HIDDEN TEXTURE DATA

V1.0 API REFERENCE & FAQ

API REFERENCE

This is a fairly heavy document, so before reading this API Reference I suggest you watch an overview & short demonstration of this asset here:

Unity: How to hide data in an image (Spore, Pico-8, etc.)

And if you are looking for a more visual API reference, this video has everything you need:

Unity: How to hide data in an image (Asset reference)

Personally I learn best from example. There is a demo scene included with buttons for all the available methods. I recommend playing around in that scene and taking a look at what methods each button calls. This will give you a good understanding of how the asset works.

If you have not yet purchased the asset and are reading this reference from the Asset Store you can try out the demo scene here: http://stijndelaruelle.com/project/hiddentexturedata/

Setup

No special setup is required apart from importing unless you want to build the example scene as a standalone executable on Windows.

The “StandaloneFileBrowser” asset used in the demo scene requires you to build with the API Compatibility Level .NET 4.x (and maybe even Mono as the scripting backend)

However I would only recommend this if you want to specifically build the demo. When building your own game, just remove all the demo files. They are not needed for this asset to work!

Using the asset

Using the asset can be divided in 2 sections

- Writing hidden data to a texture
- Reading hidden data from a texture

Both sections use the same class: HiddenTextureData, so let’s start there.

Creating an HiddenTextureData object

To write & read hidden data we need to create a HiddenTextureData object. As the HiddenTextureData class doesn't derive from MonoBehaviour it has to be created within another class instead of putting it on an object in your scene.

To create the HiddenTextureData object you can call its constructor:

```
HiddenTextureData hiddenTextureDataObject = new HiddenTextureData();
```

Once you have a HiddenTextureData object, you can use it to write & read hidden data from a texture.

Setting a texture

Before writing or reading data, we need to assign a texture to the HiddenTextureData object. This Texture2D can come from anywhere f.e.:

- Created in code
- From a RenderTexture
- Loaded from disc (via File.ReadAllBytes)
- Downloaded (via UnityWebRequest)

Assigning a texture can be done with the following method:

```
hiddenTextureDataObject.SetTexture(texture2D);
```

There are also 2 utility methods to make the creation in code & loading from disc options a little easier:

```
//Create and assign a white texture
hiddenTextureDataObject.CreateWhiteTexture(width, height);

//Load and assign a texture from disk
hiddenTextureDataObject.LoadTexture(imageFilePath);
```

In the overview video at 5:57 I explain how you can load a texture in the demo scene:
Unity: How to hide data in an image (Spore, Pico-8, etc.)

Writing hidden data

Adding regular data

Once a texture is loaded you can write data to it using a variety of methods.
Each literal type has its own method as listed below.

In the reference video at 0:40 I explain how you can write data in the demo scene:
Unity: How to hide data in an image (Asset reference)

```
//8 bit
hiddenTextureDataObject.WriteByte(byte);            //Write a byte
hiddenTextureDataObject.WriteSignedByte(sbyte);     //Write a sbyte
hiddenTextureDataObject.WriteBool(bool);            //Write a bool

//8-16 bit (8 bit when ASCII, 16 bit when Unicode)
hiddenTextureDataObject.WriteChar(char);            //Write a char

//16 bit
hiddenTextureDataObject.WriteShort(short);          //Write a short
hiddenTextureDataObject.WriteUnsignedShort(ushort); //Write a ushort

//32 bit
hiddenTextureDataObject.WriteInt(int);              //Write an int
hiddenTextureDataObject.WriteUnsignedInt(uint);     //Write an uint
hiddenTextureDataObject.WriteFloat(float);          //Write a float

//64 bit
hiddenTextureDataObject.WriteLong(long);            //Write a long
hiddenTextureDataObject.WriteUnsignedLong(ulong);   //Write an ulong
hiddenTextureDataObject.WriteDouble(double);        //Write a double

//128 bit
hiddenTextureDataObject.WriteDecimal(decimal);      //Write a decimal
```

Adding arrays

Writing arrays of literals works the exact same way, with 1 small exception for booleans:

As booleans can only be true or false, it's a waste to spend an entire byte on one of them. So when writing multiple booleans, they will be packed to only use 1 byte per 8 booleans.

