



Transform Crusher v3.5

© 2019 emotitron

Contact the author at: davincarten@gmail.com

I am also regularly in #Unity-Dev on Discord with the screenname **emotitron** if you have questions.

<https://discordapp.com/channels/85338836384628736/85593628650504192>

<https://discord.gg/OTYNJfCU4De7YIk8>

[Click here for current documentation](#)

[Doxygen Documentation](#)

Compress, bitcrush and serialize any transform, position, euler rotation, scale or float value. Can be instantiated and used entirely in code, or can be serialized in the inspector.

Compress & Serialize (with bitpacking):

- Complete GameObject transform
- Generic Vector3
- Position Vector3
- Rotation Vector3 with handling for [180°, 360°, 360°] or [360°, 360°, 360°]
- Quaternion compression (using smallest three)
- Scale Vector3 (Uniform and non-uniform scaling)
- Any number of floats.

NOTE: Use Accurate Center (*bool accurateCenter*) has been added, but is not yet not fully documented here yet. To use this feature, set the Use Accurate Center toggle to checked in the inspector, or by code:

```
FloatCrusher fc = new FloatCrusher(8, -10f, 10f) { AccurateCenter = true };
```

Upgrading from Free to Pro

When installing Pro over an already existing Free install, just be sure to let the DLLs overwrite the existing ones when importing from the Asset Store. Everything should be exactly the same, only now you will have the ability to set bit sizes manually in the crusher, and will no longer be restricted to the presets.

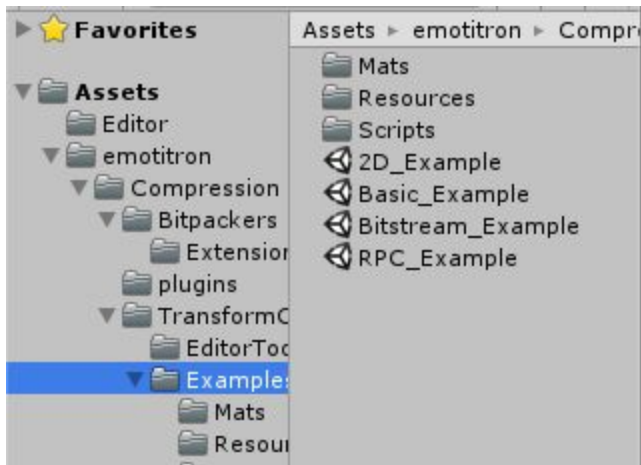
If you want to keep the using the Transform Crusher or Network Sync Transform version of that that you currently are using... **ONLY import the dlls into your project.**

While Transform Crusher and Network Sync Transform go through many radical changes with new versions, the compression DLLs should maintain backward and forward compatibility. If you have any issues at all with your upgrading, just ping me and I will be more than happy to help.

Getting Started

While the concepts of compression, serialization, and bitpacking aren't actually that complex - learning about them and how this asset makes use of them is more than most people want to absorb. Chances are you downloaded this asset to NOT have to think about any of it and have it done for you.

I strongly recommend looking at the sample projects and reading through the comments and code of the three examples.



The respective scripts in question are:

Basic_Example- A very stripped down implementation of Transform Crusher used with RPCs. UNET only.

RPC_Example - A slightly more complex scene than the Basic_Example. Syncs Rotation, Position and Scale using RPCs. This example shows how to extract primitives from a compressed transform for use with RPCs (since RPCs don't take a stream, they expect values as an argument).

Bitstream_Example - Serialized Position and Rotation of a player, and also uses the ArraySerializeExt bitpacker to include other data in the network send. This showcases how to make use of the byte[] bitstream rather than primitives.

2D_Example - An example of the crusher being used in a 2d environment where some axes are not needed.

Tinker with these samples to get familiar with how Serialization and Deserialization work with bitstreams.

Basic Usage

If you can't get through documentation, and refuse to open example scenes, here is a quick and dirty usage.

