manual, Developer's Guide, API documentation.
`/resources`	A ZIP file containing system libraries required for Unity and the Unity Editor to work with LVC Game for Unity. A ZIP file containing configuration files required for LVC Game for Unity to initialise and start.
`/scripts`	C# scripts providing some examples of how to approach the creation of an LVC enabled Unity game.

2.2 High Level Interface layout

LVC Game for Unity contains the following high level interfaces:

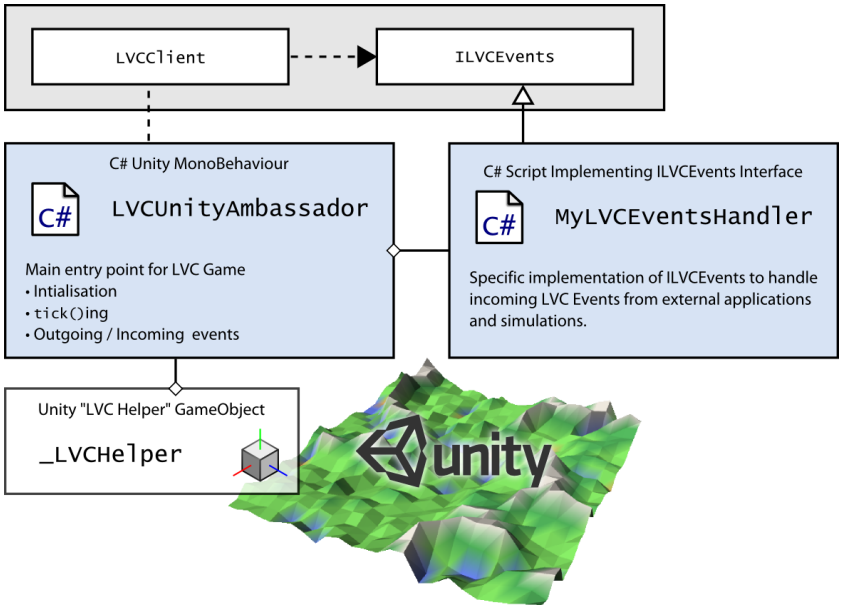


Figure 2 High Level Overview

Typically a helper Unity `GameObject` is created to "hook into" the Unity game event cycle – it is generally an "invisible" object, and takes no active part in gameplay itself. In Figure 2, the helper object is named "`_LVCHelper`".

The helper object contains a behaviour script component which, on game start, creates and initialises an `LVCClient` instance, giving the Unity game access to LVC Game. In Figure 2, this behaviour script component is named "`LVCUnityAmbassador`". The primary role of this behaviour script component is to provide a means by which the Unity event cycle and the LVC event cycle can be reconciled – this is important, as the two are quite separate.

The initialised `LVCClient` instance is then associated with an implementation of the `ILVCEvents` interface to handle incoming/outgoing LVC events ("`MyLVCEventsHandler`", in Figure 2).

In terms of sequencing, the `_LVCHelper GameObject`'s `LVCAmbassador` behaviour script component typically needs to interact with LVC Game in three phases:

1. **Creation and initialisation phase**: An instance of `LVCClient` is created, initialised and started, all configured plugins are automatically loaded and `LVCClient` is ready to begin sending and receiving events. During this phase, your application also registers an `ILVCEvents` listener to receive incoming events generated by other, external simulation applications and games during the execution phase.
2. **Execution phase**: In this phase, `LVCClient.tick()` is called at regular intervals. This causes all events to be processed (for example dead reckoning and queued incoming events). Incoming events, such as external entity creations, and updates and deletions of externally created entities, must be handled during this phase. Changes to entities created "locally" by the Unity must be sent as outgoing events so that external simulations and games are notified during this phase.
3. **Shutdown phase**: Once execution is complete, all locally created game entities are deleted and the `LVCClient` instance itself is destroyed.

These three phases are shown in Figure 3.

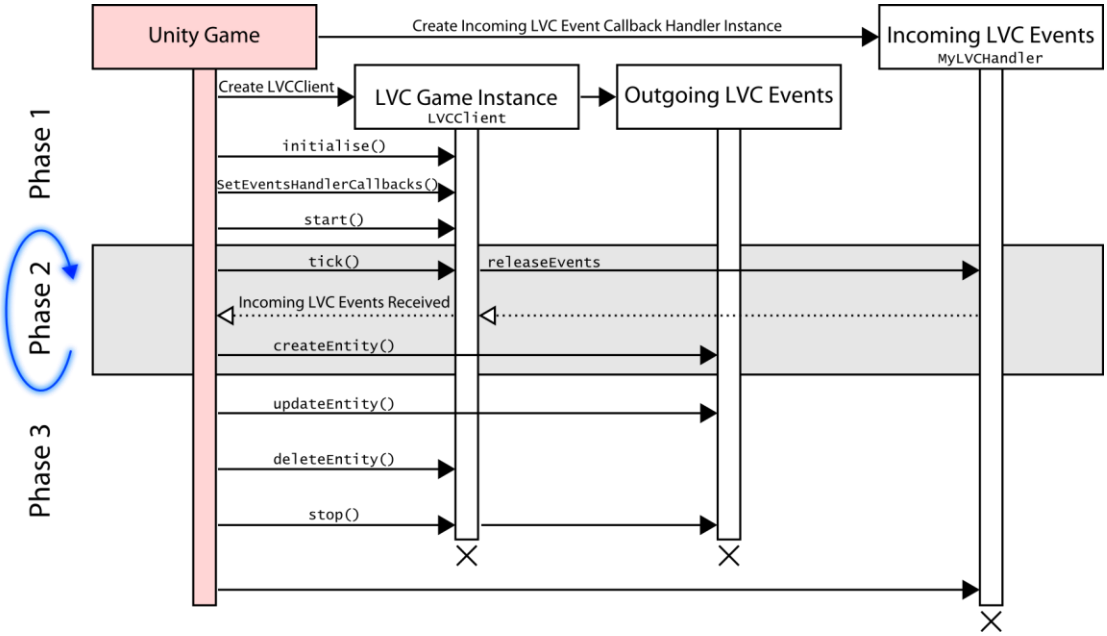


Figure 3 High Level Sequence of Execution

2.3 LVC Game Reference Frames

This section defines the coordinate systems and associated reference frames used by LVC Game.

2.3.1 World Reference Frame

World Position in LVC Game is expressed as *Latitude*, *Longitude* and *Altitude* (LLA) in reference to the World Geodetic System 1984 (WGS84):

- *Latitude* is the angle between the equator plane and the rotation axis of the earth.
- *Longitude* is the angle between the Greenwich meridian and the projection of the point in question into the equator plane.
- *Altitude* is the positive distance above the WGS84 ellipsoid.

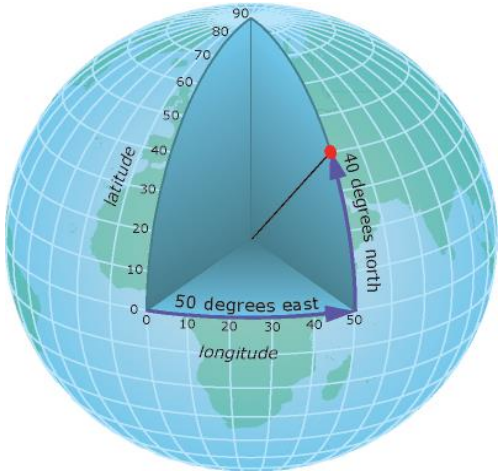


Figure 4 LLA Reference Frame

2.3.2 Local Tangent Plane Reference Frame

Whilst LLA coordinates are well suited for world position, quantities such as world velocity and acceleration are better expressed in local *East*, *North*, *Up* (ENU) Cartesian coordinates or Local Tangent Reference frame, formed from a plane tangent to the Earth's surface fixed to a specific location.

