Transform Crusher

Click here for current 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.

FloatCrusher - What it does

The float crusher is a lossy float codec. Float compression is achieved by providing a min and max values that the float can be, and then that range is subdivided by a number of steps, which is 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:

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

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 ¼ its original size.

Crusher Types

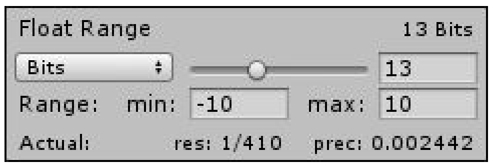
There are a few variations of crushers included.
The base crushers the others build from are the FloatCrusher and the QuatCrusher.

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.


Float Range 13 Bits
Bits 13
Range: min: -10 max: 10
Actual: res: 1/410 prec: 0.002442

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)

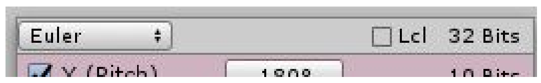
We can bitpack down to exact bit counts, but for this usage we will be rounding off to nearest primitive (8, 16, 32 & 64).

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.

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)
		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);
	}
}
```

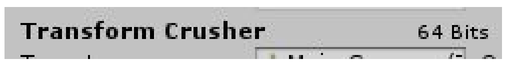

Euler ☐ Lcl 32 Bits

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 monobehavior 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 uint
		//(the default TRSType.Euler setting adds up to 32 bits)
		ulong crushed = 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
		);
	}
}
```

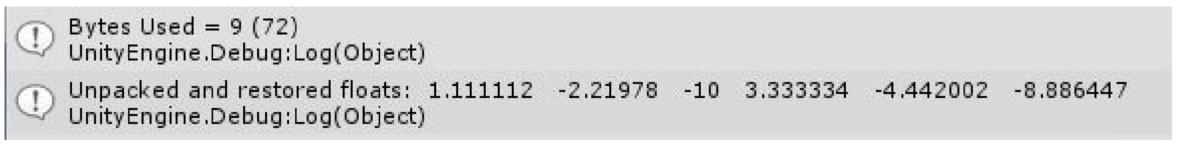

Transform Crusher 64 Bits

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

TransformCrusher has a struct-based bitstream built-in that can be used to chain any combination of compressed (or even other fields if so desired) 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)
            );
    }
}
```



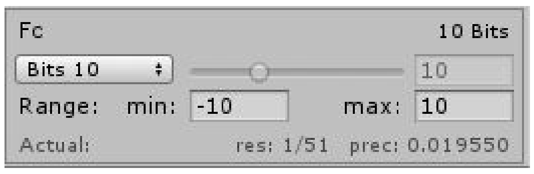
```
Bytes Used = 9 (72)
UnityEngine.Debug:Log(Object)
Unpacked and restored floats:  1.111112   -2.21978   -10   3.333334   -4.442002   -8.886447
UnityEngine.Debug:Log(Object)
```

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.

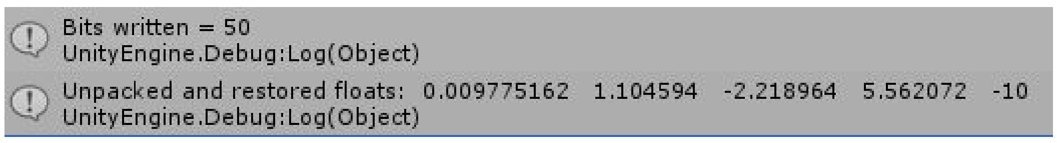
Usage : Writing Bits to Byte[] Arrays

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.
	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)
			);
	}
}
```