Example using a **TransformCrusher**:

```
public class Test : MonoBehaviour
{
    public TransformCrusher tCrusher = new TransformCrusher();
    public GameObject otherObj;

    public void Serialize()
    {
        // Crush a transform or rigidbody into a serializable value
        //or array of values into a CompressedMatrix
        CompressedMatrix cm = tCrusher.Compress(transform);
        CompressedMatrix rb_cm = tCrusher.Compress(GetComponent<Rigidbody>());

        // CompressedMatrix and CompressedElement
        // implicitly/explicitly convert to all kinds
        // of things that we can send over the network.
        var singleULong = (ulong)cm;
        var multipleULongs = (ulong[])cm;

        // send these ulong components through the network
        SendRPC(multipleULongs[0], (uint)multipleULongs[1]);
    }

    public void SendRPC(ulong frag0, uint frag1 = 0)
    {
        // This is just pseudocode of course, you would send this with a real RPC
        ReceiveRPC(frag0, frag1);
    }

    public void ReceiveRPC(ulong frag0, uint frag1)
    {
        // reconstruct a CompressedMatrix or Matrix from the serialized values
        CompressedMatrix cm = tCrusher.Read(frag0, frag1);
        Matrix m = tCrusher.Decompress(cm);

        // Apply the Compressed or Uncompressed value to another transform
        tCrusher.Apply(otherObj.transform, cm);
        tCrusher.Apply(otherObj.GetComponent<Rigidbody>(), m);
    }
}
```

A similar RPC usage example except using **ElementCrusher**:

```
public class Test : MonoBehaviour
{
    public ElementCrusher posCrusher = new ElementCrusher(TRSType.Position);
    public ElementCrusher rotCrusher = new ElementCrusher(TRSType.Quaternion);
    public GameObject otherObj;

    public void Serialize()
    {
        // Crush a Vector3 or Quaternion into a serializable value
        // or array of values into a CompressedMatrix
        CompressedElement ce = rotCrusher.Compress(transform.position);
        CompressedElement cq = rotCrusher.Compress(transform.rotation);

        // CompressedElement implicitly/explicitly converts to all kinds
        // of things that we can send over the network.
        var singleULong = (ulong)cq;
        var multipleULongs = (ulong[])ce;

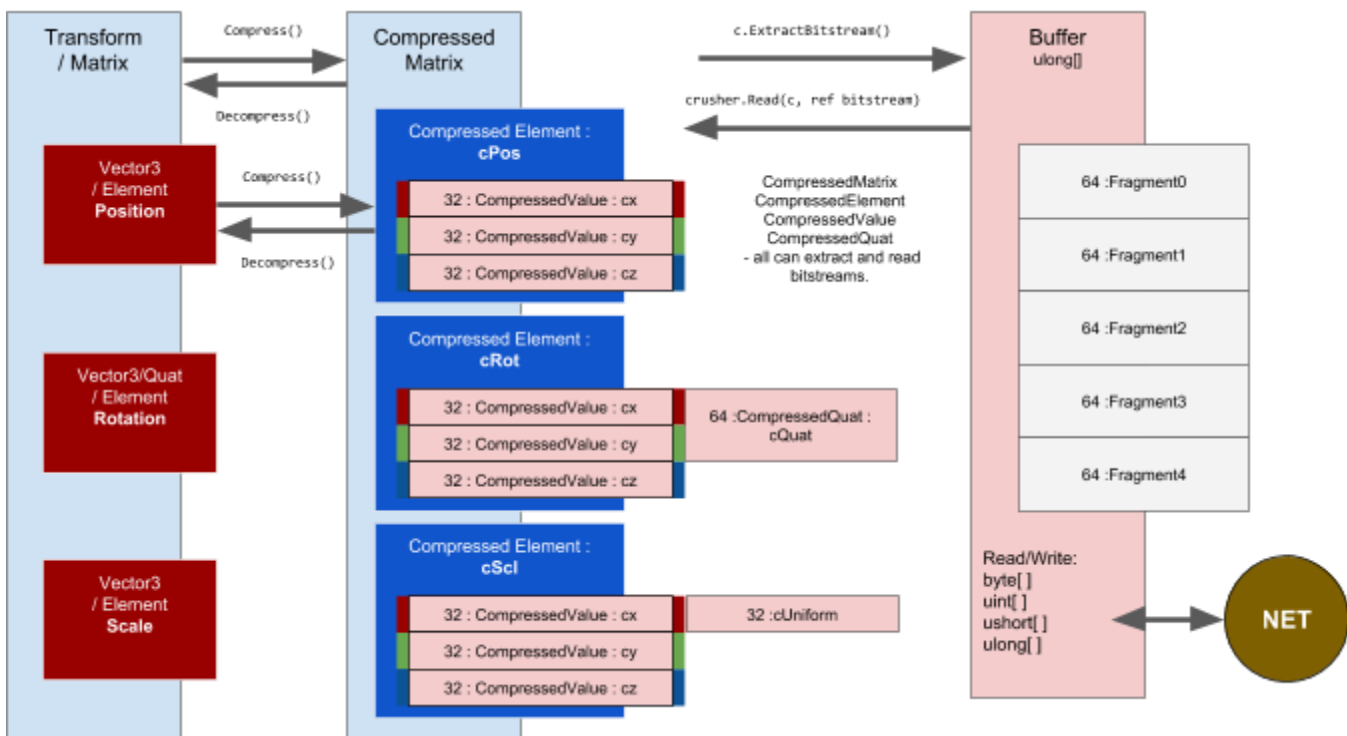
        // send these ulong components through the network
        SendRPC(multipleULongs[0], (uint)multipleULongs[1]);
        SendRPC(singleULong);
    }

    public void SendRPC(ulong frag0, uint frag1 = 0)
    {
        // This is just pseudocode of course,
        // you would send this with a real RPC
        ReceiveRPC(frag0, frag1);
    }

    public void ReceiveRPC(ulong frag0, uint frag1)
    {
        // reconstruct a CompressedElement or element from the values
        CompressedElement ce = rotCrusher.Read(frag0, frag1);
        Element e = rotCrusher.Decompress(ce);

        // Apply the Compressed or Uncompressed value to another transform
        rotCrusher.Apply(otherObj.transform, ce);
    }
}
```