By convention the *East* axis is labelled *X*, the *North Y* and the *Up Z*.

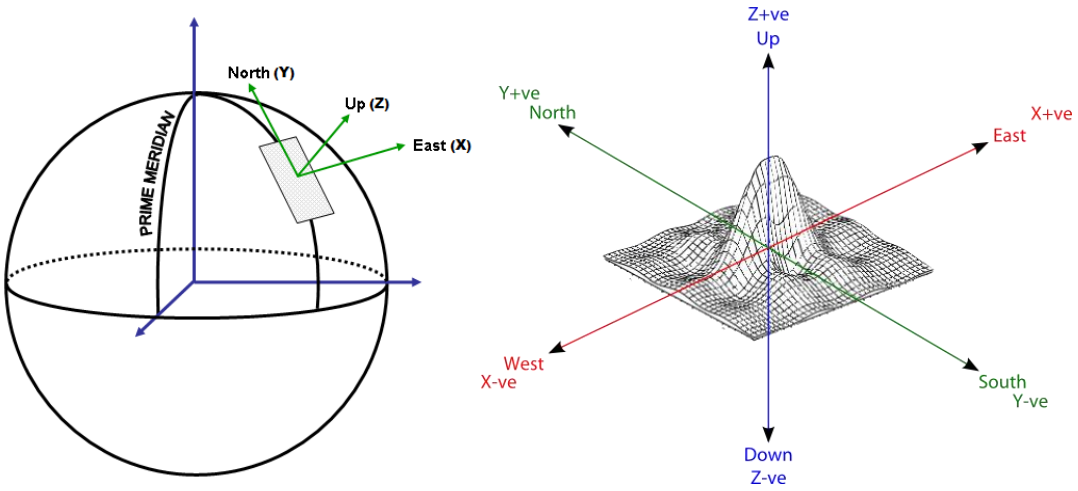


Figure 5 Local Tangent Reference Frame

2.3.3 Body Reference Frame

The Body reference frame is a right-handed coordinate system where the X is defined parallel to the facing direction, Y is perpendicular to both the X and Z axes and points to the right hand side of the object, Z specifies a vertical direction with positive values pointing downwards.

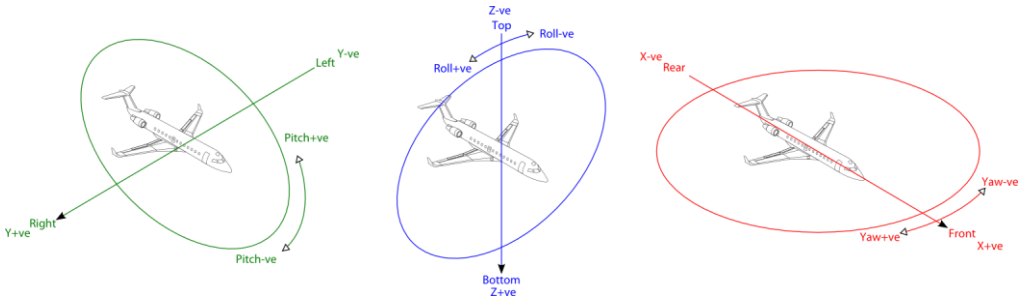


Figure 6 Body Reference Frame

2.3.4 Quantities and Associated Reference Frames

The following table describes the Reference Frame used for the physics attributes of an entity:

Quantity	Reference and units
`EntityPhysicsData.Position`	LLA World Reference Frame: • X: Latitude in radians • Y: Longitude in radians • Z: Altitude in meters

`EntityPhysicsData.WorldVelocity`	Local Tangent Plane Reference Frame: • X: Velocity on the East axis (m/s) • Y: Velocity on the North axis (m/s) • Z: Velocity on the Up axis (m/s)
`EntityPhysicsData.WorldAcceleration`	Local Tangent Plane Reference Frame: • X: Velocity on the East axis (m/s^2) • Y: Velocity on the North axis (m/s^2) • Z: Velocity on the Up axis (m/s^2)
`EntityPhysicsData.Orientation`	Orientation is expressed in radians as Heading, Pitch and Roll. • X: Heading - Angle of X Body Axis (nose/front) relative to North, also a positive (nose/front right) rotation about Z Body axis • Y: Pitch - Angle of X Body Axis (nose/front) relative to horizon, also a positive (nose/front up) rotation about Y Body Axis. • Z: Roll or the Angle of Y Body Axis relative to horizon, also a positive rotation about X Body Axis.
`EntityPhysicsData.BodyAngularVelocity`	Body Reference Frame: • X: Represents Body Angular velocity around its X or forward axis, positive rotating to the right. • Y: Represents Body Angular velocity around its Y or right axis, positive rotating to the right. • Z: Represents Body Angular velocity around its Z or down axis, positive rotating to the right.
`EntityPropertiesData.BoundingBox`	Body Reference Frame: • X: Bounding box size on the x axis, • Y: Bounding box size on the y axis, • Z: Bounding box size on the z axis.

Appendix A contains reference charts for using the LVC coordinate and reference systems with Unity.

2.4 Dead Reckoning

LVC Game reduces network traffic usage by performing Dead Reckoning on entity update events. Several Dead Reckoning algorithms are supported by LVC Game to provide linear, dynamic and rotational components to the forecast movement of each entity (See Software User Manual).

Since Dead Reckoning is specified at the simulation level, your application does not need to perform additional calculations to derive the position and orientation of entities: LVC Game automatically executes Dead Reckoning calculations for incoming and outgoing entity update events which imply the following:

1. When your application invokes `LVCClient.updateEntity()` to move an entity, LVC Game does not necessarily generate a physical update "on the wire" if the entity has not moved beyond a dead reckoning threshold (see `GameClient.config`). Your application is therefore able to repeatedly invoke `LVCClient.updateEntity()` without concern for network flooding.
2. As your application calls `LVCGame.tick()`, artificial incoming `LVCClient.updateEntity()` events are generated for each remote entity, even when no incoming data packets were received "from the wire".

For outgoing entity update events, it is important to correctly specify the velocity, acceleration, orientation and angular velocity vectors associated with an entity.

These attributes must follow the laws of Newtonian physics to avoid errors in the perceived movement and orientation of your entities within other applications connecting via LVC Game.

2.5 Licensing

A valid licence is required to execute LVC Game for Unity.

Please contact Calytrix Technologies via support@calytrix.com if you don't yet hold a developer's licence.

Software developers and partners are usually granted limited licences for the purpose of integration and testing. Licences issued under this context cannot be passed onto End Users.

End Users require a runtime licence issued under specific conditions. Please contact Calytrix Technologies via support@calytrix.com regarding conditions and details of licenses for runtimes distributed to end users.