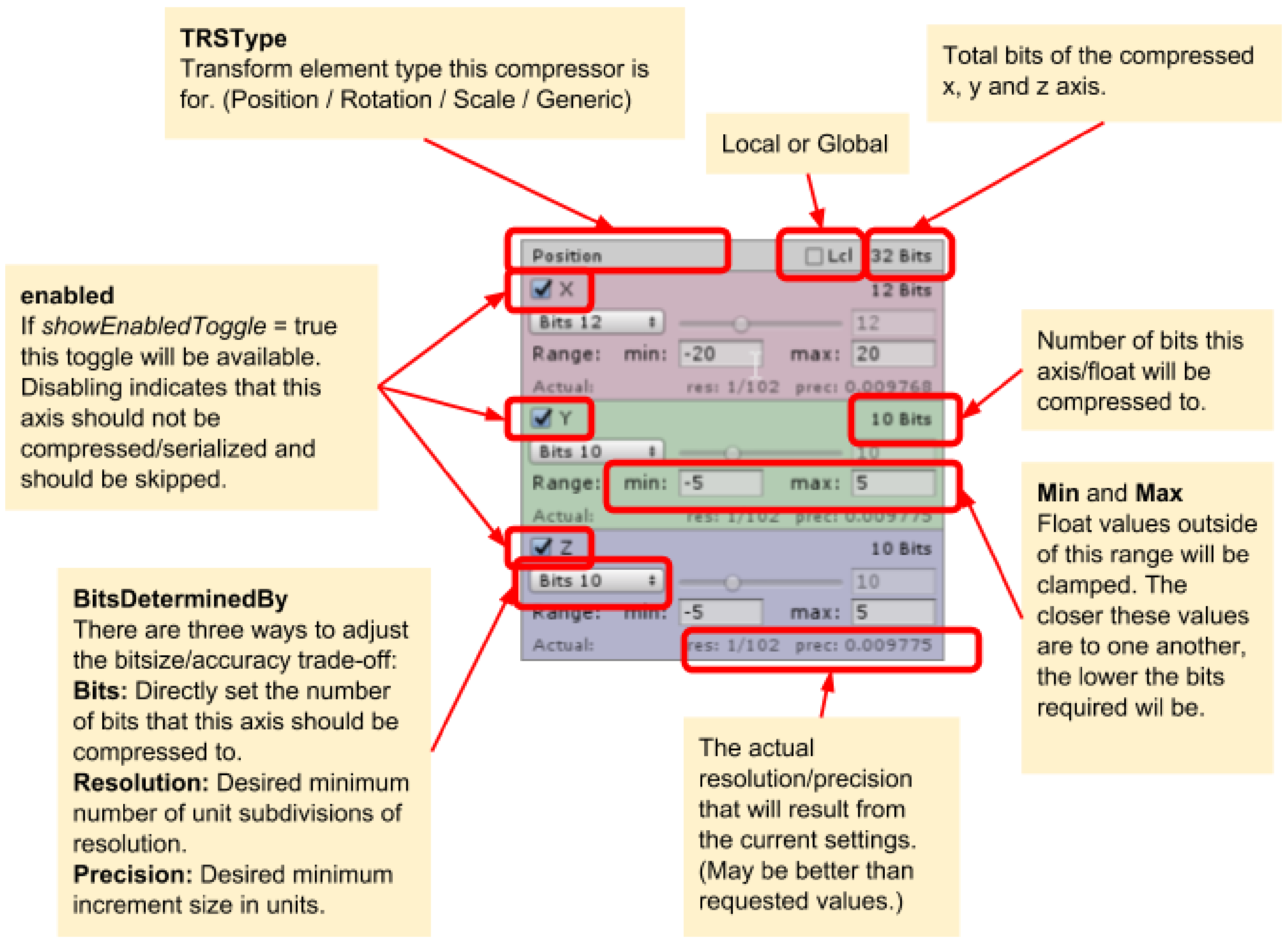

Fc 10 Bits
Bits 10 — 10
Range: min: -10 max: 10
Actual: res: 1/51 prec: 0.019550

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.


```
Bits written = 50
UnityEngine.Debug:Log(Object)
Unpacked and restored floats:  0.009775162   1.104594   -2.218964   5.562072   -10
UnityEngine.Debug:Log(Object)
```

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)

Local or Global

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.

Number of bits this axis/float will be compressed to.