Transform Crusher Process



The general flow of network compression is:

1. **Compress** the values or elements with `crusher.Compress()`
2. (Opt) **Bitpack** the compressed values into a buffer like `byte[]`, `ulong[]` or a `ulong` primitive.
3. **Network Write** from into the net library (method varies with net library and how you choose to send your data - RPC vs Writer)
4. **Network transport** `networklibrary.Send()` and `.Receive()` of some sort.
5. (Opt) **Network Read** (method varies with net library)
6. (Opt) **Unpack** the compressed values to a buffer (this is done internally in most cases)
7. **Decompress** compressed values with `Decompress()`;
8. **Apply** the values to the target transform or element.

FloatCrusher - What it does

All of the crushers (with the exception of the QuatCrusher) are built around the float crusher. The float crusher is a lossy float codec. Float compression is achieved by providing min and max values that the float can be, and then that range is subdivided by a number of steps, the number of which are determined by the bits/precision/resolution setting.

For example:

We have a gameobject that the player can move left and right from -2 units to 2 units in the world. If we compress this range down to 8 bits the resulting scale is:

	<div>-2 to 2 range</div> <div>8 bit scale (0 - 255)</div> <div>Precision = (4 units / 256) = ~0.0156</div> <div>Resolution = (255 / 4 units) = ~1/64 of a unit</div>				
Compressed	0	63	127/128	192	255
Uncompressed	Far Left x = -2	50% Left x = -1	Center x +/- ~.007	50% Right x = 1	Far Right x =2

Note: Actual 0 in this situation is impossible. 127 and 128 values will be ~.01 left or right of center. To get actual zero as a center, set AccurateCenter = true. This will drop reduce the range by 1, resulting in an odd number of subdivisions, allowing for one of the values to act as a true center or zero.

With AccurateCenter == true

	<div>-2 to 2 range</div> <div>8 bit scale (0 - 255)</div> <div>Precision = (4 units / 255) = ~0.0156</div> <div>Resolution = (254 / 4 units) = ~1/64 of a unit</div>				
Compressed	0	63	127	192	254
Uncompressed	Far Left x = -2	~50% Left x = ~-1	Center x = 0	~50% Right x = ~1	Far Right x =2

We can now describe the x position of the player with 8 bits, rather than 32 bits. Here is a very basic usage of the example above.

```
FloatCrusher fr = new FloatCrusher(8, -10f, 10f);
uint compressedfloat = fr.Compress(transform.position.x);
float restoredfloat = fr.Decompress(compressedfloat);
```

This example doesn't actually do anything useful. In actual usage the compressedfloat would be synced over the network. In this case we would sync it as a Byte rather than a Float, at 1/4 its original size.

Crusher Types

There are a few variations of crushers included.
The base crushers the others build from are the FloatCrusher and the QuatCrusher.