3 Creating LVC Enabled Games with LVC Game for Unity

Some brief programming guidelines are presented in this section. For in–depth information, please refer to the C# API documentation and example projects provided with the SDK.

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 Setting up the Unity Development Environment

Import the LVC Game for Unity package into a new or existing Unity project.

The imported package will contain the following folders:

`/Plugins`

`/LVCGame`

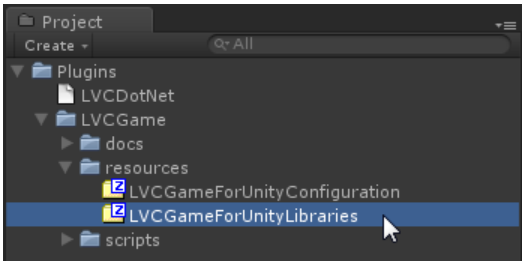
`/docs`	Documentation
`/resources`	Libraries and Configuration Resources
`/scripts`	Sample C# scripts

3.2.1 Install LVC Game for Unity Plugin Libraries for the Unity Editor

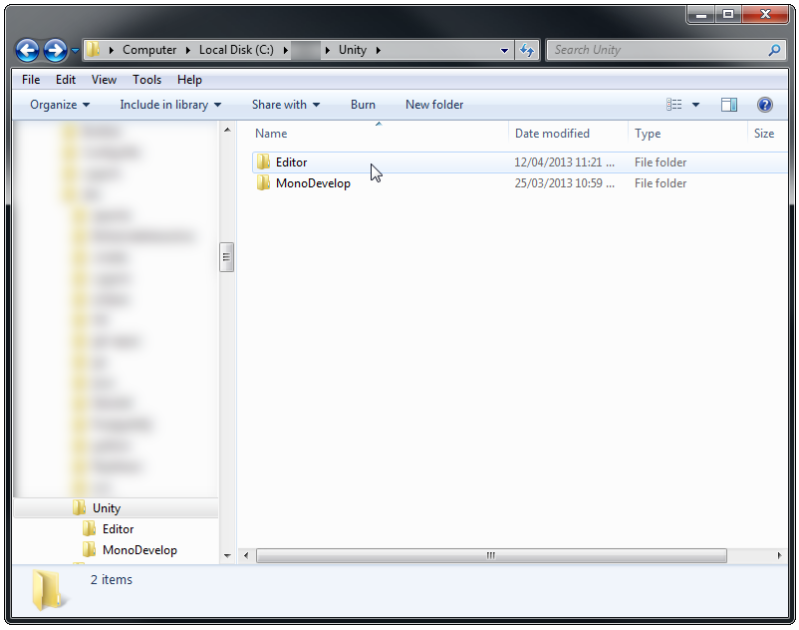
In order for the LVC Game for Unity Plugin to be used within the Unity Editor, some additional libraries are required. These must be placed in the installation directory of Unity itself.

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 as detailed in 3.2.2.

Double click on the `LVCGameForUnityLibraries` zip file in the `resources` folder to open it.



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

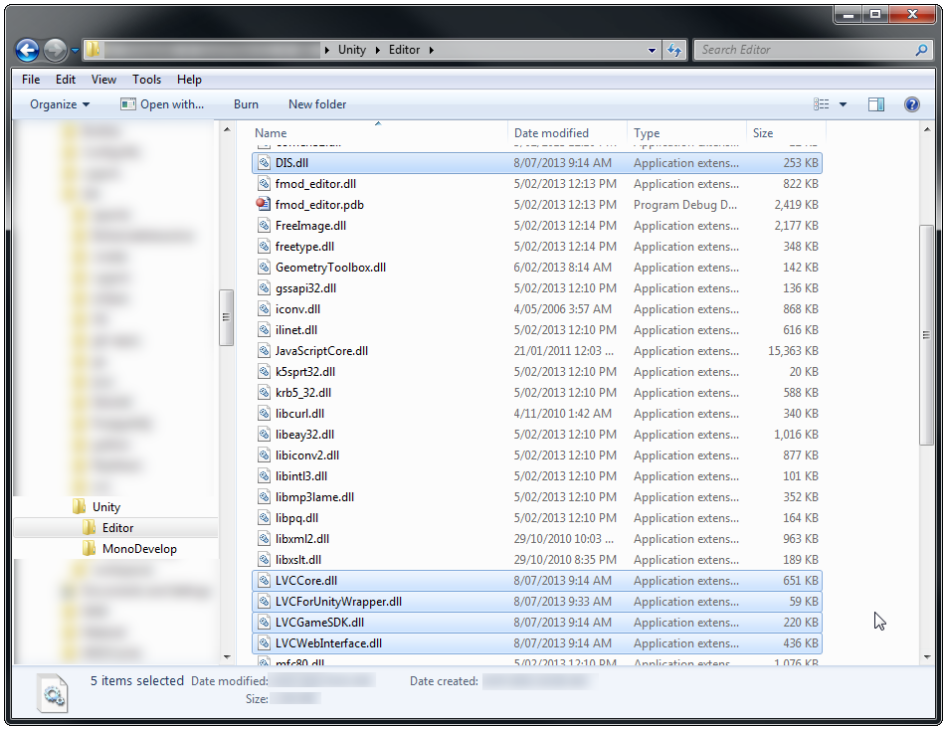


You should see two folders called Editor and MonoDevelop.

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

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

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



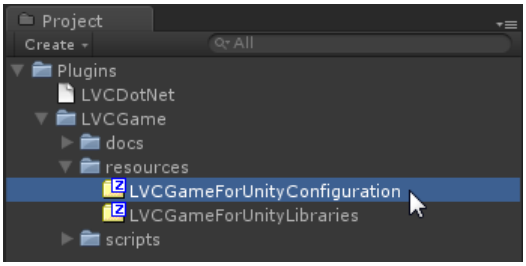
The Unity Editor will need to be restarted at this point – exit Unity, and start it up again.

3.2.2 Add LVC Game for Unity Configuration to the Unity Project

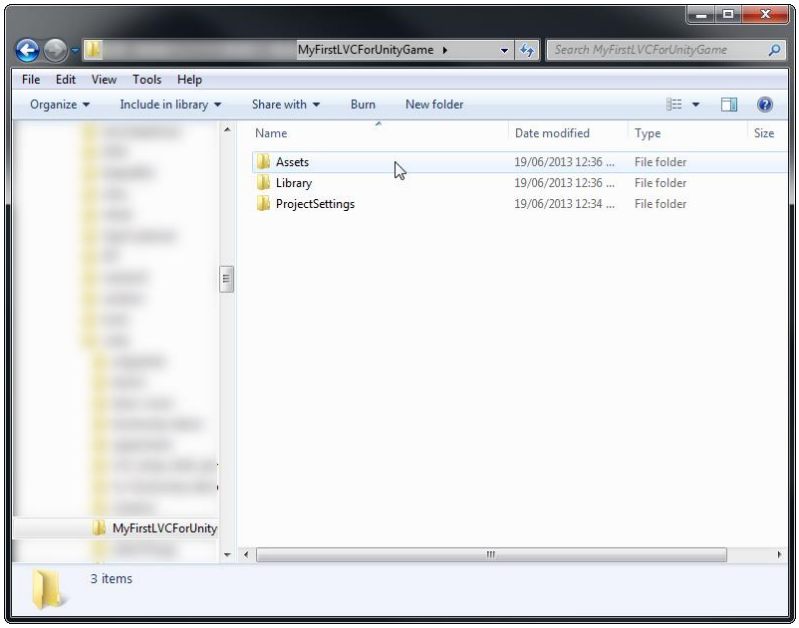
In order to initialise an `LVCClient` instance, several configuration files are required. If this configuration information is not in the expected location, an instantiated `LVCClient` will fail to initialise itself, and will not send or receive LVC events.

