Manual

Version 1.2

Asset Store Description

An Open Sound Control (OSC) implementation tailored for Unity. OSC is a protocol for communicating between applications and devices easily using URL-style messages with mixed argument types.

Supports

- All OSC argument types (except MIDI)
- Bundles with timetags
- UDP IPv4 Unicast, Broadcast and Multicast
- Unity's .Net 2.0 Subset
- OSX, Windows and probably other platforms

Includes

- Manual
- Reference
- Examples
- Full source code
- Runtime UI prefabs

Features

- Map addresses from scripts and inspector
- Monitor messages
- Monitor remote connection status
- Filter message duplicates (optional)
- Auto bundle messages (optional)
- Add timetag to bundled message (optional)
- Disable multicast loopback (optional)

Tested with

OpenFrameworks, Processing, Max/MSP, VVVV, TouchOSC, Lemur, Iannix and Vezer.

Underlying Concepts

OSC simpl transmits messages targeting IPv4 addresses over unicast, broadcast and multicast UDP.
If those words sound familiar to you then skip this page.

OSC

Open Sound Control (OSC) is network protocol initiated in 1997 and developed at Center for New Music and Audio Technologies (CNMAT). All about OSC at http://opensoundcontrol.org/

UDP

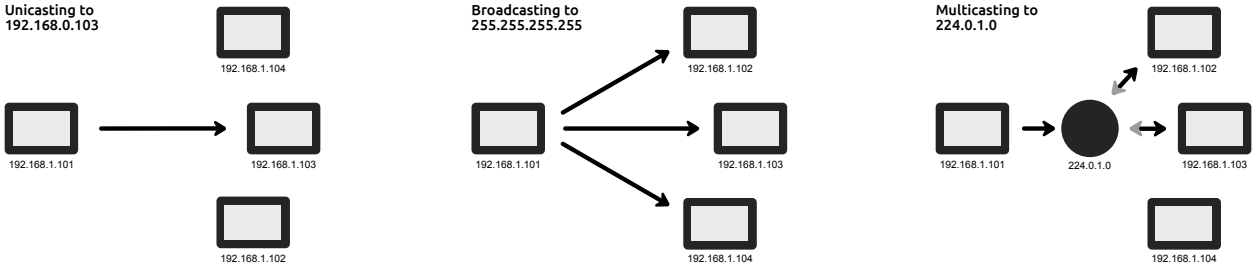
The User Datagram Protocol (UDP) is an network protocol that offers very fasts but unreliable transmission. Contrary to the The File Transfer Protocol (FTP), UDP has no native mechanism for checking if messages reach their destination. No connection is established. The messages are simply send to a network destination. All about UDP at https://en.wikipedia.org/wiki/User_Datagram_Protocol

IPv4

The Internet Protocol version 4 (IPv4) is a version of the Internet Protocol (IP). For the purpose of *OSC simpl*, all we need to know is that IPv4 defines the format of the IP addresses we use to target devices. The format is XXX.XXX.XXX.XXX, where XXX is an integer between 0 to 255. For example; 192.168.1.101.

Unicast, Broadcast and Multicast

Three modes of transmission is available unicast, broadcast and multicast. For all modes, the sender needs to provide an IPv4 address, that will define the mode of transmission, and a port number that applications on the target device can listen on.



Unicast

Unicast transmission is used for targeting a singe device. A sender needs to know the IP address of the target device and a port number for applications to listen on. The address 127.0.0.1 is called the loopback address, and is used for sending to other applications on the same device. Unicasting is the fastest and most reliable mode of transmission.

For applications where low latency is critical, use unicast. Unless you are sending to many devices, unicast offers better performance than broadcast and multicast.

Broadcast

Broadcast transmission is used for targeting all devices on the local network. A sender must send to the global address 255.255.255.255. Broadcasting is the slowest method of sending, but it is useful because the sender does not need to know any IP addresses.

Multicast

Multicast is used for targeting a "multicast group" that applications on other devices may be listening to. A sender must know a valid multicast address, ranging between 224.0.0.0 and 239.255.255.255. A receiver must also know the multicast address. Multicasting is potentially faster than broadcasting, but slower than unicasting.