ElementCrusher and TransformCrusher are collections of these crushers with special handlers applied for the unique qualities of Position, Rotation and Scale.

Crushers are serializable classes and are simply added to a gameobject like so:

```
using emotitron.Compression;

public class MyPlayer : MonoBehaviour
{
    // A single float compressor
    public FloatCrusher fcrusher;

    // Element Compressor, with handlers for Position, Eulers, Quaternions, Scale
    public ElementCrusher ecrusher = new ElementCrusher (TRSType.Position);

    // Quaternion compression
    public QuatCrusher qcrusher;

    // Transform compressor combining Position Scale & Rotation)
    public TransformCrusher tcrusher;
}
```

If the class is a MonoBehaviour, the crusher settings will appear in the editor.



These compressor instances have Compress, Decompress, Read and Write as well as many other methods available for compressing, bitpacking and serializing float and transform data.

Usage : (Simple) Compress to Primitive

The easiest way to use any of the crushers is to compress to and from unsigned primitives

- byte (UInt8)
- ushort (UInt16)
- uint (UInt32)
- ulong (UInt64)

We can bitpack down to exact bit counts, but for this usage we will be rounding off to nearest primitive (8, 16, 32 & 64).

Here is a very simple example of crushing a rotation to a uint (suitable for syncvars and RPCs) and back to a Vector3. In this example we cast the results of compression to uint, but it can be cast to ulong, ushort and byte as well - IF the compression settings result in a size that can be contained in that.

You can see the bits used for a crusher's settings in the top left corner.



Example 1 : Rotation Crusher

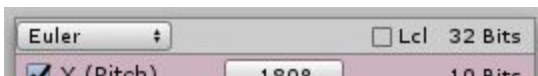
This example creates a Rotation Crusher, then uses it to compress and restore a Vector3.

```
using emotitron.Compression;

public class Test : MonoBehaviour
{
    // Create the crusher object. It will appear in the component.
    public ElementCrusher rCrusher = new ElementCrusher(TRSType.Euler);

    public void Start()
    {
        // Rotate this gameobject to some random rotation
        transform.eulerAngles = new Vector3(-33, 33, 66);

        // Crush the rotation to a uint
        // (the default TRSType.Euler setting adds up to 32 bits)
        // CompressedElement explicitly converts to serializable primitives
        uint crushed = (uint)rCrusher.Compress(transform);
        // Restore the vector from the crushed value and see that it worked
        Vector3 restoredRotation = rCrusher.Decompress(crushed);
        Debug.Log("Uncrushed = " + restoredRotation);
    }
}
```



Notice the bit total in the top right corner of the crusher. In this case our total bits is 32, so this crushed rotation will fit in a uint.

Example 2 : Transform Crusher

This example creates a Transform Crusher, then uses it to compress the transform of the object the MonoBehaviour is attached to, and then decompress and apply it to another game object.

```
public class Test : MonoBehaviour
{
    public TransformCrusher tCrusher = new TransformCrusher();
    public GameObject otherObj;

    public void Start()
    {
        // Transform this gameobject to some random TRS
        transform.position = new Vector3(-5f, 2f, 3f);
        transform.eulerAngles = new Vector3(20, 30f, 0);
        transform.localScale = new Vector3(.5f, .5f, .5f);

        // Crush the transform to a ulong (TransformCrushers default
        // settings add up to 64 bits, so it will fit.
        // Bitstream implicitly casts to ulong. For settings greater
        // than 64bits we have to start storing Bitstream instead of ulong
        // to hold all of the fragments.
        ulong crushed = (ulong)tCrusher.Compress(transform);

        // Apply the crushed value to another transform
        tCrusher.Apply(otherObj.transform, crushed);

        // Check the results
        Debug.Log("OtherObj = " +
            " pos: " + otherObj.transform.position +
            " rot: " + otherObj.transform.eulerAngles +
            " scl: " + otherObj.transform.localScale
        );
    }
}
```



Notice the bit total in the top right corner of the crusher. In this case our total bits is 64, so this crushed transform will fit in a ulong.