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

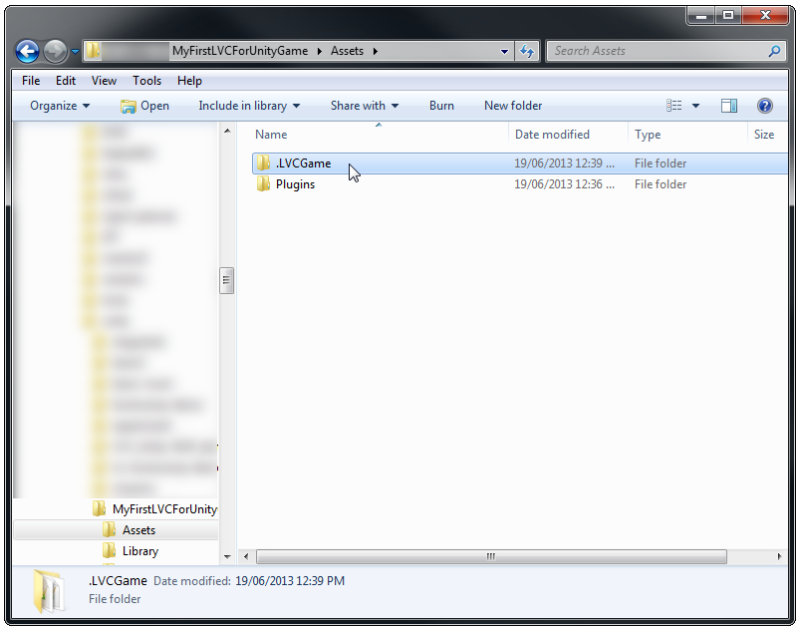
Double click on the `LVCGameForUnityConfiguration` zip file in the `resources` folder to open it.



Use the file explorer to navigate to the directory which contains your 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[1] contains configuration information for the LVC Game for Unity plugin.

Note that the LVC Game for Unity Configuration ZIP file is not required to be in the project after this point, and may be deleted from the project if desired.

[1] 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.

3.3 Creating and initialising an instance of LVCClient

`LVCClient` is the main API entry point for all operations and can be created by instantiating a new `LVCClient` and providing:

- The application name that uses LVC Game, such as “Unity” or “MyGame”. The application name can be any alphanumeric name without spaces. This field is used to reference the configuration files such as *ApplicationName*`_to_lvc.config`, and *ApplicationName*`Client.config`.
- The full path for the LVC Game logging file, including the name of the log file itself.

The following C# code snippet illustrates how `LVCClient` is created:

```
// Either fully qualify all LVC Game types or bring in the LVCGame namespace:
using LVCGame;

…

// Create a handler for incoming LVC events
ILVCEvents myLVCEventsHandler = new MyLVCEventsHandler(); // developer to create

// Create an instance of LVCClient
LVCClient lvcClient = new LVCClient( "Unity", "LVCGame.log" );

// Initialise LVCClient
bool isInitialised = lvcClient.Initialise("<CONFIGURATION PATH>");
if( !isInitialised )
{
        // handle initialisation failure
}

// Set the incoming LVC event callback handler
lvcClient.SetEventsHandlerCallbacks( myLVCEventsHandler, false );

// Start LVC Client
lvcClient.Start();

…

// Finally, shut down the LVC Client when no longer required
lvcClient.Stop()
```

Once `LVCClient` is created, you can initialise it by specifying the location of the LVC Game configuration files (use the sample configuration files distributed with LVC Game as a starting point). Note that a valid licence will be required during the initialisation phase.

An event listener will need to be defined to receive all desired notifications from the network. The `ILVCEvents` interface needs to be implemented as a concrete class in your application, such as:

```
// Basic definition of a listener class to receive incoming LVC events
public class MyLVCEventsHandler : ILVCEvents
{
    // These methods are overridden from ILVCEvents
    // and will be used to notify your application of new Entity
    // creation, and updates and deletion for previously created
    // entities.
    public bool CreateEntity( ref EntityData data )
    {
        …
    }

    public bool UpdateEntity( ref EntityData data )
    {
        …
```

```
    }

    public bool DeleteEntity( long instanceID )
    {
        …
    }
}
```

The above listener can be instantiated and registered with `LVCClient` to start receiving remote notifications:

```
ILVCEvents myLVCEventsHandler = new MyLVCEventsHandler();
bool isAsynchronous = false;
lvcClient.SetEventsHandlerCallbacks( myLVCEventsHandler, isAsynchronous );
```

Note that it is possible to receive remote notifications either synchronously or asynchronously with regards to `LVCClient::tick()` (See API documentation for `SetEventsHandlerCallbacks`).

When `LVCClient` is initialised and an incoming event listener is registered, LVC Game for Unity is ready to start processing outgoing and incoming events.

Conceptually, the following loop is required:

```
// Start LVC Game so all plugins are loaded and placed in the ready state
if( ! lvcClient.Start() )
{
    // handle failure to start
}
else
{
    // While your application is executing, tick LVC Game
    // regularly to process incoming and outgoing events.
    while( isExecuting )
    {
        // Your application performs its own tasks

        …

        // tick LVC Game to trigger event processing
        // (maximum allowance of 50ms in this example)
        lvcClient.Tick( 50 );
    }
}
```

3.3.1 Unity, LVC and Threading

Developers integrating LVC into their Unity game are reminded that the Unity API is not thread-safe. Typically, `LVCClient` should run in a thread separate to that of the Unity game engine. The rationale for this division is to allow LVC to tick itself on a consistent basis, so as to not introduce additional overhead directly to the processing pipeline of Unity, which includes rendering and other internal game calculations. This decoupled design is evident by observing the classes `ExternalLVCEntityHelper` and `LVCUnityAmbassador`.

`ExternalLVCEntityHelper` is a behaviour script component that hooks into the main processing pipeline of Unity, since it is derived from `MonoBehaviour`. It exists to notify LVC of updates to `GameObjects` in a Unity game as they occur. `LVCUnityAmbassador` encapsulates the separate thread of execution that LVC uses. Since it exists outside of the Unity processing pipeline, it is able to freely update itself when necessary.

3.4 LVC Events Management

To send and receive entity information, please ensure your mapping files contain the appropriate entity mappings for incoming and outgoing events, see:

- *GameName*_to_lvc.config
- lvc_to_*GameName*.config

Note that the actual name of the configuration files will depend on the application names supplied to LVCClient when it was created.

For example, if "Unity" were supplied as the application name, LVC Game will expect to find mapping configuration in files named Unity_to_lvc.config and lvc_to_Unity.config.

3.4.1 Entity Creation Example

The following code illustrates how an outgoing entity is created.

Please note that not all fields are filled-in for the purpose of this example, refer to the API documentation for a complete set of available attributes.