For multicast to work properly, all routers involved must be "multicast enabled" (most routers are).

Getting started

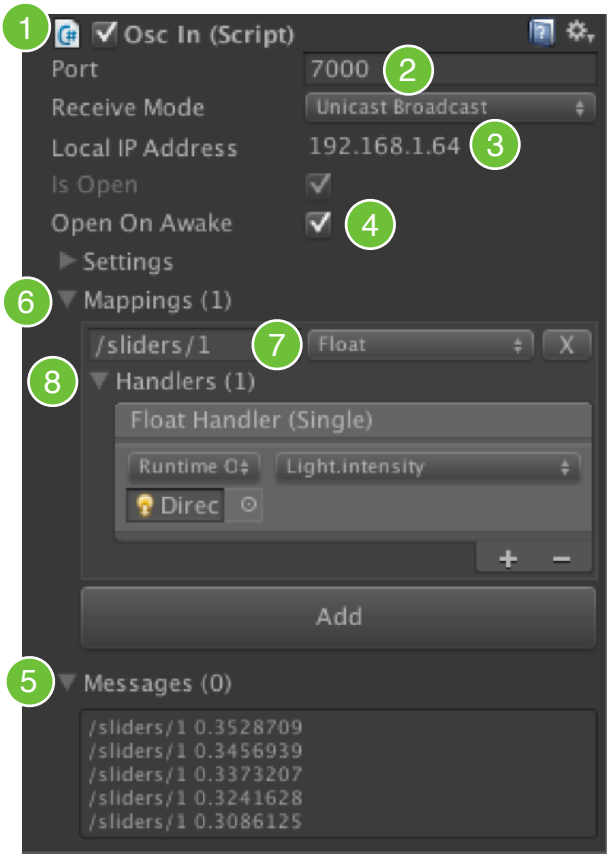
Here is a step by step guide to get started receiving and sending OSC messages, assuming you know the basics of OSC, UDP and IPv4. If not, then read the previous section *Underlying Concepts* and follow the links.

In most cases, you will want to script your OSC mappings to be set up in runtime. Check out the examples provided with the package and the Reference.pdf to see how. If you love clicking and dragging then please proceed below.

How to receive without scripting

When receiving messages that contain a single argument then you can map (route a message to a method) entirely without scripting.

1) Add a OscIn component to a GameObject in your scene.
2) Make sure that *Port* is set to the port number the remote application is sending to.
3) Make sure that the remote application is sending to the *Local IP Address*. If the field spells *"Not found"* then find your IP address in your network settings for your OS.
4) Enable *Open On Awake*.
5) Optionally, unfold *Messages* to see incoming messages.
6) Unfold *Mappings* and press the *Add* button to create a new mapping.
7) Click the text field of your new mapping under *Mappings*, type the desired OSC address and set the expected argument type in the dropdown (in this case *Float*).
8) Unfold *Handlers* under your mapping and set up a Unity Event like you usually do in Unity. In this case we control the intensity of a light in the scene.
9) Press Play.



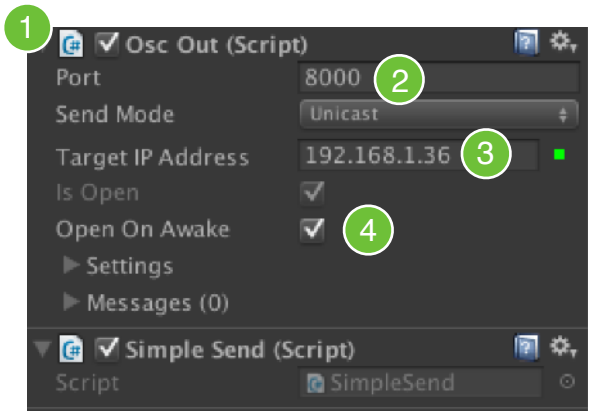
How to send almost without scripting

To send messages you have to create a script that defines what you will be sending.