Usage : Compress to Bitstream **DEPRECATED**

~~TransformCrusher has a 40 byte struct-based bitstream built-in (backed internally by an unsafe Fixed ulong[5]) that can be used to chain any combination of compressed elements into a packed bitstream. This bitstream can then be written to any serializer. Check the **Bitstream_Example** scene in the Examples folder to see an example of the bitstream being used to pack data for serialization with a network writer..~~

```
public class Test : MonoBehaviour
{
    // Create a generic float crusher
    public FloatCrusher fc = new FloatCrusher(BitPresets.Bits12, -10f, 10f);

    byte[] simulatednetwork = new byte[20];
    int bytecount;

    public void Start()
    {
        // Create a bitstream struct
        // REMEMBER this is a struct and not a class - it HAS to be passed by ref
        Bitstream outstream = new Bitstream();


        // Crush some random values between -10 and 10
        // and pack them into bitstream. Each write is 10 bits.
        fc.Write(1.111f, ref outstream);
        fc.Write(-2.222f, ref outstream);
        fc.Write(-9.999f, ref outstream);
        // Alternate way of writing to the bitstream
        outstream.Write(fc.Compress(3.333f));
        outstream.Write(fc.Compress(-4.444f));
        outstream.Write(fc.Compress(-8.888f));


        Debug.Log("Bytes Used = " + outstream.BytesUsed +
            " (" + outstream.WritePtr + ")");

        // You can write bytes out of this to your network engine
        for (int i = 0; i < outstream.BytesUsed; ++i)
            simulatednetwork[i] = outstream.ReadByte();

        // simulatednetwork represents the byte[] array of a network engine
        // Create a bitstream from the incoming byte[]
        Bitstream instream = new Bitstream(simulatednetwork);

        // The crushers read and decompress the floats from the stream
        Debug.Log("Unpacked and restored floats: " +
            fc.ReadAndDecompress(ref instream) + " " +
            fc.ReadAndDecompress(ref instream) + " " +
            fc.ReadAndDecompress(ref instream) + " " +
            fc.ReadAndDecompress(ref instream) + " " +
            fc.ReadAndDecompress(ref instream) + " " +
            fc.ReadAndDecompress(ref instream)
        );
    }
}
```

 Bytes Used = 9 (72)
UnityEngine.Debug:Log(Object)

 Unpacked and restored floats: 1.111112 -2.21978 -10 3.333334 -4.442002 -8.886447
UnityEngine.Debug:Log(Object)

~~We used 72 bits for this test, which turns out to be an even 9 bytes. If we write an odd number of bits, BytesUsed will round up the next whole byte.~~

Usage : Writing Bits to Byte[] Arrays

The crushers are capable of packing bits directly to any supplied Byte[]. The bitposition is passed as a reference, so that the crusher can increment the arrays bit pointer.

```
public class Test : MonoBehaviour
{
    // Create a generic float crusher
    public FloatCrusher fc = new FloatCrusher(BitPresets.Bits10, -10f, 10f);

    // This is our byte buffer that we are using as a bitstream.
    readonly byte[] buffer = new byte[9];
    int bitposition;

    public void Start()
    {
        // Crush some random values between -10 and 10
        // and pack them into the byte[]. Each write is 10 bits.
        bitposition = 0;
        fc.Write(.003f, buffer, ref bitposition);
        fc.Write(1.111f, buffer, ref bitposition);
        fc.Write(-2.222f, buffer, ref bitposition);
        fc.Write(5.555f, buffer, ref bitposition);
        fc.Write(-9.999f, buffer, ref bitposition);

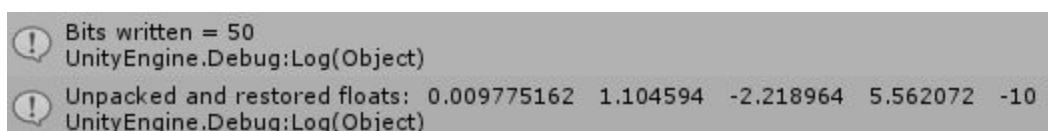
        Debug.Log("Bits written = " + bitposition);

        // Reset our bitpointer to 0 to start our read.
        bitposition = 0;

        Debug.Log("Unpacked and restored floats: " +
            fc.ReadAndDecompress(buffer, ref bitposition) + " " +
            fc.ReadAndDecompress(buffer, ref bitposition) + " " +
            fc.ReadAndDecompress(buffer, ref bitposition) + " " +
            fc.ReadAndDecompress(buffer, ref bitposition) + " " +
            fc.ReadAndDecompress(buffer, ref bitposition)
        );
    }
}
```



Notice the bit total in the top right corner of the crusher. In this case bits are 10, so each crushed float is reduced down to a 10 bit value.