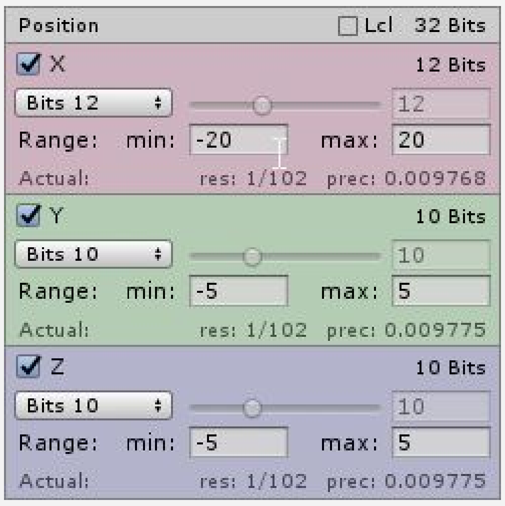
Min and **Max**
Float values outside of this range will be clamped. The closer these values are to one another, the lower the bits required wil be.

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.

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

```
Position                              Lcl  32 Bits
X                                          12 Bits
Bits 12                                    12
Range:   min: -20           max: 20
Actual:            res: 1/102   prec: 0.009768
Y                                          10 Bits
Bits 10                                    10
Range:   min: -5            max: 5
Actual:            res: 1/102   prec: 0.009775
Z                                          10 Bits
Bits 10                                    10
Range:   min: -5            max: 5
Actual:            res: 1/102   prec: 0.009775
```

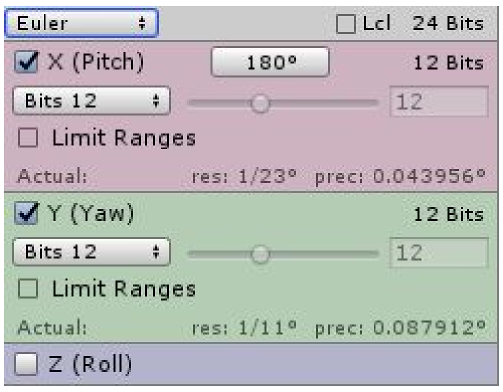
Vector3Crusher Element Type Differences



```
Position                      ☐ Lcl  32 Bits
☑ X                                  12 Bits
Bits 12                                 12
Range:   min: -20          max: 20
Actual:          res: 1/102   prec: 0.009768
☑ Y                                  10 Bits
Bits 10                                 10
Range:   min: -5           max: 5
Actual:          res: 1/102   prec: 0.009775
☑ Z                                  10 Bits
Bits 10                                 10
Range:   min: -5           max: 5
Actual:          res: 1/102   prec: 0.009775
```

TRSType = Position / Generic

Position and Generic have the standard settings.



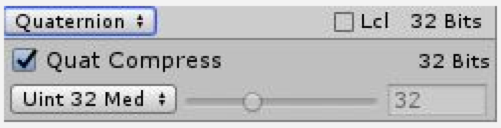
```
Euler                         ☐ Lcl  24 Bits
☑ X (Pitch)        180°              12 Bits
Bits 12                                 12
☐ Limit Ranges
Actual:          res: 1/23°   prec: 0.043956°
☑ Y (Yaw)                            12 Bits
Bits 12                                 12
☐ Limit Ranges
Actual:          res: 1/11°   prec: 0.087912°
☐ Z (Roll)
```

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.

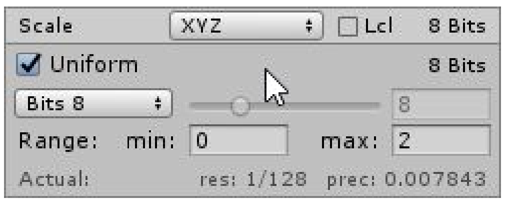
The X axis may be restricted to 180°. When set to 180° any X values outside of -90° and 90° are mirrored, and the Y and Z rotations are flipped 180°. For example, an compressed vector3 of (92, 0, 0) will be turned into (88, 180, 180) which is the same orientation - while keeping the X within the gimbal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



```
Quaternion                    ☐ Lcl  32 Bits
☑ Quat Compress                      32 Bits
Uint 32 Med                             32
```

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.



```
Scale          XYZ            ☐ Lcl   8 Bits
☑ Uniform                             8 Bits
Bits 8                                  8
Range:   min: 0            max: 2
Actual:          res: 1/128   prec: 0.007843
```

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.