A string is in a sense also an array of characters, so it's also included here.

When an array is variable in size it is recommended to write its length before writing the array itself, as reading an array requires its length to be known.

In the reference video at 4:22 I explain how you can add arrays in the demo scene:
Unity: How to hide data in an image (Asset reference)

```
//8 bit per index
hiddenTextureDataObject.WriteByteArray(byte[]);        //Write byte[]
hiddenTextureDataObject.WriteSignedByteArray(sbyte[]); //Write sbyte[]

//Write bool[] (1 byte per 8 booleans)
hiddenTextureDataObject.WriteBoolArray(bool[]);

//8 16 bit per index
hiddenTextureDataObject.WriteCharArray(char[]);          //Write char[]

//16 bit per index
hiddenTextureDataObject.WriteShortArray(short[]);         //Write
short[]
hiddenTextureDataObject.WriteUnsignedShortArray(ushort[]); //Wr.
ushort[]

//32 bit per index
hiddenTextureDataObject.WriteIntArray(int[]);            //Write int[]
hiddenTextureDataObject.WriteUnsignedIntArray(uint[]);   //Write uint[]
hiddenTextureDataObject.WriteFloatArray(float[]);        //Write float[]

//64 bit per index
hiddenTextureDataObject.WriteLongArray(long[]);          //Write long[]
hiddenTextureDataObject.WriteUnsignedLongArray(ulong[]); //Write ulong[]
hiddenTextureDataObject.WriteDoubleArray(double[]);      //Write
double[]

//128 bit per index
hiddenTextureDataObject.WriteDecimalArray(decimal[]); //Write decimal[]
```

```
//Variable bits per index
hiddenTextureDataObject.WriteString(string);        //Write string
hiddenTextureDataObject.WriteStringArray(string[]); //Write string[]
```

Writing to the texture

Once data is added, it can be written to the texture by calling the following method.

```
hiddenTextureDataObject.WriteDataToTexture();
```

In the overview video at 6:55 I explain how you can write the data in the demo scene:
Unity: How to hide data in an image (Spore, Pico-8, etc.)

Saving the texture

Once data is written to the texture, one option is to write it to disk so players can share the image however they want. This can be done with the following method.

```
hiddenTextureDataObject.SaveTexture(imageFilePath);
```

In the overview video at 8:48 I explain how you can save the texture in the demo scene:
Unity: How to hide data in an image (Spore, Pico-8, etc.)

Basic write example

Putting everything together in its most basic form, it should look something like this:

```
//Construct the HiddenTextureData object
HiddenTextureData hiddenTextureDataObject = new HiddenTextureData();

//Set a texture (05:57 in the overview video)
hiddenTextureDataObject.SetTexture(new Texture2D(10, 10));

//Add some data (00:40 in the reference video)
//All value C# types have functions, including array variants (04:22 in
the reference video)
hiddenTextureDataObject.WriteByte(42);

//Write the data to the texture (06:55 in the overview video)
hiddenTextureDataObject.WriteDataToTexture();

//Save the texture to disk (8:48 in the overview video)
hiddenTextureDataObject.SaveTexture("hiddendata_image.png");
```

Utility methods

There are also a couple of writing utility methods that you most likely won't need but they are here just in case.

To clear all the data that is ready to be written use:

```
hiddenTextureDataObject.ClearWriteHiddenData();
```

To get the number of bytes that are currently going to be written use:

```
long byteCount = hiddenTextureDataObject.GetWriteByteCount();
```

Reading hidden data

Reading hidden data works very similar to writing. After creating a HiddenTextureData object & assigning a texture we can start reading data from the texture.

The first method that should be called is the following:

```
hiddenTextureDataObject.ReadDataFromTexture();
```

Afterwards you can start reading regular data & arrays in a similar fashion as writing them:

In the overview video at 7:18 I explain how you can read the data in the demo scene:
Unity: How to hide data in an image (Spore, Pico-8, etc.)