The numbers come back close, but not exact. This is because this compression is lossy.

ElementCrusher Editor Parameters

TRSType
Transform element type this compressor is for. (Position / Rotation / Scale / Generic)

If localPosition/
localRotation/
localScale for
Apply()

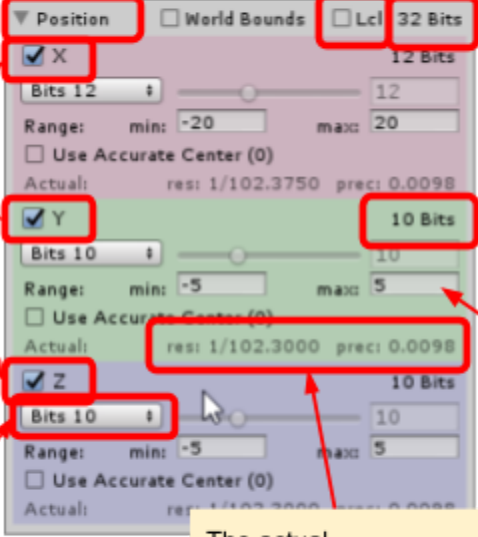
Total bits of the compressed
x, y and z axis.

enabled
If `showEnabledToggle = true`
this toggle will be available.
Disabling indicates that this
axis should not be
compressed/serialized and
should be skipped.

BitsDeterminedBy
There are three ways to adjust
the bitsize/accuracy trade-off:
Bits: Directly set the number
of bits that this axis should be
compressed to.
Resolution: Desired minimum
number of unit subdivisions of
resolution.
Precision: Desired minimum
increment size in units.

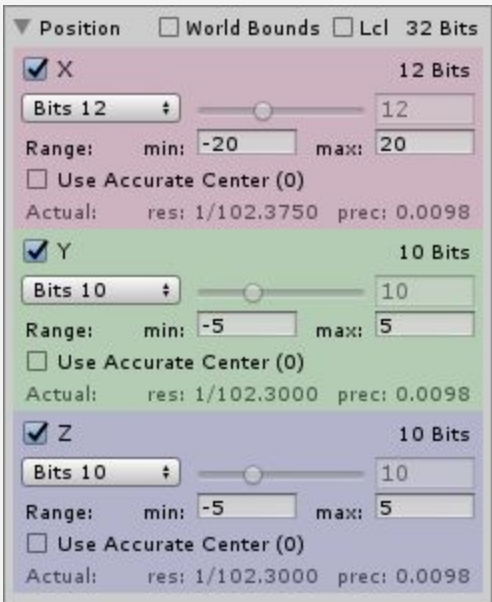
Min and Max
Float values outside
of this range will be
clamped. The
closer these values
are to one another
the better the
compression.

The actual
resolution/precision
that will result from
the current settings.
(May be better than
requested values.)



The screenshot shows the 'Position' section of the ElementCrusher Editor. It features three axes: X, Y, and Z. Each axis has a 'Bits' slider, a 'Range' (min/max), and an 'Actual' resolution/precision. The X-axis is set to 12 bits, Y to 10 bits, and Z to 10 bits. The 'Lcl' checkbox is checked, and the 'World Bounds' checkbox is unchecked. The 'Use Accurate Center (0)' checkbox is unchecked for all axes. The 'Actual' resolution/precision for X is 1/102.3750 and 0.0098, for Y is 1/102.3000 and 0.0098, and for Z is 1/102.3000 and 0.0098. Red boxes highlight the 'Position' dropdown, the 'Lcl' checkbox, the '32 Bits' label, the 'X', 'Y', and 'Z' checkboxes, the 'Bits' sliders, the 'Range' min/max fields, the 'Actual' resolution/precision fields, and the 'Use Accurate Center (0)' checkboxes. Red arrows point from the surrounding text boxes to these highlighted elements.