Here is a c# script that sends a message with address "sliders/1" and one float argument with a random value. The script assumes that is it put on a GameObject that has a OscOut component that has already been set up.

```
[RequireComponent(typeof(OscOut))]
public class SimpleSend : MonoBehaviour
{
    OscOut oscOut;

    void Start(){
        oscOut = gameObject.GetComponent<OscOut>();
    }

    void Update(){
        oscOut.Send( "/sliders/1", Random.value );
    }
```

1) Add a OscOut component and the SimpleSend script above to a GameObject in your scene.
2) Make sure that *Port* is set to the port number the remote application is listening to.
3) Make sure that *Target IP Address* is set to the address of the target device. The square LED to the right will emit green if the device is responding.
4) Enable *Open On Awake*.
5) Press Play.



How to receive using scripting

Script Execution Order

OscIn needs to be executed first to ensure low latency. When adding a *OscIn* component in the Unity Editor it is set automatically. When adding the component from a script you have to do it manually, just once. Find Unity's MonoManager in top the menu: Edit/Project Settings/Script Execution Order.

From included example at path *OSC simpl/Examples/01 GettingStarted/*.

```
public class GettingStartedReceiving : MonoBehaviour
{
    public OscIn oscIn;


    void Start()
    {
        // Ensure that we have a OscIn component.
        if( !oscIn ) oscIn = gameObject.AddComponent<OscIn>();

        // Start receiving from unicast and broadcast sources on port 7000.
        oscIn.Open( 7000 );
    }


    void OnEnable()
    {
        // You can "map" messages to methods in three ways:

        // 1) For messages with one argument, simply provide the address and
        // a method with one argument. In this case, OnTest1 takes a float argument.
        oscIn.MapFloat( "/test1", OnTest1 );

        // 2) The same can be achieved using a delgate.
        oscIn.MapFloat( "/test2", delegate( float value ){ Debug.Log( "Received: " + value ); });

        // 3) For messages with multiple arguments, provide the address and a method
        // that takes a OscMessage object argument, then process the message manually.
        // See the OnTest3 method.
        oscIn.Map( "/test3", OnTest3 );
    }


    void OnDisable()
    {
        // If you want to stop receiving messages you have to "unmap".

        // For mapped methods, simply pass them to Unmap.
        oscIn.Unmap( OnTest1 );
        oscIn.Unmap( OnTest3 );

        // For mapped delegates, pass the address. Note that this will cause all mappings
        // made to that address to be unmapped.
        oscIn.Unmap( "/test2" );
    }


    void OnTest1( float value )
    {
        Debug.Log( "Received: " + value );
    }


    void OnTest3( OscMessage message )
    {
        // Get string arguments at index 0 and 1 safely.
        string text0, text1;
        if( message.TryGet( 0, out text0 ) && message.TryGet( 1, out text1 ) ){
            Debug.Log( "Received: " + text0 + " " + text1 );
        }

        // If you wish to mess with the arguments yourself, you can.
        foreach( object a in message.args ) if( a is string ) Debug.Log( "Received: " + a );

        // NEVER DO THIS AT HOME
        // Never cast directly, without ensuring that index is inside bounds and encapsulating
        // the cast in try-catch statement.
        //float value = (float) message.args[0]; // No no!
    }
}
```