It is also assumed that the developer has created a function called makeUniqueInstanceID() which generates a unique numeric instance identifier each time it is called.

```
// Create EntityDataStruct structure and initialise it
LVCGame.EntityData entityData = new LVCGame.EntityData();
entityData.id = new LVCGame.EntityID();
entityData.physics = new LVCGame.EntityPhysics();
entityData.properties = new LVCGame.EntityProperties();

// Assign unique instance ID value, marking and entity type
entityData.id.instance = makeUniqueInstanceID(); // developer to create
entityData.id.gameType = "m1a1";
entityData.id.marking = "tank001";

// Force and appearance
entityData.properties.force = LVCGame.UnitForceType.Opposing;
entityData.properties.appearance = STATUS_FIREPOWER_DISABLED | STATUS_ENGINE_SMOKE;

// Assign Position in LLA coordinates at (lat=0, long=0, atitude=0)
entityData.physics.position.x = 0.0f;
entityData.physics.position.y = 0.0f;
entityData.physics.position.z = 0.0f;

// Assign a linear velocity on the East axis
entityData.physics.worldVelocity.x = 12.0f;
entityData.physics.worldVelocity.y = 0.0f;
entityData.physics.worldVelocity.z = 0.0f;

// Create an Articulated Part
LVCGame.ArticulatedPart articulatedPart = new LVCGame.ArticulatedPart();
articulatedPart.typeDesignator = LVCGame.ParameterType.ArticulatedPart;
articulatedPart.change = 0;
articulatedPart.partID = 0;
articulatedPart.partsType = LVCGame.ArticulatedPartsType.PrimaryTurretNumber1;
articulatedPart.metricType = LVCGame.ArticulatedTypeMetricType. Azimuth;
articulatedPart.value = 10.0f;

// Create array of articulated parts (only one in this case)
LVCGame.ArticulatedPart[] articulatedParts = new LVCGame.ArticulatedPart[1];
articulatedParts[0] = articulatedPart;

// Convert array of articulated part structs to IntPtr
int apStructSize = Marshal.SizeOf( typeof(LVCGame.ArticulatedPart) );
int totalSize = apStructSize * articulatedParts.Length;
IntPtr pUnmanagedArticulatedParts = Marshal.AllocHGlobal( totalSize );
IntPtr currentPtr = new IntPtr( pUnmanagedArticulatedParts.ToInt32() );
for( int i=0; i<articulatedParts.Length; i++ )
{
    Marshal.StructureToPtr( articulatedParts[i], currentPtr, true );
    currentPtr = new IntPtr( currentPtr.ToInt32() + apStructSize );
}
```

```
// Assign the articulated part to the Entity
entityData.articulatedParts.numParts = articulatedParts.Length;
entityData.articulatedParts.part = pUnmanagedArticulatedParts;

// Create the entity
if( ! lvcClient.createEntity(ref entityData) )
{
    // handle failure to create
}
```

Note that `EntityData.id.instance` *must* be assigned a *unique* instance ID and is used to track an entity instance during its lifetime.

This requirement applies to outgoing and incoming events. In the case of incoming events, failing to assign a unique instance identifier to `EntityData.id.instance` will result in a failure to receive subsequent update and delete events.

3.4.2 Entity Update Example

The following code shows how the position and velocity of an entity are updated.

In this example, it is assumed that the developer has created a function called `getEntityDataFor( instanceID )`, which retrieves the appropriate `EntityData` structure from a cache:

```
// You can retrieve the entity data associated with a known instance ('1' in this example)
long instanceID = 1;
LVCGame.EntityData entityData = getEntityDataFor( instanceID ); // developer to create

// Update altitude to 10 meters above ground
entityData.physics.position.z = 10.0f;

// Update velocity in the z direction (example only; ensure you use real physics)
entityData.physics.velocity.z = 2.0f;

// Send update event
if( ! lvcClient.updateEntity(entityData) )
{
    // handle failure to update
}
```

3.4.3 Entity Delete Example

An entity is deleted by de-referencing its instance identifier:

```
long uniqueInstanceID = 1;
// send delete event
if( ! lvcClient.deleteEntity(uniqueInstanceID) )
{
    // handle failure to delete
}
```

Invoking `LVCClient.deleteEntity()` to create an outgoing event causes remote systems to delete the specified entity. Similarly your application should react to incoming delete events and remove references to the specified entity instance.

3.4.4 Fire and Detonations Example

Fire events are generated when an entity fires a weapon; detonation events are generated when ammunitions hit a target or detonate.

These events are typically used by simulation systems to evaluate entity and environment damage assessment.

To generate a Fire event, an existing entity and desired ammunition type must be selected. Typically, the following sequencing must be observed:

1. Fire and Detonation events must be paired when fired ammunition hits a target, linked via `TargetingData.eventID`. That is, each Fire event must be accompanied with a corresponding Detonation event.
2. Where a Fire causes no detonation (a misfire, or a missed target, for example), no detonation events are generated.
3. Detonation events may be generated on their own without a pre-existing Fire event where no weapon was fired in the first instance (example: IED). In this case, `TargetingData.eventID` is set to zero on the Detonation event.

The following pseudo code illustrates how a Fire Event is generated:

```
// The unique entity instance ID for the entity which fired
firingEntityInstanceID = 1;

// Create the data container for this event
LVCGame.FireWeaponData fireData = new LVCGame.FireWeaponData();

// Obtain the next firing event ID
fireData.targeting.eventID = getNextFireEventID(); // developer to create
fireData.targeting.firingId = firingEntityInstanceID;

// Fill in the location of the firing event
fireData.targeting.position = <Use location of weapon>;

// Fill fire velocity, as 1000m/s on the X axis
fireData.Targeting.Velocity.x = 1000.0f;
fireData.Targeting.Velocity.y = 0.0f;
fireData.Targeting.Velocity.z = 0.0f;

// Which ammunition was fired?
fireData.targeting.munitionType = "155mm Impact How(M107)";

// Additional information
fireData.d escriptor.fuse = LVCGame.FuseType.Multifunction;
fireData.descriptor.quantity = 1;
fireData.descriptor.rate = 1;
fireData.descriptor.warHead = LVCGame.WareHeadType_Kinetic;

// Send the fire event
if ( ! lvcClient ->fireWeapon(data) )
{
    // Handle failure to send fire event
}
```

4 Example LVC Game for Unity Projects

An example is distributed with the LVC Game for Unity package to illustrate how a simple LVC enabled Unity game can be created with LVC Game for Unity. It uses the example scripts provided in the package.

The following diagram is a high level overview of the Unity Quick Start Guide example – please refer to the guide itself for step-by-step instructions.