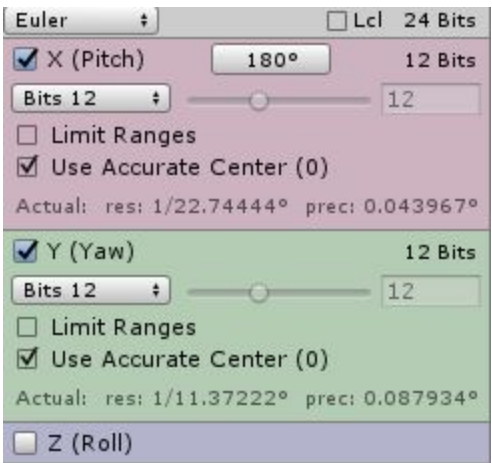
ElementCrusher Type Differences



TRSType = Position / Generic

Position and Generic have the standard settings.

Position gains a Use World Bounds toggle at the top. Selecting this will use the global position settings that are determined by WorldBounds components in the scene.

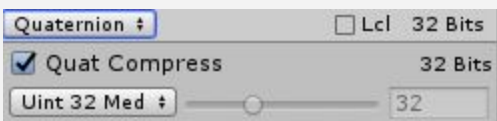


TRSType = Euler

Euler Rotation is different from Position/Scale/Generic in several ways.

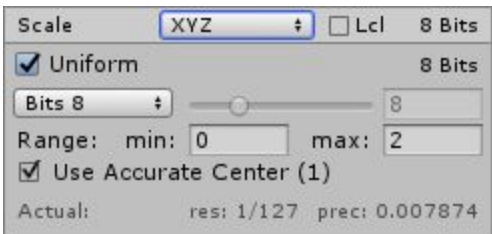
Each rotation axis has a max range of 360 degrees. Disabling Limit Ranges will default to the full 360° being used.

The X axis may be restricted to 180°. When set to 180° any X values outside of -90° and 90° are mirrored, and the Y and Z rotations are flipped 180°. For example, an compressed vector3 of (92, 0, 0) will be turned into (88, 180, 180) which is the same orientation - while keeping the X within the gimbal range of -90 to 90. This setting reduces the size of the x axis by 1 bit.



TRSType = Quaternion

Rather than compressing rotation into 3 floats as with Euler, Quaternion compresses the entire Quaternion using a smallest three compression, and compresses the quaternion into a single value ranging from 16 to 64 bits. The ideal range is between 24 and 48. I have some handy charts [here](#) if you are interested in the stats.

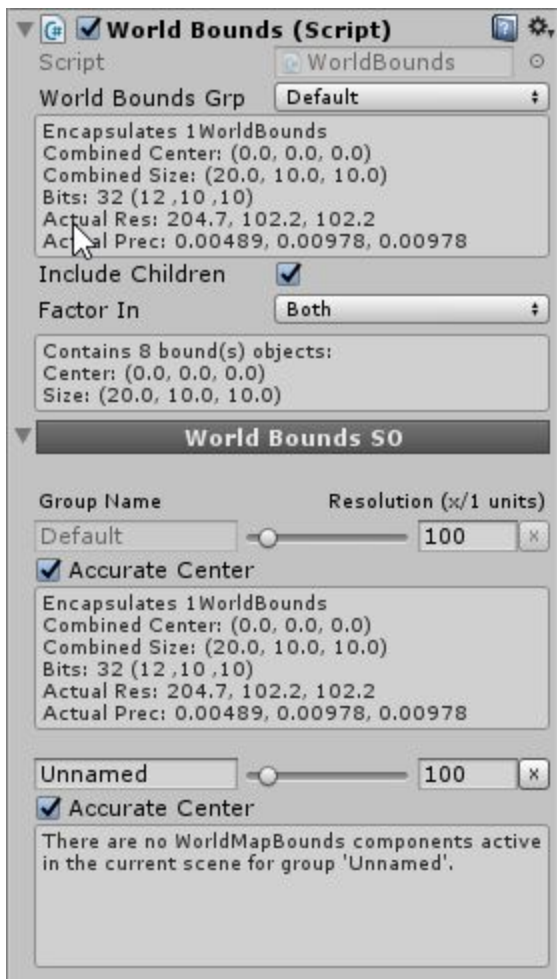


TRSType = Scale

Scale is unique in that it supports uniform scaling - which uses one axis to describe multiple axes. The Enum Popup at the top of the drawer lets you select Non-Uniform for normal independent axis compressors, or any combination of the 3 axes, which will all be scaled the same amount.