How to send using scripting

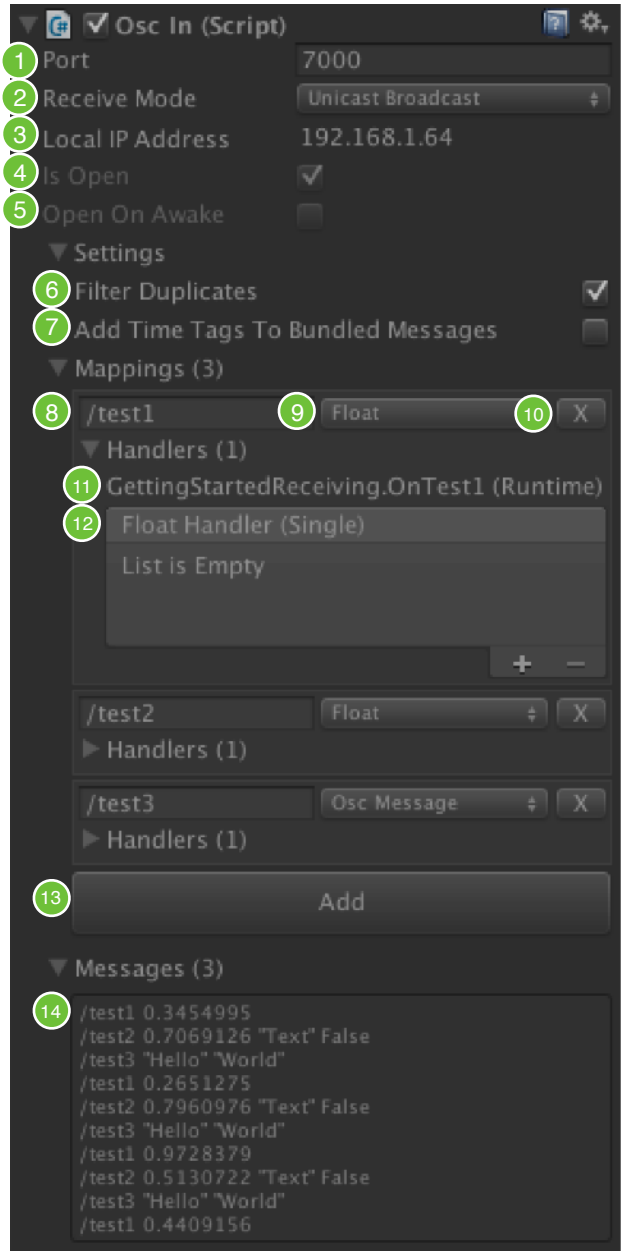
From included example at path *OSC simpl/Examples/01 GettingStarted/*.

```
public class GettingStartedSending : MonoBehaviour
{
    public OscOut oscOut;


    void Start()
    {
        // Ensure that we have a OscOut component.
        if( !oscOut ) oscOut = gameObject.AddComponent<OscOut>();

        // Prepare for sending messages to applications on this device on port 7000.
        oscOut.Open( 7000 );

        // Or, to a target IP Address (Unicast).
        //oscOut.Open( 7000, "192.168.1.101" );

        // Or to all devices on the local network (Broadcast).
        //oscOut.Open( 7000, "255.255.255.255" );

        // Or to a multicast group (Multicast).
        //oscOut.Open( 7000, "224.1.1.101" );
    }


    void Update()
    {
        // Send a message with one float argument.
        oscOut.Send( "/test1", Random.value );

        // Send a message with a number of assorted argument types.
        oscOut.Send( "/test2", Random.value, "Text", false );

        // Create a message and send it.
        OscMessage message = new OscMessage( "/test3" );
        message.Add( "Allo" );
        message.Add( "World" );
        message.args[0] = "Hello"; // Let's say we want overwrite the first argument
        oscOut.Send( message );
    }
}
```

Inspector Panel Reference

Below is a description of each field in the inspector panels for the components *OscIn* and *OscOut*. In most cases, the fields are exactly the same as the public properties found in the included Reference.pdf.

OscIn



Port (1)
The local network port that this component is set to listen on.

Receive Mode (2)
The type of transmission the component will listen to. If set to UnicastBroadcastMulticast, and additional Multicast Address field needs to be filled.

Local IP Address (3)
The local network IP address that this device will listen on. If the field displays "Not Found" then find your local IP address in the network settings for your OS. In some situations, like when you use a DNS, the IP can't be found by *OSC simpl*.

Is Open (4)
Indicates whether the component is open for incoming messages.

Open On Awake (5)
When enabled, the component will automatically open when it gets the Awake call from Unity. Default is false.

Filter Duplicates (6)
When enabled, only one message per OSC address will be forwarded every Update call. The last (newest) message received will be used. Default is true.

Add Time Tags To Bundles Messages (7)
When enabled, timetags from bundles are added to contained messages as last argument. Incoming bundles are never exposed, so if you want to access a time tag from a incoming bundle then enable this. Default is false.

(8), (9) and (10)
OSC address, expected OSC argument type and remove button for a mapping.