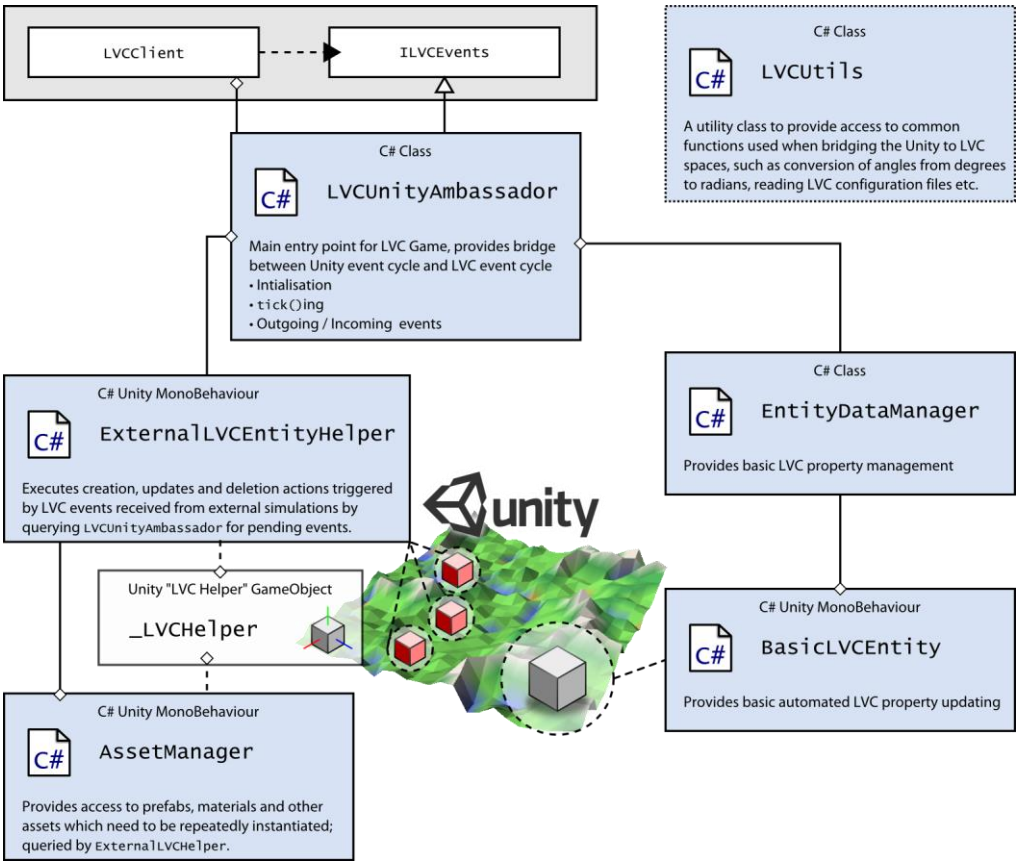


Figure 7 Quick Start Guide Example Overview

5 Example C# Scripts

5.1 AssetManager

This is a `MonoBehaviour` which provides a way to manage standard prefabs, materials and other resources which must be instantiated during the course of an LVC Game.

The primary purpose is associate a Unity game type name, such as '`m1a1`', with the prefab representing that type within Unity.

More concretely, when the `ExternalLVCEntityHelper` receives notification of a new entity being created in the simulation, it asks the `AssetManager` to provide the appropriate Unity prefab for it. The prefab is then instantiated to represent that entity in Unity. If no such prefab exists, the `ExternalLVCEntityHelper` will fall back to using either the Default Prefab (if one is defined), or a standard grey Unity cube.

Like the `ExternalLVCEntityHelper`, this behaviour should be attached to a *single* Unity `GameObject` acting as an "LVC Helper". This `GameObject` is invisible in the game, and does not directly take part in gameplay.

Usually this is done as follows:

- Go to the `GameObject` menu in the Unity editor
- Select "`Create Empty`"
- Rename the created `GameObject` "`_LVCHelper`"
- Drag this behaviour on top of that empty object

You may also wish to position this helper object at some "extreme" coordinates, such as (1000, 1000, 1000), to keep it out of the way and avoid cluttering up the scene editor view, but this is not necessary.

See also: `ExternalLVCEntityHelper`

5.2 BasicLVCEntity

This is a simple `MonoBehaviour` which monitors the position and orientation of a Unity `GameObject` and sends out "automated" LVC updates containing this information.

Compare this with `ExternalLVCEntityHelper` which handles incoming updates related to entities controlled by an external simulation system.

`BasicLVCEntity` can be used to make a Unity `GameObject` a part of an LVC Game simply by dragging this behaviour onto the `GameObject` in the `Hierarchy` view of the Unity Editor.

The updates sent by `BasicLVCEntity` populate the following `EntityData` properties with values obtained by interrogating the Unity `GameObject`:

- `position`
- `orientation`
- `worldVelocity`
- `worldAcceleration`

- bodyAngularVelocity

The following values should also be set (manually) as script parameters in the Inspector pane:

- gameType (default: "UNKNOWN TYPE")
- marking (default: "UNMARKED")
- force (default: “Neutral”)

BasicLVCEntity handles the generation of the unique entity ID value.

These automated updates are triggered by any one of the following conditions:

- A certain time interval has passed. By default this is 1 second.
- The game object has moved more than a certain distance from its last reported position. By default this is 1.0 units (i.e., 1 metre).
- The game object has rotated more than a certain angle from its last reported position. By default this is 1.0 degrees.

These thresholds may be set (manually) as script parameters in the Inspector pane.

Though there are many other properties which can be set for LVC Game entities, values other than those listed above are not populated/modified in the updates.

This is because other properties (such as lights, damage state, stance and so on) are very specific to the underlying implementation of the entity in question and, cannot be obtained by simply interrogating the GameObject.

For example, while it can comfortably be said that all entities must have a position and orientation, not entities will necessarily have lights, a secondary weapon or a concept of stance.

Handling of other properties must be implemented by the developer. Indeed, the BasicLVCEntity MonoBehaviour would likely be used only for initial testing, with a customised handler developed to suit particular entities in the final game.

5.3 EntityDataManager

This is a utility class which encapsulates the LVC entity data for a Unity GameObject representation of an LVC entity.

It should be used as a proxy to communicate entity state to a LVC simulation system.

Refer to BasicLVCEntity for example usage.

5.4 ExternalLVCEntityHelper

This is a MonoBehaviour which provides basic "automated" handling of creation, update, deletion, weapon fire and detonation information contained in *incoming* LVC traffic.

Like the AssetManager, this behaviour should be attached to a *single* Unity GameObject acting as an "LVC Helper". This GameObject is invisible in the game, and does not directly take part in gameplay.

Usually this is done as follows:

- Go to the GameObject menu in the Unity editor
- Select "Create Empty"
- Rename the created GameObject "_LVCHelper"
- Drag this behaviour on top of that empty object

You may also wish to position this helper object at some "extreme" coordinates, such as (1000, 1000, 1000), to keep it out of the way and avoid cluttering up the scene editor view, but this is not necessary.

Compare this with BasicLVCEntity, which is assigned to *multiple* individual Unity GameObjects and provides basic automated *outgoing* LVC event updates related to these Unity controlled entities.

The ExternalLVCEntityHelper internally maintains a reference to the LVCUnityAmbassador singleton, and the AssetManager.

During the Unity FixedUpdate() cycle, ExternalLVCEntityHelper queries LVCUnityAmbassador for pending updates in its LVC event queues which need processing.

It then provides the concrete handling of these events by creating, updating and deleting entities, and also managing weapon fire and detonations.

For entity creation events ExternalLVCEntityHelper queries the AssetManager for a prefab matching the game type of the created entity. If a suitable prefab is returned, a copy of the prefab is instantiated to represent the entity. If a suitable prefab is not found, the configured Default Prefab is used. If the Default Prefab is not configured, a standard grey Unity “Cube” GameObject is used to represent the entity.

See also: LVCUnityAmbassador, AssetManager

5.5 LVCExtraFireHandler

An abstract class defining the interface which must be realised for an LVCExtraFireHandler.