WorldBounds

WorldBounds components can be placed and static objects in a scene to define the Bounds of the networked world. The bounds of the Mesh and/or Colliders of the GameObject (and children optionally) are factored into the selected World Bounds Group.



Position ElementCrusher's have a toggle that allows you to use the world bounds crushers. Selecting it will hide the settings for the crusher and replace it with a selector for which World Bounds Group to use, and a summary of the current values.



You can also access the WorldBoundGroups and their ElementCrusher in code with:

```
WorldBoundsSO.Single.worldBoundsGroups[boundsGroup].crusher
```

More convenience methods will will eventually be written for this, but I first need to consider the startup order before making a tangled mess.

Bitpacking/Serializers

The internal bitpacking tools used by this Asset have been shared with an MIT license and can be found along with documentation at:

<https://github.com/emotitron/BitpackingTools>

First let's define Bitpacking and Serialization.

Serialization

Serialization/Deserialization is the act of writing data in a linear fashion. The key to making it work is that you must Read in the exact number of bits and exact as what you wrote. For example if you serialize three bytes to a buffer (using the built in PrimitiveSerializerExt in this example)... you need to read them out exactly the same. Internally all of the crushers automatically do this for you.

```
uint buffer = 0; // 32 bit buffer
int bitposition = 0;

byte a = 11, b = 22, c = 33; // bytes are 8 bits

a.Inject(ref buffer, ref bitposition);
b.Inject(ref buffer, ref bitposition);
c.Inject(ref buffer, ref bitposition);

// buffer now contains
// 00000000 - CCCCCCCC - BBBB BBBB - AAAAAAAA

bitposition = 0;

Debug.Log(
    "a = " + buffer.Read(ref bitposition, 8) +
    "b = " + buffer.Read(ref bitposition, 8) +
    "c = " + buffer.Read(ref bitposition, 8));
```

Bitpacking/Serializing options in Transform Crusher

- **Bitstream** **DEPRECATED** - A 40byte fixed buffer struct that uses unsafe to pack, unpack, serialize and deserialize data. The Bitstream struct is actually 48 bytes in size (40 for the data and 8 for the read/write positions). This is a bit hefty for struct, so there are ref options for most functions surrounding the bitstream class when performance gets more critical. The function of Bitstream is to provide a fast reading/writing go between for creating blocks of serialized elements, with methods for getting that data in and out of network library Send/Rev methods.
- **ArraySerializer** - This actually isn't a class or a struct, but rather is just an extension class for primitive arrays (byte[], uint[], ulong[]), You can bitpack values into an array of your own. You just need an array and an int field to hold the bitposition value that represents the current position in the buffer of the writer or reader.
- **PrimitiveSerializer** - Like the ArraySerializer, this is just an extension set that lets you treat any primitive as as buffer. All you need is a primitive (such as a ulong) and a companion int to keep track of the bitposition.

Bitstream **DEPRECATED**

The Bitstream struct is a bitpacker and serializer. It is not always needed in the process of crushing floats or transform elements for the network. The reasons to make use of the Bitstream are:

- You are packing multiple items.
- A compressed items size exceeds 64bits. The largest primitive you can compress to is a ulong/Uint64. Anything larger and you need to fragment the data into multiple primitives.

SharedCrushers **EXPERIMENTAL**

SharedTransformCrusher / SharedElementCrusher / SharedFloatCrusher

I make use of a wrapper for crushers for my own assets and projects, that allow a crusher to be declared in multiple places and appear in the inspector, but all of those instances are tied together to one instance that is synchronized using a ScriptableObject.

Having the crusher on a component of an object that gets instantiated creates new crusher instances with it, which in small numbers is tiny and not worth thinking about, but these redundant copies may start to become a concern if you have 100s of networked objects. SharedCrushers are one answer to prevent that waste of memory.

There are three synchronization methods:

- **ShareByCommon.ComponentAndFieldName**
All instances of this component will use the same crusher instance for this field.
- **ShareByCommon.FieldName**
All instances of ANY component will use the same crusher for this field name.
- **ShareByCommon.Prefab**
All instances of a Prefab will use the same crusher for this field on this Component. This is highly dependent on OnGUI code in the inspector, and will only work after the component has constructed and has been inspected (properties have been shown in the editor).

Basic Usage:

```
public SharedTransformCrusher sc =  
    new SharedTransformCrusher(ShareByCommon.ComponentAndFieldName);
```