Reading regular data

```
//8 bit
byte  b  = hiddenTextureDataObject.ReadByte();          //Read a byte
sbyte sb = hiddenTextureDataObject.ReadSignedByte();    //Read a sbyte
bool  bl = hiddenTextureDataObject.ReadBool();          //Read a bool

//8-16 bit (8 bit when ASCII, 16 bit when Unicode)
char c = hiddenTextureDataObject.ReadChar();            //Read a char

//16 bit
short  s  = hiddenTextureDataObject.ReadShort();        //Read a short
ushort us = hiddenTextureDataObject.ReadUnsignedShort(); //Read a ushort
```

```
//32 bit
int   i  = hiddenTextureDataObject.ReadInt();            //Read an int
uint  ui = hiddenTextureDataObject.ReadUnsignedInt();    //Read an uint
float f  = hiddenTextureDataObject.ReadFloat();          //Read a float

//64 bit
long   l  = hiddenTextureDataObject.ReadLong();          //Read a long
ulong  ul = hiddenTextureDataObject.ReadUnsignedLong();  //Read an ulong
double d  = hiddenTextureDataObject.ReadDouble();        //Read a double

//128 bit
decimal dec = hiddenTextureDataObject.ReadDecimal();     //Read a
decimal
```

Reading arrays

When reading an array you always have to pass the length, as otherwise it doesn't know when to stop reading. (string is an exception, as a string prepends it's length automatically)

If the array is variable in size it is recommended to prepend its length when writing. That way you can read the length before reading the array itself.

```
//8 bit per index
byte[]  byteArr = hiddenTextureDataObject.ReadByteArray();
sbyte[] sbyteArr = hiddenTextureDataObject.ReadSignedByteArray();

//Read bool[] (1 byte per 8 booleans)
bool[] boolArr = hiddenTextureDataObject.ReadBoolArray();

//8 16 bit per index
char[] charArr = hiddenTextureDataObject.ReadCharArray();

//16 bit per index
short[]  shortArr  = hiddenTextureDataObject.ReadShortArray();
ushort[] ushortArr = hiddenTextureDataObject.ReadUnsignedShortArray();

//32 bit per index
int[]   intArr   = hiddenTextureDataObject.ReadIntArray();
uint[]  uintArr  = hiddenTextureDataObject.ReadUnsignedIntArray();
float[] floatArr = hiddenTextureDataObject.ReadFloatArray();

//64 bit per index
long[]   longArr   = hiddenTextureDataObject.ReadLongArray();
ulong[]  ulongArr  = hiddenTextureDataObject.ReadUnsignedLongArray();
```

```
double[] doubleArr = hiddenTextureDataObject.ReadDoubleArray();
//128 bit per index
decimal[] decimalArr = hiddenTextureDataObject.ReadDecimalArray();

//Variable bits per index
string   str       = hiddenTextureDataObject.ReadString();
string[] stringArr = hiddenTextureDataObject.ReadStringArray();
```

Basic read example

Putting everything together in its most basic form, it should look something like this:

```
//Construct the HiddenTextureData object
HiddenTextureData hiddenTextureDataObject = new HiddenTextureData();

//Load a texture (05:57 in the overview video)
hiddenTextureDataObject.LoadTexture("hiddendata_image.png");

//Read the data from the texture (7:18 in the overview video)
hiddenTextureDataObject.ReadDataFromTexture();

//Read some data (8:28 in the overview video)
//All value C# types have functions, including array variants (04:22 in
the reference video)
byte readByte = hiddenTextureDataObject.ReadByte();
```

Utility methods

There are also a couple of reading utility methods that you most likely won't need but they are here just in case.

If you want to set the byte index where you want to start reading from use:

```
hiddenTextureDataObject.SetHiddenDataReadPosition(index);
```

If you want to get the current byte index where you are going to read from use:

```
long byteIndex = hiddenTextureDataObject.GetHiddenDataReadPosition();
```

Settings

The setting methods help you tweak where the data is hidden in a couple of different ways.

- Tweaking which bits are used to hide it in
- Tweak what pixels are used to hide it in

Keep in mind that these settings should be the same for both reading and writing!