A realised LVCExtraFireHandler provides handling specific to an entity type for LVC fire events.

For example, an LVCExtraFireHandler could be used to create a muzzle flash from the main barrel of an M1A1 tank representation, or emit shell casings from the secondary machine gun.

5.6 LVCExtraUpdateHandler

An abstract class defining the interface which must be realised for an LVCExtraUpdateHandler.

A realised LVCExtraUpdateHandler provides handling specific to an entity type for LVC update events.

For example, an LVCExtraUpdateHandler could be used to inspect articulation data and rotate the turret of an M1A1 tank representation, or emit a dust trail if the tank is moving.

5.7 LVCGameTypes

This class is intended as a centralised "repository" of LVC Game type names and should match up with:

- the names used for DIS enumerations in the "`lvc_to_Unity.config`" file
- the key values populated in the `AssetManager`

The main aim is to avoid hard coded strings appearing everywhere, requiring a change in a name to be corrected everywhere it occurs. By using the static string constants defined in this class, a name change only needs to be corrected in this class.

There is no necessity to use this class, but generally speaking it tends to keep the code cleaner and easier to maintain.

5.8 LVCPair

A simple class to hold a pair of (generic) items.

5.9 LVCUnityAmbassador

This class is used to obtain a reference to the `LVCClient` instance (through which notifications of entity creation, updates and deletions may be sent).

It implements the `ILVCEvents` interface, allowing it to register itself to respond to incoming LVC events from external simulations.

`LVCUnityAmbassador` also…

- Provides the `GetNextEntityID()` method to obtain unique entity IDs for new simulation entity instances
- Provides the `GetNextTargetingEventID()` method to obtain unqiue targeting event IDs for fire and detonation events.
- Maintains a table matching unique entity IDs to the `EntityData` of the entity, and methods for interrogating this table
- Maintains a table matching `GameObjects` to with the unique entity ID of the LVC Game entity they represent, and methods for interrogating this table

The `LVCUnityAmbassador` is a singleton instance – it may be obtained at any time using the `GetInstance()` method.

When the `LVCUnityAmbassador` singleton is instantiated by the first `GetInstance()`, it…

- Creates and initialises an `LVCClient` instance
- Registers *itself* with the `LVCClient` instance to handle incoming LVC events
- Begins `tick()`ing the `LVCClient` instance so that it processes events

During execution it…

- Periodically `tick()`s the `LVCClient` so that it processes events
- Stores *incoming* event information in separate FIFO queues for each of the following `ILVCEvent` events:
    - Entity creation, i.e., `CreateEntity( ref EntityData data )`

- Entity update, i.e., `UpdateEntity( ref EntityData data )`
- Entity deletion, i.e., `DeleteEntity( long id )`
- Weapon fire, i.e., `FireWeapon(ref FireWeaponData data )`
- Detonation, i.e., `DetonateMunition(ref DetonateMunitionData data )`

Note that `LVCUnityAmbassador` does not actually provide any handling for any of the incoming events, but merely stores the events as they arrive in the queues. The queues may be interrogated by the following `LVCUnityAmbassador` methods:

- `public EntityData[] GetPendingRemoteCreates()`
- `public EntityData[] GetPendingRemoteUpdates()`
- `public long[] GetPendingRemoteDeletes()`
- `public FireWeaponData[] GetPendingRemoteFires()`
- `public DetonateMunitionData[] GetPendingRemoteDetonations()`

Calling any of these methods will…

- return an array containing data for the corresponding events, and
- ***clear the corresponding event queue***

It is expected that the caller will handle any processing required to deal with the events[2].

Note that care should be given to the order in which the events are processed – generally speaking the event processing order should be as follows:

- Creations
- Updates
- Deletions
- Fires
- Detonation

When `ShutDown()` is called on the `LVCUnityAmbassador` instance, it…

- Stops `tick()`ing the `LVCClient` instance
- `Stop()`s the `LVCClient` instance

Note that `Stop()`ing the `LVCClient` instance also causes deletion events to be sent out for all remaining entities.

5.10 LVCUtils

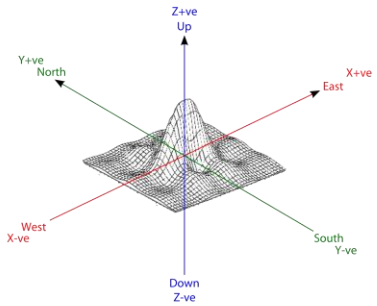
A utility class which provides access to common functions used when bridging Unity and LVC. These include functions such as conversion of angles from degrees to radians, converting coordinate systems, reading LVC configuration files, and so on.

[2] Refer to the `ExternalLVCEntityHelper` MonoBehaviour, which queries the `LVCUnityAmbassador`'s queues periodically (during the Unity `FixedUpdate()` cycle) to provide appropriate handling for these queued LVC events received from the network.

6 Appendix A

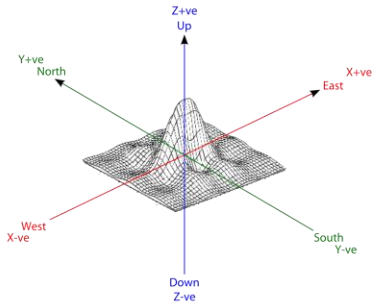
The following diagrams visually illustrate the coordinate systems in use by LVC Game and Unity. They also provide a guide for conversion between the two systems.

LVC Game Local Tangent Plane Coordinate System



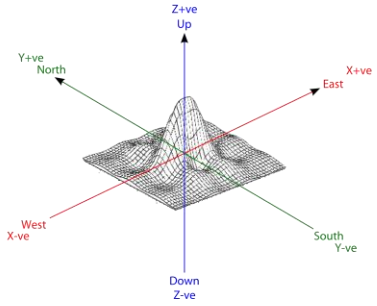
LVC ► Unity

X^{rad}_{long}	X_m
Y^{rad}_{lat}	Z_m
Z_m	Y_m



LVC ► Unity

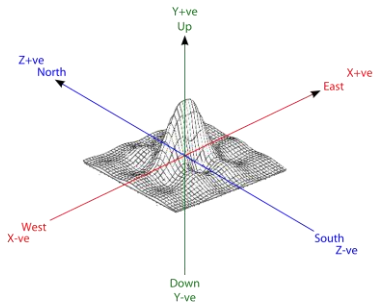
$X_{m/s}$	$X_{m/s}$
$Y_{m/s}$	$Z_{m/s}$
$Z_{m/s}$	$Y_{m/s}$



LVC ► Unity

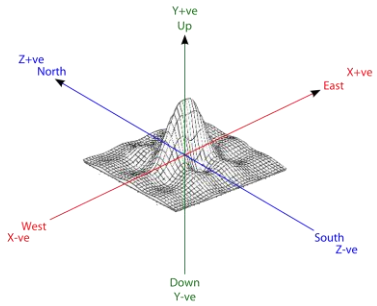
X_{m/s^2}	X_{m/s^2}
Y_{m/s^2}	Z_{m/s^2}
Z_{m/s^2}	Y_{m/s^2}