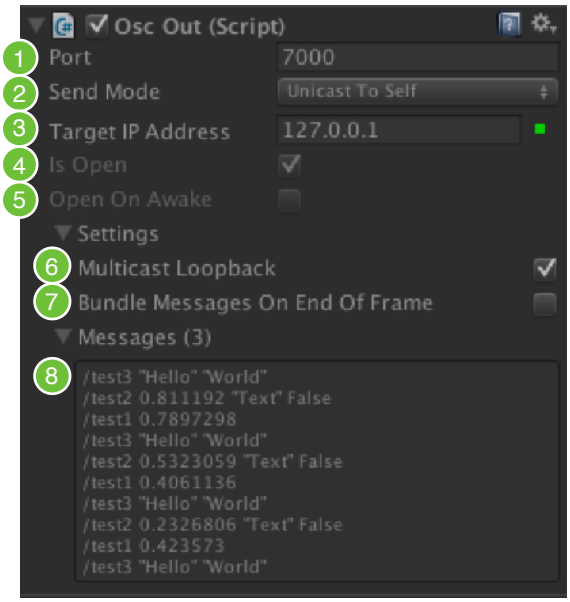
(11)
List of runtime handlers; mappings created from scripts.

(12)
List of editor handlers; mappings created from in the editor.

(13)
Add new mapping button.

(14)
Monitoring of incoming messages.

OscOut



Port (1)
The remote network port that this component is set to send to.

Send Mode (2)
The type of transmission the component will send. The mode is derived from the *Target IP Address* field.

Target IP Address (3)
The remote network IP address that this device will send to.

Is Open (4)
Indicates whether the component is open and ready to send messages.

Open On Awake (5)
When enabled, the component will automatically open when it gets the Awake call from Unity. Default is false.

Multicast Loopback (6)
Indicates whether outgoing multicast messages are also delivered to the sending application. Default is true.

Bundle Messages On End Of Frame (7)
When enabled, messages will automatically be buffered in a single OscBundle and send at the end of the frame (i.e. Unity's WaitForEndOfFrame). Default is false.

(8)
Monitoring of outgoing messages.

Specific topics

Timetags

In *OSC simpl* timetags are represented by the type OscTimeTag. OscTimeTag objects are send implicitly with bundles and explicitly as message arguments.

Incoming bundles are never exposed to the user. Instead, the contained messages are unwrapped automatically and send to mapped methods. If you want to receive timetags from bundles, then enable 'addTimeTagsToBundledMessages' on OscIn and grab the timetag from the last argument of your incoming bundled message.

OscTimeTag supports a precision of 0.0001 milliseconds (one DateTime tick) using the 'time' property, and a precision of about 0.0000002 milliseconds if you manipulate the 'oscNtp' property directly (not recommended unless you understand the protocol). Note that the precision of DateTime.Now is about 1 millisecond.

Troubleshooting

Messages are neither send nor received

Network
Try to ping your remote target to test the connection outside the Unity environment. In OSX, simply open the Terminal and write *ping* followed by the IP address (i.e. ping 192.168.1.39).

Firewall
Check your firewall settings. Allow Your Unity/your app incoming and outgoing connections.

Some messages are lost

UDP limitations
OSC simpl relies on UDP, a very fast but unreliable network protocol. UDP does not guarantee that messages reach their destination. If the routers involved in your network are too busy then messages may get lost.

Internet limitations
If you are sending across the internet, then keep the size of your data packets below 512 bytes to increase the chance of survival.

[OscIn] Multiple OscIn objects with same port
OSC simple allows multiple OscIn objects with same port, however only one of them will be receiving at the same time. That's the nature of sockets. Even in cases where multiple applications can access the same port, the data is handed out first-come, first-serve. Only one socket will receive.

[OscIn] Error occurred while receiving message.
If you get this warning in the console, and the next line spells "System.ArgumentException: length", then it is likely that the package size of the message was too big. In the authors tests, the size limit for packages send in broadcast mode is 1472 bytes. This limit may vary depending on the system.

Incoming bundles are not dispatched at timetag time

Timed scheduling of incoming bundled messages is not supported. All messages are dispatched immediately. This is the case for most OSC implementations and complies with a paper published in 2009 describing the forthcoming OSC 1.1.

License

OSC simpl is a Unity Asset Store product created by Danish interaction design consultancy Sixth Sensor. Please read the End User License Agreement on Unity's website.

https://unity3d.com/legal/as_terms