The hiding bits

The following method allows you to define how many bits per channel you want to use for hidden data. This can range from 0 to 8 where 0 means no pixels are used to hide data and 8 meaning all bits are used for hiding data. The least significant bits will always be used first.

Keep in mind that the higher this number, the more visible the hidden data becomes.
I recommend starting out by only using 2 alpha channel bits and increasing the amount evenly across all channels if you need to hide more data. Personally I wouldn't go higher than 2 bits per channel as any higher will definitely be very visible.

In the reference video at 5:51 I explain this setting:
Unity: How to hide data in an image (Asset reference)

```
hiddenTextureDataObject.SetDataBitsPerChannel(red, green, blue, alpha);
```

The data zones

The following methods allow you to define where and in what direction hidden data will be written to/read from. By default it will use the entire texture, starting in the top left corner and writing from left to right, top to bottom.

In the overview video at 11:00 I explain this setting:
Unity: How to hide data in an image (Asset reference)

```
hiddenTextureDataObject.AddDataZone(rect, //Regular C# rectangle
                                    axis, //Horizontal or Vertical
                                    horizontalDirection, //Left or Right
                                    verticalDirection);  //Up or Down
```

Spiral data zone

Spiral data zones don't really exist, it's an utility function that automatically adds many regular data zones with the correct settings.

In the reference video at 13:31 I explain this setting:
Unity: How to hide data in an image (Asset reference)

```
hiddenTextureDataObject.AddSpiralDataZone(rect, //Regular C# rectangle
                                          Thickness, //In pixels
                                          startCorner, //TopLeft, …
                                          clockDirection); //Clockwise,
…
```

Clearing data zones

If you want to clear all the data zones again, you can use this utility method:

```
hiddenTextureDataObject.ClearDataZones();
```

The length prefix

By default all the hidden data is prefixed with its length. This speeds up the reading process by not needing to read every single pixel but only the ones that are used to hide data.

If you are not short on data I would leave this setting on it's default "Dynamic" type. (which writes the length in the smallest required type)

However if you really want to control this prefix, or want to have none at all you can tweak with the following method.

In the reference video at 7:50 I explain this setting:
Unity: How to hide data in an image (Asset reference)

```
//Prefix type: None, Dynamic, Byte, Ushort, UInt or ULong
hiddenTextureDataObject.SetLengthPrefixType(lengthPrefixType);
```

FAQ

How can I verify that an image actually contains data from my game?

There are multiple ways to ensure your game only reads images that were written by it.

Add a couple of bytes or a string in front of all the other data. This way if you don't read these very specific bytes, the image was definitely not written by your game.

Add a checksum in front or at the end of all your data, to check with.

If you use one or both of these solutions the chance that a random image will be accepted is extremely small.

But can't people figure this out and read all the data?

Yes they can, the point is to hide it. Not to encrypt it. You can of course encrypt your data and store that in the image if you want to.

If you want to use this feature for user generated content, I wouldn't throw up any more barriers than needed. But if you intend to use it for a difficult puzzle or ARG, go ahead! (and let me know, I love those! :))

Why not use a QR code?

QR codes are very great at being scanned by a camera, and even contain some extra data to ensure it's readable when it's damaged. (compression also keeps them readable)

They are however very limited in the way they look and the data they can contain.
So if your game has camera access, a QR code is something that you can consider.

Otherwise this solution is my personal preference. But it's up to you!

I can't build the demo on Windows

The "StandaloneFileBrowser" asset requires you to build with API Compatibility Level .NET 4.x (and maybe even Mono as the scripting backend)

However I would only recommend this if you want to specifically build the demo. When building your own game, just remove all the demo files. They are not needed for this asset to work.

My question is not in the FAQ

If you have any other questions, don't hesitate to ask them on the Unity forum post for this asset!

CREDITS

This asset is created and maintained by Stijn Delaruelle (https://stijndelaruelle.com)

The StandaloneFileBrowser was created by Gökhan Gökçe & Ricardo Rodrigues
And can be found at this repository: https://github.com/gkngkc/UnityStandaloneFileBrowser

If you enjoy this asset, please consider giving it a rating & leaving a review. Thank you!