Unity World Coordinate System



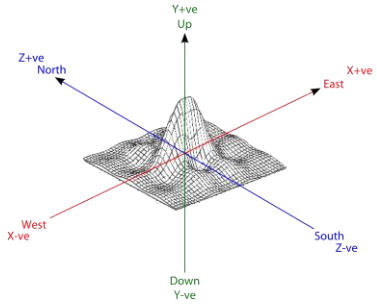
Unity ▶ LVC

X_{m}	X^{rad}_{long}
Y_{m}	Z_{m}
Z_{m}	Y^{rad}_{lat}



Unity ▶ LVC

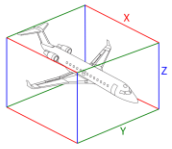
$X_{m/s}$	$X_{m/s}$
$Y_{m/s}$	$Z_{m/s}$
$Z_{m/s}$	$Y_{m/s}$



Unity ▶ LVC

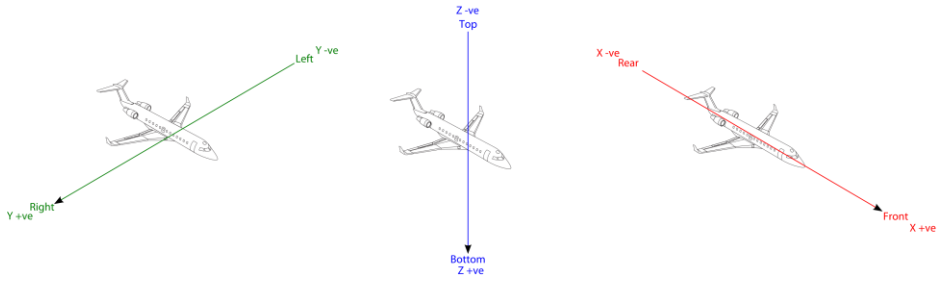
X_{m/s^2}	X_{m/s^2}
Y_{m/s^2}	Z_{m/s^2}
Z_{m/s^2}	Y_{m/s^2}

LVC Game Body Reference Frame



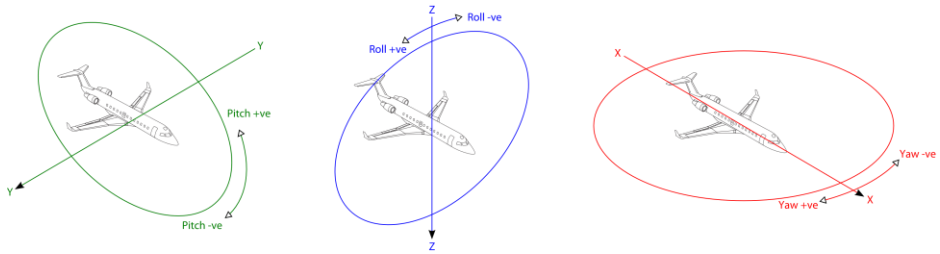
LVC ► Unity

X_m	X_m
Y_m	Z_m
Z_m	Y_m



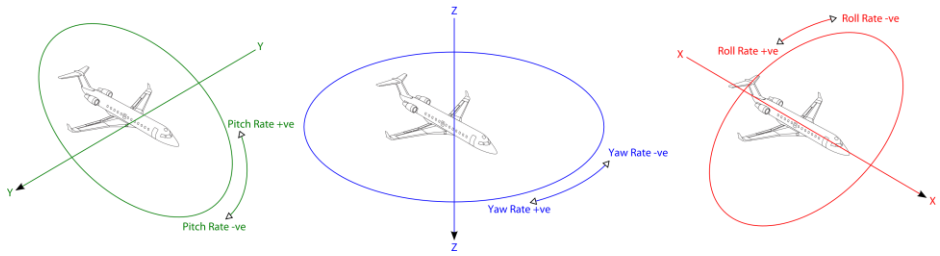
LVC ► Unity

X_m	X_m
Y_m	$-Z_m$
Z_m	$-Y_m$



LVC ► Unity

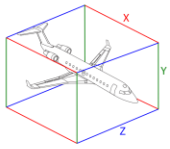
X^{rad}	$Y°$
Y^{rad}	$Z°$
Z^{rad}	$-X°$



LVC ► Unity

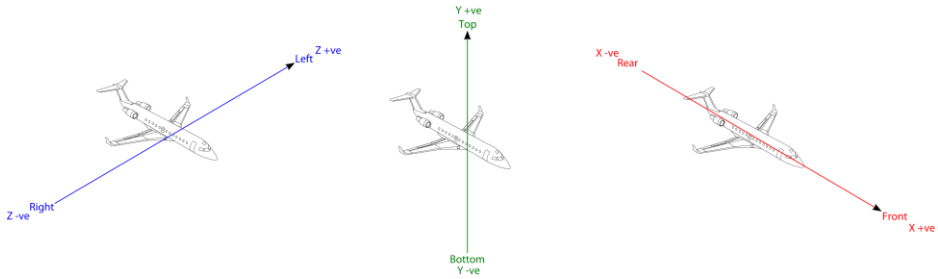
$X^{rad}_{/sec}$	$-X°_{/sec}$
$Y^{rad}_{/sec}$	$Z°_{/sec}$
$Z^{rad}_{/sec}$	$Y°_{/sec}$

Unity Body Reference Frame



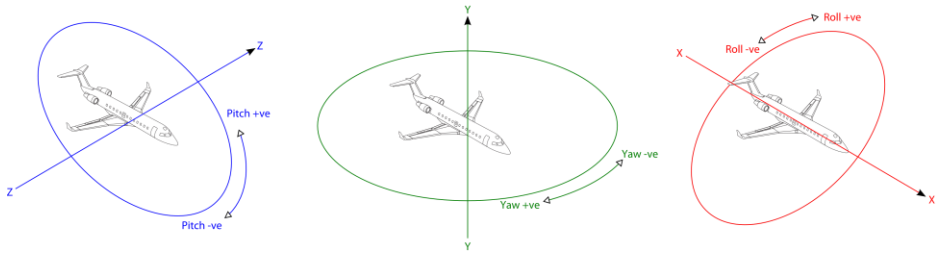
Unity ► LVC

X_m	X_m
Y_m	Z_m
Z_m	Y_m



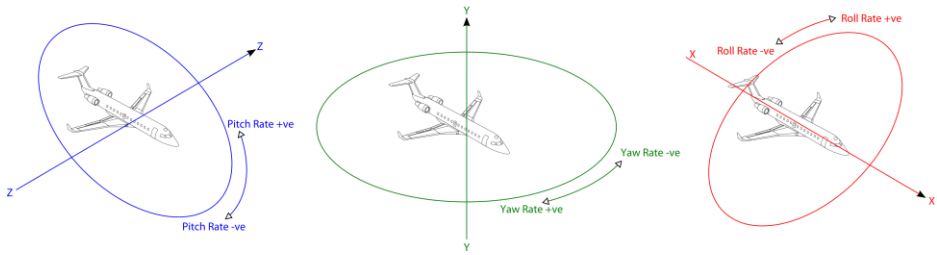
Unity ► LVC

X_m	X_m
Y_m	$-Z_m$
Z_m	$-Y_m$



Unity ► LVC

X°	$-Z^{rad}$
Y°	X^{rad}
Z°	Y^{rad}



Unity ► LVC

$X^\circ_{/sec}$	$-X^{rad}_{/sec}$
$Y^\circ_{/sec}$	$Z^{rad}_{/sec}$
$Z^\circ_{/sec}$	$Y^{rad}_{/sec}$